Hamilton

Version latest

Table of Contents

USER GUIDE

Slack

GitHub

REFERENCE

tryhamilton.dev

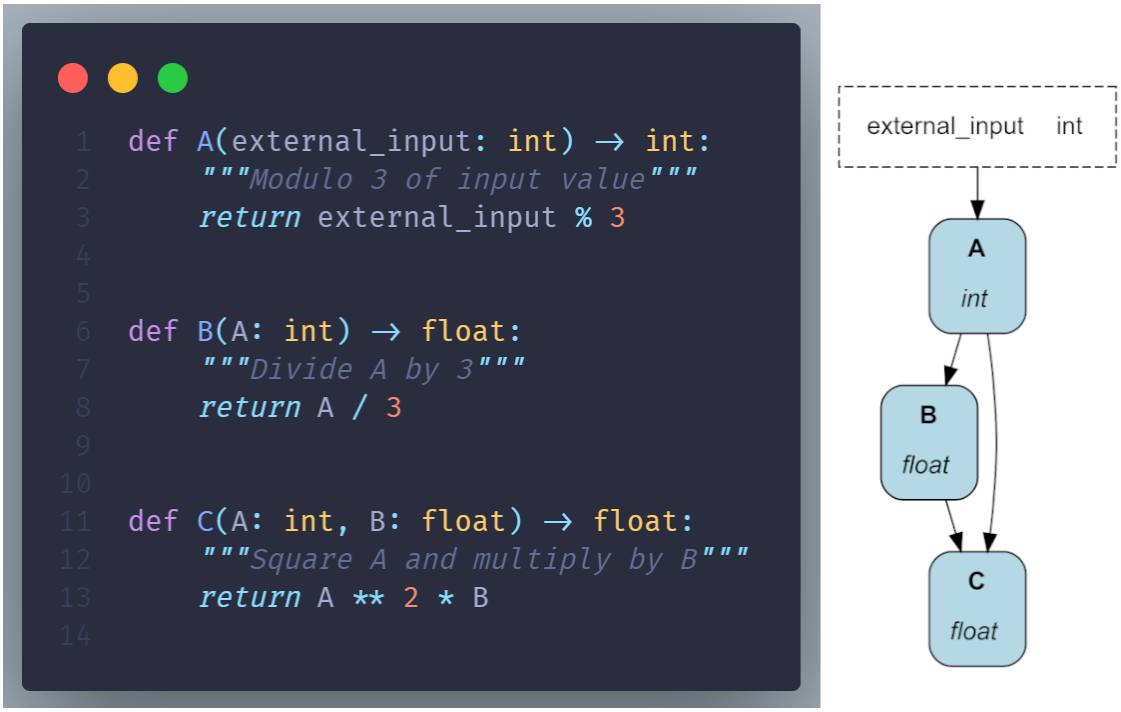
Decorators 319

- `Builder.with_adapter()`
- `Builder.with_adapters()`
- `Builder.with_cache()`
- `Builder.with_config()`
- `Builder.with_execution_manager()`
- `Builder.with_grouping_strategy()`
- `Builder.with_local_executor()`
- `Builder.with_materializers()`
- `Builder.with_modules()`
- `Builder.with_remote_executor()`

- Driver
  - `Driver`
    - `Driver.__init__()`
    - `Driver.cache`
    - `Driver.capture_constructor_telemetry()`
    - `Driver.capture_execute_telemetry()`
    - `Driver.display_all_functions()`
    - `Driver.display_downstream_of()`
    - `Driver.display_upstream_of()`
    - `Driver.execute()`
    - `Driver.export_execution()`
    - `Driver.has_cycles()`
    - `Driver.list_available_variables()`
    - `Driver.materialize()`
    - `Driver.normalize_adapter_input()`
    - `Driver.raw_execute()`
    - `Driver.validate_execution()`
    - `Driver.validate_inputs()`
    - `Driver.validate_materialization()`
    - `Driver.visualize_execution()`
    - `Driver.visualize_materialization()`
    - `Driver.visualize_path_between()`
    - `Driver.what_is_downstream_of()`
    - `Driver.what_is_the_path_between()`
    - `Driver.what_is_upstream_of()`

Welcome to Apache Hamilton

Apache Hamilton (incubating) is a general-purpose framework to write dataflows using regular Python functions. At the core, each function defines a transformation and its parameters indicates its dependencies. Apache Hamilton automatically connects individual functions into a Directed Acyclic Graph (DAG) that can be executed, visualized, optimized, and reported on. Apache Hamilton also comes with a UI to visualize, catalog, and monitor your dataflows.



The ABC of Apache Hamilton

Why should you use Apache Hamilton (incubating)?

Facilitate collaboration. By focusing on functions, Apache Hamilton avoids sprawling code hierarchy and generates flat dataflows. Well-scoped functions make it easier to add features, complete code reviews, debug pipeline failures, and hand-off projects. Visualizations can be generated directly from your code to better understand and document it. Integration with the Apache Hamilton UI allows you to track lineage, catalog code & artifacts, and monitor your dataflows.

Reduce development time. Apache Hamilton dataflows are reusable across projects and context (e.g., pipeline vs. web service). The benefits of developing robust and well-tested solutions are multiplied by reusability. Explore community-contributed dataflows in the ecosystem.

Own your platform. Apache Hamilton helps you integrate the frameworks and tools of your stack. Apache Hamilton's features are easy to extend and customize to your needs. This flexibility enables self-serve designs and ultimately reduces the risks of vendor lock-in.

Scale your dataflow. Apache Hamilton separates transformation logic from execution, allowing you to seamlessly scale via remote execution (AWS, Modal, etc.) and specialized computation engines (Spark, Ray, duckdb etc.). Apache Hamilton was battle tested under intensive enterprise data workloads.

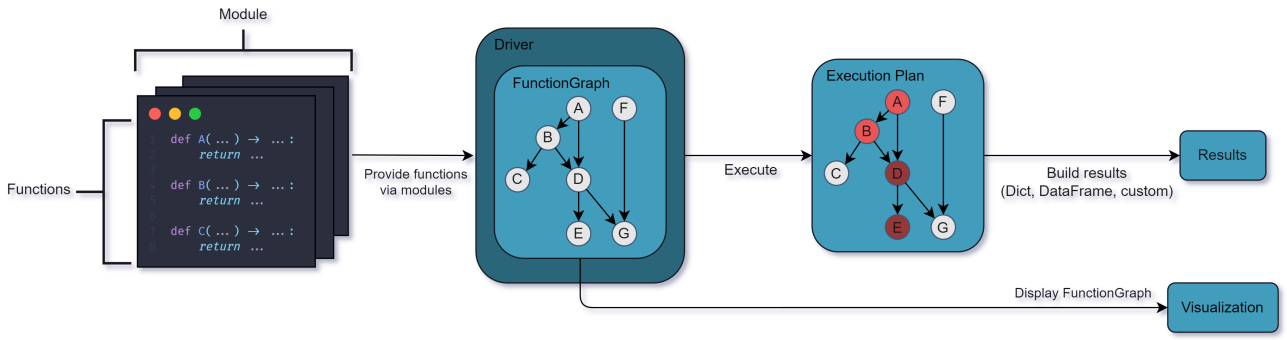
Here's a quick overview of benefits that Apache Hamilton provides as compared to other tools:

Feature	Apache Hamilton	Macro orchestration systems (e.g. Airflow)	Feast	dbt	Dask
Execute a graph of data transformations	✅	✅	❌	✅	✅
Can visualize lineage easily	✅	❌	❌	✅	✅
Can model GenerativeAI/ LLM based workflows	✅	❌	❌	❌	❌
Is a feature store	❌	❌	✅	❌	❌
Helps you structure your code base	✅	❌	❌	✅	❌
Is just a library	✅	❌	❌	❌	✅
Runs anywhere python runs	✅	❌	❌	❌	✅
Documentation friendly	✅	❌	❌	❌	❌

Feature	Apache Hamilton	Macro orchestration systems (e.g. Airflow)	Feast	dbt	Dask
Code is always unit testable	✅	❌	❌	❌	❌

Architecture Overview

The following diagram gives a simplified overview of the main components of Apache Hamilton.



Functions & Module. Transformations are regular Python functions organized into modules. Functions must be type-annotated, but hold no dependency with Apache Hamilton and can be reused outside of it.

Driver & FunctionGraph. The `Driver` will automatically assemble the `FunctionGraph` from the modules given. The `Driver` can be configured to modify and extend the execution behavior (e.g., remote execution, monitoring, webhooks, caching).

Visualization. The `FunctionGraph` can be visualized without executing code. This coupling ensures visualizations always match the code from modules.

Execution. When requesting variables, the `Driver` establishes an execution plan to only compute the required functions. Then, results are gathered and returned to the user.

Who is using Apache Hamilton?

Multiple companies are doing cool stuff with Apache Hamilton! Come chat with members of the community and the development team on Slack:

- **Wealth.com** - Async Python LLM document processing pipelines
- **Wren.ai** - Async RAG pipelines
- **Oxehealth** - Multi-modal prediction
- **PupPilot** - Async python LLM transcript processing pipelines
- **Stitch Fix** — Time series forecasting
- **British cycling** — Telemetry analysis
- **Joby** - Flight data processing
- **Transfix** - Online featurization and prediction
- **IBM** - Internal search and ML pipelines
- **Ascena** - Feature engineering

- **Adobe** - Prompt engineering research
- **Axiom Cloud** - IoT data processing
- **Oak Ridge & PNNL** - Naturf project
- **Habitat** - Time-series feature engineering
- **UK Government Digital Service** - National feedback pipeline (processing & analysis)
- **Railoify** - Orchestrate pandas code
- **Lexis Nexis** - Feature processing and lineage
- **Opendoor** - Manage PySpark pipelines
- **KI** - Feature engineering
- **Kora Money** - DS/ML Workflows
- **Capitec Bank** - Financial decisions
- **Best Egg** - Feature engineering

- **RTV Euro AGD** - General feature engineering & machine learning

Testimonials

"Apache Hamilton provides a modular and compatible framework that has significantly empowered our data science team. We've been able to build robust and flexible data pipelines with ease. The documentation is thorough and regularly updated... Even with no prior experience with the package, our team successfully migrated one of our legacy data pipelines to the Apache Hamilton structure within a month. This transition has greatly enhanced our productivity, enabling us to focus more on feature engineering and model iteration while Apache Hamilton's DAG approach seamlessly manages data lineage. I highly recommend Apache Hamilton to data professionals looking for a reliable, standardized solution for creating and managing data pipelines."

Yuan Liu

DS, Kora Financial

"How (with good software practices) do you orchestrate a system of asynchronous LLM calls, but where some of them depend on others? How do you build such a system so that it's modular and testable? At wealth.com we've selected Apache Hamilton to help us solve these problems and others. And today our product, Ester AI, an AI legal assistant that extracts information from estate planning documents, is running in production with Apache Hamilton under the hood."

Kyle Pounder

CTO, Wealth.com

"Apache Hamilton is simplicity. Its declarative approach to defining pipelines (as well as the UI to visualize them) makes testing and modifying the code easy, and onboarding is quick and painless. Since using Apache Hamilton, we have improved our efficiency of both developing new functionality and onboarding new developers to work on the code. We deliver solutions more quickly than before."

Michał Siedlaczek

Senior DS/SWE, IBM

"...The companion Apache Hamilton UI has taken the value proposition up enormously with the ability to clearly show lineage & track execution times, covering a major part of our observability needs"

Fran Boon

Director, Oxehealth.com

"Many thanks to writing such a great library. We are very excited about it and very pleased with so many decisions you've made. 🙏"

Louwrens

Software Engineer, luoautomation.com

Get Started

Welcome to Apache Hamilton's documentation!

www.tryhamilton.dev

Before diving in, we highly recommend you try Apache Hamilton in your browser at https://www.tryhamilton.dev. It allows you to:

1. run python in the browser, so you can get a feel for the basics of Apache Hamilton *without installing anything!*
2. it includes various examples that you can run and modify.
3. it represents an easy hands-on introduction to Apache Hamilton that should get you comfortable with the framework and its basic capabilities.

Get started with Apache Hamilton locally

The following section of the docs will teach you how to install Apache Hamilton and get started with your own project.

Why use Apache Hamilton?

There are many choices for building dataflows/pipelines/workflows/ETLs. Let's compare Apache Hamilton to some of the other options to help answer this question.

Comparison to Other Frameworks

There are a lot of frameworks out there, especially in the pipeline space. This section should help you figure out when to use Apache Hamilton with another framework, or in place of a framework, or when to use another framework altogether.

Let's go over some groups of "competitive" or "complimentary" products. For a basic overview, see the product matrix on the homepage.

Orchestration Systems

Examples include:

- Airflow
- Metaflow
- Luigi
- dbt

Apache Hamilton is not, in itself a macro, i.e. high level, task orchestration system. While it does orchestrate functions, and the DAG abstraction is very powerful, it does not provision compute, or schedule long-running jobs. Apache Hamilton works well in conjunction with these macro systems. Apache Hamilton provides the capabilities of fine-grained lineage, highly readable code, and self-documenting pipelines, which many of these systems lack.

Apache Hamilton can be used within any python orchestration system in the following ways:

1. *Hamilton DAGs can be called within orchestration system tasks.* See the Apache Hamilton + Airflow example. The integration is generally trivial – all you have to do is call out to the hamilton library within your task. If your orchestrator supports python, then you're good to go. Some pseudocode (if your orchestrator handles scripts like airflow):

```
#my_task.py
import hamilton
import my_transformations
dr = hamilton.driver.Driver({}, my_functions)
output = dr.execute(['final_var'], inputs=...)
do_something_with(output)
```

2. *Hamilton DAGs can be broken up to run as components within an orchestration system.* With the ability to include overrides, you can run the DAG on each task, overloading the outputs of the last task + any static inputs/configuration, and pass it into the next task. This is more of a manual/power-user feature. Some pseudocode:

```
#my_task.py
import hamilton
import my_functions
prior_inputs = load_relevant_task_results()
desired_outputs = ['final_var_1', 'final_var_2']
inputs = my_inputs
dr = hamilton.driver.Driver({}, my_functions)
output = dr.execute(
   desired_outputs,
   inputs=inputs,
```

```
    overrides=prior_inputs)
save_for_later(output)
```

Feature Stores

Examples include:

- Hopsworks
- Feast
- Tecton

One can think of Apache Hamilton as a being your "feature definition store", where "store" is code + git. While it does not provide all the capabilities of a standard feature store, it provides a source of truth for the code that generated the features, and can be run in a portable method. *So*, if your desire is just to be able to run the same code in different environments, and have an online/offline store of features, you can use hamilton both to save the features offline, and generate features online on the fly.

See the feature engineering example for more possibilities, as well as blogs on the feature topic.

Note that in small cases, you probably don't need a true feature store – recomputing derived features in an ETL and online can be very efficient, as long as you have some database to look values up (or have them passed in).

Also note that joins and aggregations can get tricky. We often recommend using our "polymorphic function definition" i.e. functions decorated with `@config.when`, to either load up the non-online-friendly features from a feature store or do an external lookup to simulate an online join.

We expect Apache Hamilton to play a prominent role in the way feature stores work in the future.

Data Science Ecosystems/ML platforms

Examples include:

- Kedro
- Domino Data Labs
- Dataiku
- SageMaker
- Google Cloud Vertex AI Platform
- etc.

We've kind of grouped a whole suite of platforms into the same bucket here. These tend to have a lot of capabilities all related to ML. Apache Hamilton can be used in conjunction with these platforms in a variety of ways. For example, you can use Apache Hamilton to generate features for a model that you train in one of these platforms. Or you can use Apache Hamilton to generate a model using the platform's compute, and then save the model to the platform's registry.

Registries / Experiment Tracking

Examples include:

- MLflow
- Weights and Biases
- DVC

Most pipelines have a "reverse ETL problem" – they need to get the results of the pipeline into a some sort of datastore or registry. Apache Hamilton can be used in conjunction with these tools as the glue code that helps everything work together. For example, you can use Apache Hamilton to generate a model and then store metrics computed by Apache Hamilton to one of these "destinations".

There are three main ways to integrate with these tools:

- inside a function that Apache Hamilton orchestrates
- outside Apache Hamilton (e.g. in a script that calls Apache Hamilton)
- using "materializers" (see materializers) (see this blog).

See this ML reference post for examples of how to use Apache Hamilton with these tools.

Python Dataframe/manipulation Libraries

Examples include:

- pandas
- dask
- modin
- polars
- duckdb

Apache Hamilton works with any python dataframe/manipulation oriented libraries. See our examples folder to see how to use Apache Hamilton with these libraries.

Python “big data” systems

The following systems are ones that you would resort to using when wanting to scale up your data processing.

Examples include:

- dask
- ray
- pyspark
- pandas-on-spark

These all provide capabilities to either (a) express and execute computation over datasets in python or (b) parallelize it. Often both. Apache Hamilton has a variety of integrations with these systems. The basics is that Apache Hamilton can make use of these systems to execute the DAG using the GraphAdapter abstraction and Lifecycle Hooks.

See our examples folder to see how to use Apache Hamilton with these systems.

Installing hamilton is easy!

Install

Apache Hamilton is a lightweight framework with a variety of extensions/plugins. To get started, you’ll need the following:

- `python >= 3.10`
- `pip`

For help with python/pip/managing virtual environments see the python docs.

Installing with pip

Apache Hamilton is published on pypi under `sf-hamilton`. To install, run:

```
pip install sf-hamilton
```

To use the DAG visualization functionality, instead install with

```
pip install sf-hamilton[visualization]
```

Note: for visualization you may additionally need to install graphviz externally – see graphviz for instructions on the correct way for your operating system.

Installing with conda

Apache Hamilton is also available on conda if you prefer:

```
conda install -c hamilton-opensource sf-hamilton
```

Installing from source

You can also download the code and run it from the source.

```
git clone https://github.com/apache/hamilton.git
cd hamilton
pip install -e .
```

Your First Dataflow

Let's get started with a dataflow that computes statistics on a time-series of marketing spend.

We're jumping in head-first. If you want to start with an overview, skip ahead to Concepts.

Note

You can follow along in the examples directory of the hamilton repo. We highly recommend forking the repo and playing around with the code to get comfortable.

Write transformation functions

Create a file `my_functions.py` and add the following two functions:

```
import pandas as pd

def avg_3wk_spend(spend: pd.Series) -> pd.Series:
    """Rolling 3 week average spend."""
    return spend.rolling(3).mean()

def acquisition_cost(avg_3wk_spend: pd.Series, signups: pd.Series) -
> pd.Series:
    """The cost per signup in relation to a rolling average of
spend."""
    return avg_3wk_spend / signups
```

An astute observer might ask the following questions:

1. **Why do the parameter names clash with the function names?** This is core to how hamilton works. It utilizes dependency injection to create a DAG of computation. Parameter names tell the framework where your function gets its data.

2. **OK, if the parameter names determine the source of the data, why have we not defined defined `spend` or `signups` as functions?** This is OK, as we will provide this data as an input when we actually want to materialize our functions. The DAG doesn't have to be complete when it is compiled.

3. **Why is there no main line to call these functions?** Good observation. In fact, we never will call them (directly)! This is one of the core principles of Apache Hamilton. You write individual transforms and the rest is handled by the framework. More on that next.

4. **The functions all output pandas series. What if I don't want to use series?** You don't have to! Apache Hamilton is not opinionated on the data type you use. The following are all perfectly valid as well (and we support dask/spark/ray/other distributed frameworks).

Let's add a few more functions to our `my_functions.py` file:

```
def spend_mean(spend: pd.Series) -> float:

"""Shows function creating a scalar. In this case it computes the
mean of the entire column."""
    return spend.mean()

def spend_zero_mean(spend: pd.Series, spend_mean: float) ->
pd.Series:
    """Shows function that takes a scalar. In this case to zero mean
spend."""
    return spend - spend_mean

def spend_std_dev(spend: pd.Series) -> float:
    """Function that computes the standard deviation of the spend
column."""
    return spend.std()

def spend_zero_mean_unit_variance(spend_zero_mean: pd.Series,
spend_std_dev: float) -> pd.Series:

"""Function showing one way to make spend have zero mean and unit
variance."""
    return spend_zero_mean / spend_std_dev
```

Let's give these functions a spin!

Run your dataflow

To actually run the dataflow, we'll need to write a driver. Create a `my_script.py` with the following contents:

```
import logging
import sys

import pandas as pd

# We add this to speed up running things if you have a lot in your
python environment.
from hamilton import registry; registry.disable_autoload()
from hamilton import driver, base
import my_functions  # we import the module here!


logger = logging.getLogger(__name__)
logging.basicConfig(stream=sys.stdout)

if __name__ == '__main__':
    # Instantiate a common spine for your pipeline
    index = pd.date_range("2022-01-01", periods=6, freq="w")
    initial_columns = {  # load from actuals or wherever -- this is
our initial data we use as input.
        # Note: these do not have to be all series, they could be
scalar inputs.
        'signups': pd.Series([1, 10, 50, 100, 200, 400],
index=index),
        'spend': pd.Series([10, 10, 20, 40, 40, 50], index=index),
    }
    dr = (
      driver.Builder()
        .with_config({})  # we don't have any configuration or
invariant data for this example.
        .with_modules(my_functions)
# we need to tell hamilton where to load function definitions from
        .with_adapters(base.PandasDataFrameResult())  # we want a
pandas dataframe as output
        .build()
    )
    # we need to specify what we want in the final dataframe (these
could be function pointers).
    output_columns = [
        'spend',
        'signups',
        'avg_3wk_spend',
        'acquisition_cost',
    ]
    # let's create the dataframe!
    df = dr.execute(output_columns, inputs=initial_columns)
```

```
    # `pip install sf-hamilton[visualization]` earlier you can also
do
    # dr.visualize_execution(output_columns,'./my_dag.png', {})
    print(df)
```

Run the script with the following command:

```
python my_script.py
```

And you should see the following output:

```
             spend  signups  avg_3wk_spend  acquisition_cost
2022-01-02     10        1            NaN            10.000
2022-01-09     10       10            NaN             1.000
2022-01-16     20       50      13.333333             0.400
2022-01-23     40      100      23.333333             0.400
2022-01-30     40      200      33.333333             0.200
2022-02-06     50      400      43.333333             0.125
```

Not only is your spend to signup ratio decreasing exponentially (your product is going viral!), but you've also successfully run your first Apache Hamilton Dataflow. Kudos!

See, wasn't that quick and easy?

Note: if you're ever like "why are things taking a while to execute?", then you might have too much in your python environment and Apache Hamilton is auto-loading all the extensions. You can disable this by setting the environment variable `HAMILTON_AUTOLOAD_EXTENSIONS=0` or programmatically via `from hamilton import registry; registry.disable_autoload()` - for more see Extension autoloading.

Learning Resources

Several channels are available to get started with Apache Hamilton, learn advanced usage, and participate in the latest feature development.

📒 User Guide Documentation

The user guide gives a complete overview of Apache Hamilton's features.

📚 Reference Documentation

The reference documentation details Apache Hamilton's public API.

🌐 Ecosystem & Integrations

The ecosystem page lists all built-in integrations (pandas, Polars, Spark, etc.) and external community resources. Find reusable dataflows, blog posts, and video tutorials there.

✍️ tryhamilton.dev

The tryhamilton.dev website provides interactive tutorials in-browser to learn specific Apache Hamilton concepts.

👋 Slack

The Slack channel is the ideal place to ask questions, request features, and give feedback.

📣 Talks & Videos

See the ecosystem page for links to video content and conference talks.

- 2024-02 Apache Hamilton Meet-up for February
  - Recording
  - [Slides](https://github.com/skrawcz/talks/files/14351139/Apache Hamilton.February.2024.Meetup.pdf)
- 2023-12 Why you should build your GenAI/LLM apps using Apache Hamilton. AICamp End of Year in SF
  - Recording
  - [Slides](https://github.com/skrawcz/talks/files/13666470/Why.you.should.build.your.GenAI_LLM.apps.using.Apache Hamilton.pdf)
- 2023-12 Bridging Classic ML Pipelines with the World of LLMs. PyData Global
  - Slides
- 2023-11 Apache Hamilton: Natively bringing software engineering best practices to python data transformations. Scale by the Bay.
  - Recording
  - Slides

- 2023-09 Apache Hamilton: Natively bringing software engineering best practices to python data transformations. Bay Area Python Interest Group (BAYPIGgies)
  - [Slides](https://github.com/skrawcz/talks/files/12785978/BayPIGgies_.Apache Hamilton.Talk.pdf)
- 2023-08 dbt + Apache Hamilton: Enabling you to maintain complex Python within dbt models. MDSFest'23
  - Recording
  - Slides
- 2023-06 Apache Hamilton: an OS tool to add to your LLM App toolbelt. LLM Avalanche.
  - Slides
- 2023-06 Feature Engineering with Apache Hamilton: Portability & Lineage. Budapest ML Forum June 2023
  - Slides
- 2023-06 British Cycling Data Platform in Python. Manchester PyData Meetup
  - Slides
  - Co-presented with Peter Robinson, and Murray Tait.
- 2023-04 Lightweight Lineage with Apache Hamilton. PyData Seattle
  - [Slides](https://github.com/skrawcz/talks/files/11399972/PyData-Seattl-Lightning-Talk-2023-Lighweight-Lineage-with-Apache Hamilton.pdf)
- 2023-01 Apache Hamilton: Natively bringing software engineering best practices to python data transformations. AI Camp Meetup San Jose
  - Slides
- 2022-10 Apache Hamilton: an open source, declarative, micro-framework for clean & robust feature transform code in Python. Feature Store Summit
  - Event
  - [Slides](https://github.com/skrawcz/talks/files/9759661/FS.Summit.2022.-.Apache Hamilton.pdf)

- 2022-09 Apache Hamilton: enabling software engineering best practices for data transformations via generalized dataflow graphs. DEco - First International Workshop on Data Ecosystems
  - Event
  - Slides
- 2022-09 Apache Hamilton: a modular open source declarative paradigm for high level modeling of dataflows. CDMS - First International Workshop on Composable Data Management Systems
  - Event
  - Slides
  - Paper
- 2022-08 Apache Hamilton: A Python Micro-Framework for tidy scalable Pandas. Scalable Pandas Meetup
  - Recording
  - [Slides](https://github.com/skrawcz/talks/files/9428705/Apache Hamilton. %40.Ponder.Pandas.meetup.pdf)
- 2022-08 Scalable feature engineering with Apache Hamilton on Ray. Ray Summit
  - [Slides](https://github.com/skrawcz/talks/files/9411082/Submitted.Slides.-.Ray.Summit_.Scalable.feature.engineering.with.Apache Hamilton.on.Ray.pdf)
- 2022-07 Apache Hamilton: A Python Micro-Framework for Data / Feature Engineering. MLOPsWorld Bay Area
  - Slides
- 2022-05 Apache Hamilton: a python micro-framework for data / feature engineering at Stitch Fix. AICamp
  - Recording
  - [Slides](https://github.com/skrawcz/talks/files/8691633/AICamp.Apache Hamilton.Presentation.pdf)
- 2022-02 [Open Source] Apache Hamilton, a micro framework for creating dataframes, and its application at Stitch Fix. Apply(Meetup)
  - Event.

    - Recording
    - Slides
- 2021-12 Apache Hamilton an open source micro framework for creating dataframes. SF Python Meetup
    - Recording
    - Slides

📰 External Blogs

For external resources including blogs, see the ecosystem page. Here are some notable blog posts about Apache Hamilton:

- 2024-03 RAG: ingestion and chunking using Apache Hamilton and scaling to Ray, Dask, or PySpark
- 2024-02 A command line tool to improve your development workflow
- 2024-02 Monthly Meetup Recap and office hours
- 2024-02 Using IPython Jupyter Magic commands to improve the notebook experience
- 2024-02 Building a lightweight experiment manager
- 2024-01 Customizing Apache Hamilton's Execution with the new Lifecycle API
- 2024-01 How well-structured should your data code be?
- 2024-01 From Dev to Prod: a ML Pipeline Reference Post
- 2023-12 Winning over hearts and minds at work: ADKAR my favorite change management approach
- 2023-11 🚀 We're launching the Apache Hamilton Dataflow Hub!
- 2023-10 Separate data I/O from transformation – your future self will thank you.
- 2023-09 Retrieval augmented generation (RAG) with Streamlit, FastAPI, Weaviate, and Apache Hamilton!
- 2023-09 LLMOps: Production prompt engineering patterns with Apache Hamilton
- 2023-09 Feature Engineering with Apache Hamilton
- 2023-08 Expressing PySpark Transformations Declaratively with Apache Hamilton
- 2023-08 Containerized PDF Summarizer with FastAPI and Apache Hamilton

- 2023-08 Dynamic DAGs: Counting Stars with Apache Hamilton
- 2023-08 Featurization: Integrating Apache Hamilton with Feast
- 2023-07 Simplify Prefect Workflow Creation and Maintenance with Apache Hamilton
- 2023-07 Building a maintainable and modular LLM application stack with Apache Hamilton
- 2023-06 Simplify Airflow DAG Creation and Maintenance with Apache Hamilton
- 2023-05 Lineage + Apache Hamilton in 10 minutes
- 2022-11 Apache Hamilton + DBT in 5 minutes
- 2022-07 Tidy production pandas with Apache Hamilton
- 2022-06 Developing Scalable Feature Engineering DAGs with Metaflow & Apache Hamilton
- 2022-05 Apache Hamilton backstory and intro post on TDS
- 2022-05 Apache Hamilton + Pandas in five minutes
- 2022-05 Iterating with Apache Hamilton in a Notebook

🎙️ Podcasts

- 2024-03 Apache Hamilton mention in Real Python, about ipython magic command post
- 2023-06 Exploring the Intersection of DAGs, ML Code, and Complex Code Bases: An Elegant Solution Unveiled with Stefan Krawczyk of DAGWorks
- 2022-08 S01 E08 - MLOps Week 8: The MLOps Mindset with Stefan Krawczyk
- 2022-04 MLOps dla 100 data scientistów (in Polish)
- 2021-09 Aggressively Helpful Platform teams

Contributing

We are open contributions big and small. See our contributing guidelines.

We also operate under a Code of Conduct, and expect contributors to do the same.

License

Apache Hamilton is released under the Apache 2.0 License.

Usage analytics & data privacy

By default, when using Apache Hamilton, it collects anonymous usage data to help improve Apache Hamilton and know where to apply development efforts.

We capture three types of events: one when the *Driver* object is instantiated, one when the *execute()* call on the *Driver* object completes, and one for most *Driver* object function invocations. No user data or potentially sensitive information is or ever will be collected. The captured data is limited to:

- Operating System and Python version
- A persistent UUID to indentify the session, stored in ~/.hamilton.conf.
- Error stack trace limited to Apache Hamilton code, if one occurs.
- Information on what features you're using from Apache Hamilton: decorators, adapters, result builders.
- How Apache Hamilton is being used: number of final nodes in DAG, number of modules, size of objects passed to *execute()*, the name of the Driver function being invoked.

Else see Telemetry for how to disable telemetry.

Otherwise we invite you to inspect telemetry.py for details.

Concepts

Now that you're familiar with the basics and have run your own dataflow, let's dive into the concepts that makes Apache Hamilton unique and powerful.

Glossary

Before we dive into the concepts, let's clarify the terminology we'll be using:

Directed Acyclic Graph (DAG)	A directed acyclic graph is a computer science/mathematics term for representing the world with "nodes" and "edges", where "edges" only flow in one direction. It is called a graph because it can be drawn and visualized.
Dataflow	The organization of functions and dependencies. This is a DAG – it's directed (one function is running before the other), acyclic, (there are no cycles, i.e., no function runs before itself), and a graph (it is easily naturally represented by nodes and edges) and can be represented visually. See Functions, nodes & dataflow.
Node | Hamilton node | Transform	A single step in the dataflow DAG representing a computation – usually 1:1 with functions but decorators break that pattern – in which case multiple transforms trace back to a single function. See Functions, nodes & dataflow.
Function | Python function | Hamilton function | Node definition	A Python function written by a user to create a single node (in the standard case) or many (using function modifiers). See Functions, nodes & dataflow.
Module | Python module	Python code organized into a `.py` file. These are natural groupings of functions that turn to a set of nodes. See Code Organization for more details.

Driver | Hamilton Driver	An object that loads Python modules to build a dataflow. It is responsible for visualizing and executing the dataflow. See Driver.
script | runner | driver code	The piece of code where you create the Driver and execute the dataflow to get results.
Config	Data that dictates the way the DAG is constructed. See Driver.
Function modifiers | Decorators	A function that modifies how your Hamilton function is compiled into a Hamilton node. See Function modifiers.

Functions, nodes & dataflow

On this page, you'll learn how Apache Hamilton converts your Python functions into nodes and then creates a dataflow.

Functions

Apache Hamilton requires you to write your code using functions. To get started, you simply need to:

- Annotate the type of the function parameters and return value.
- Specify the function dependencies with the parameter names.
- Store your code in Python modules (`.py` files).

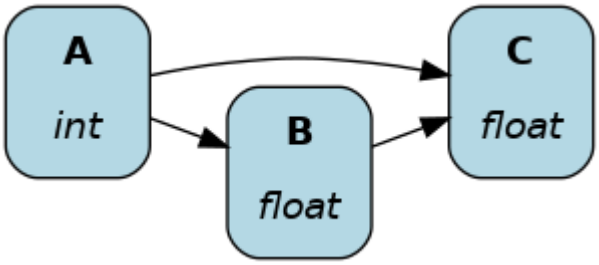
Since your code doesn't depend on special "Apache Hamilton code", you can reuse it however you want!

Specifying dependencies

In Apache Hamilton, you define dependencies by matching parameter names with the names of other functions. Below, the function name and return type `A() -> int` match the parameter `A: int` found in functions `B()` and `C()`.

```
def A() -> int:
  """Constant value 35"""
```

```
  return 35

def B(A: int) -> float:
    """Divide A by 3"""
    return A / 3

def C(A: int, B: float) -> float:
    """Square A and multiply by B"""
    return A**2 * B
```



The figure shows how Apache Hamilton automatically assembled the functions `A()`, `B()`, and `C()`.

Helper function

You can prefix a function name with an underscore (`_`) to prevent it from being included in a dataflow. Below, `A()` and `B()` are part of the dataflow, but `_round_three_decimals()` isn't.

```
def _round_three_decimals(value: float) -> float:
    """Round value by 3 decimals"""
    return round(value, 3)

def A(external_input: int) -> int:
    """Modulo 3 of input value"""
    return external_input % 3

def B(A: int) -> float:
    """Divide A by 3"""
    b = A / 3
    return _round_three_decimals(b)
```

Function naming tips

Apache Hamilton strongly agrees with the Zen of Python #2: "Explicit is better than implicit". Meaningful function names help document what functions do, so don't shy away from longer names. If you were to come across a function named `life_time_value` versus `ltv` versus `l_t_v`, which one is most obvious? Remember your code usually lives a lot longer than you ever think it will.

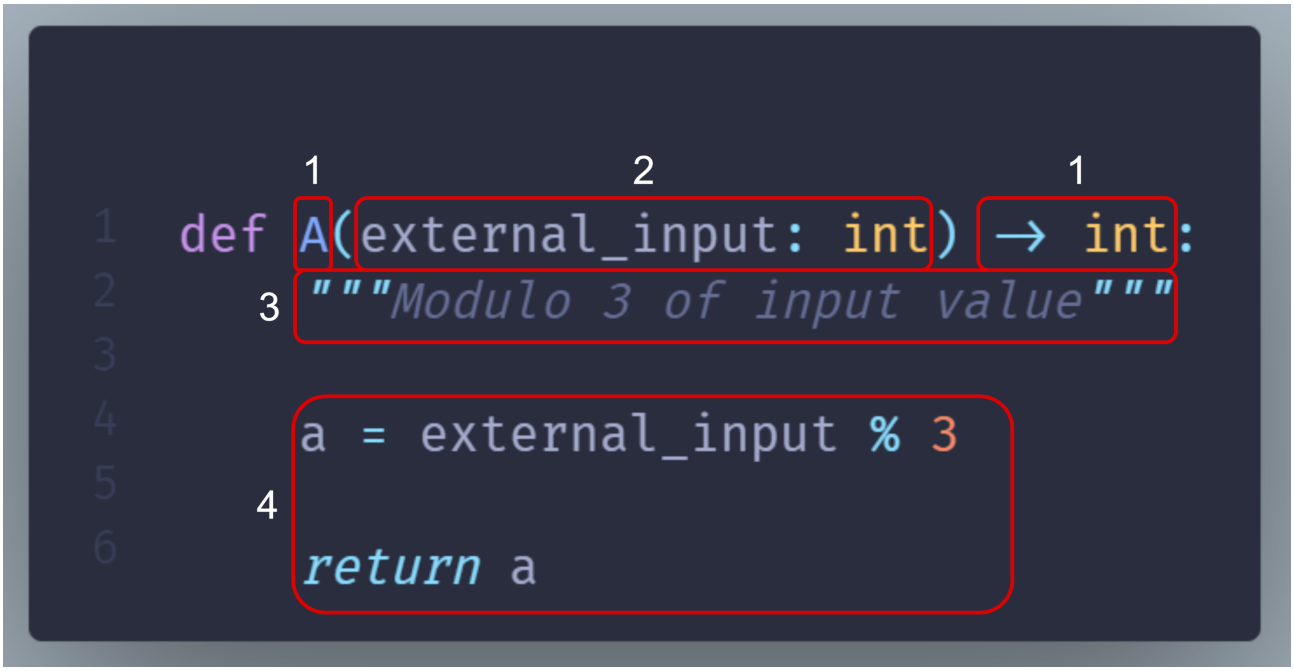
Unlike the common practice of including meaningful verbs in function names (e.g., `get_credentials()`, `statistical_test()`), with Apache Hamilton, the function name should more closely align with nouns. That's because the function name determines the node name and how data will be queried. Therefore, names that describe the node result rather than its action may be more readable (e.g., `credentials()`, `statistical_results()`).

Nodes

A node is a single "operation" or "step" in a dataflow. Apache Hamilton users write Python *functions* that Apache Hamilton converts into *nodes*. User never directly create nodes.

Anatomy of a node

The following figure and table detail how a Python function maps to a Hamilton node.



id	Function components	Node components
1	Function name and return type annotation	Node name and type
2	Parameter names and type annotations	Node dependencies
3	Docstring	Description of the node return value

id	Function components	Node components
4	Function body	Implementation of the node

Since functions almost always map to nodes 1-to-1, the two terms are often used interchangeably. However, there are exceptions that we'll discuss later in this guide.

Dataflow

From a collection of nodes, Apache Hamilton automatically assembles the dataflow. For each node, it creates edges between itself and its dependencies, resulting in a dataflow (or a graph in more mathematical terms).

From the user perspective, you give Apache Hamilton a Python module containing your functions and it will generate your dataflow! This is a key difference with popular orchestration / pipeline / workflow frameworks (Airflow, Kedro, Prefect, VertexAI, SageMaker, etc.)

How other frameworks build graphs

In most frameworks, you first define nodes / steps / tasks / components. Then, you need to create your dataflow by explicitly specifying the relationship between each node.

Readability

In that case, the code for `step A` doesn't tell you how it relates `step B` or the broader dataflow. Apache Hamilton solves this problem by tying functions, nodes, and dataflow definitions in a single place. The ratio of reading to writing code can be as high as 10:1, especially for complex dataflows, so optimizing for readability is high-value.

Maintainability

Typically, editing a dataflow (new feature, debugging, etc.) alters both what a **node** does and how the **dataflow** is structured. Consequently, changes to `step A` require you to manually ensure consistent edits to the definition of dataflows, which is likely in another file. In enterprise settings, it can become difficult to discover and track every location where `step A` is used (potentially 10s or 100s of pipelines), increasing the likelihood of breaking changes. Apache Hamilton avoids this problem entirely because changes to the node definitions, and thus the dataflow, will propagate to all places the code is used. This greatly improves maintainability and development speed by facilitating code changes.

Recap

- Users write Python functions into modules with proper naming and typing
- Helper functions use an underscore prefix (e.g., `_helper()`)
- Apache Hamilton converts functions into nodes
- Apache Hamilton automatically assembles nodes into a dataflow

Next step

So far, we learned how to write Apache Hamilton code for our dataflow. Next, we'll explore how we can effectively

1. Convert a Python module into dataflow
2. Visualize a dataflow
3. Execute a dataflow
4. Gather and store results of a dataflow

Driver

Once you defined your dataflow in a Python module, you need to create a Hamilton Driver to execute it. This page details the Driver basics, which include:

1. Defining the Driver
2. Visualizing the dataflow
3. Executing the dataflow

For this page, let's pretend we defined the following module `my_dataflow.py`:

```
# my_dataflow.py
def A() -> int:
    """Constant value 35"""
    return 35

def B(A: int) -> float:
    """Divide A by 3"""
    return A / 3
```

```
def C(A: int, B: float) -> float:
    """Square A and multiply by B"""
    return A**2 * B
```

Define the Driver

First, you need to create a `driver.Driver` object. This is done by passing Python modules to the `driver.Builder()` object along other configurations and calling `.build()`.

The most basic Driver is built like this:

```
# run.py
from hamilton import driver
import my_dataflow  # <- module containing functions to define
dataflow

# variable `dr` is of type `driver.Driver`
# it is created by a `driver.Builder` object
dr = driver.Builder().with_modules(my_dataflow).build()
```

The `.build()` method will fail if the definition found in `my_dataflow` is invalid (e.g., type mismatch, missing annotations) allowing you to fix issues and iterate quickly.

The `Driver` is defined in the context you intend to run, separately from your dataflow module. It can be in a script, notebook, server, web app, or anywhere else Python can run. As a convention, most Apache Hamilton code examples use a script named `run.py`.

Visualize the dataflow

Once you successfully created your Driver, you can visualize the entire dataflow with the following:

```
# run.py
from hamilton import driver
import my_dataflow

dr = driver.Builder().with_modules(my_dataflow).build()
dr.display_all_functions("dag.png")  # outputs a file dag.png
dr.display_all_functions()  # to view directly in a notebook
```

Dataflow visualizations are useful for documenting your project and quickly making sense of what a dataflow does (see Visualization).

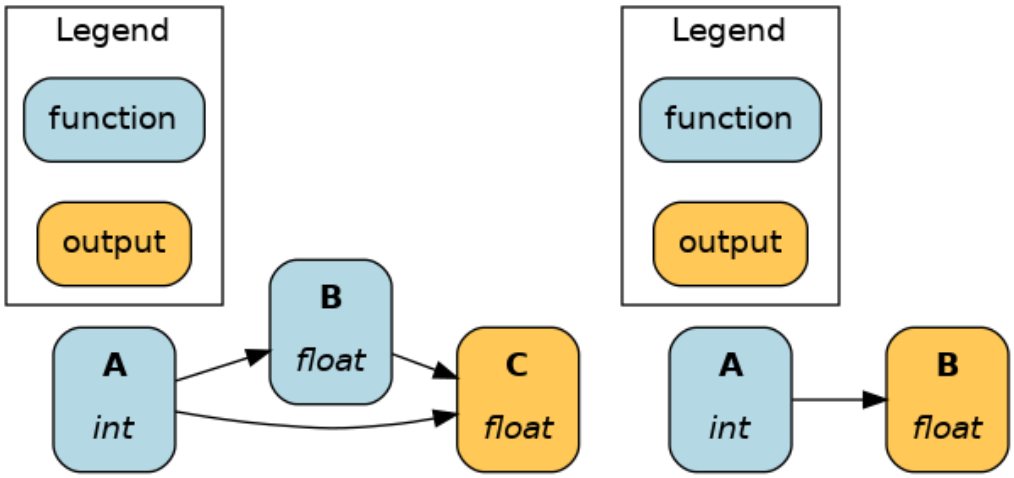
Execute the dataflow

From the Driver, you can request the value of specific nodes by calling `dr.execute(final_vars=[...])`, where `final_vars` is a list of node names. By default, results are returned in a dictionary with `{node_name: result}`.

The following requests the node `C` and visualizes the dataflow execution:

```
# run.py
from hamilton import driver
import my_dataflow

dr = driver.Builder().with_modules(my_dataflow).build()
dr.visualize_execution(["C"], "execute_c.png")
results = dr.execute(["C"])

print(results["C"])  # access results dictionary
```

The Driver automatically determines the minimum required path to compute requested nodes. See the respective outputs for `dr.visualize_execution(["C"])` and `dr.visualize_execution(["B"])`:



Development tips

With Apache Hamilton, development time is mostly spent writing functions for your dataflow in a Python module. Rebuilding the Driver and visualizing your dataflow as you make changes helps iterative development. Find below two useful development workflows.

With a Python module

One approach is to define the dataflow and the Driver in the same file (e.g., `my_dataflow.py`). Then, you can execute it as a script with `python my_dataflow.py` to rebuild the Driver and visualize your dataflow. This ensures your dataflow definition remains valid as you make changes.

For example:

```
# my_dataflow.py
def A() -> int:
    """Constant value 35"""
    return 35

# ... more functions

# is True when calling `python my_dataflow.py`
if __name__ == "__main__":
    from hamilton import driver
    # __main__ refers to the file itself
    # and yes, a file can import itself as a module!
    import __main__

    dr = driver.Builder().with_modules(__main__).build()
    dr.display_all_functions("dag.png")
    dr.execute(["C"])
```

With a Jupyter notebook

Another approach is to define the dataflow in a module (e.g., `my_dataflow.py`) and reload the Driver in a Jupyter notebook. This allows for a more interactive experience when you want to inspect the results of functions as you're developing.

By default, Python only imports a module once and subsequent `import` statements don't reload the module. We reload our imported module with `importlib.reload(my_dataflow)` and rebuild the Driver as we make changes to our dataflow.

```
# notebook.ipynb
# %%cell 1
import importlib
from hamilton import driver
import my_dataflow

# %%cell 2
# this will reload an already imported module
importlib.reload(my_dataflow)

# rebuild the `Driver` with the reloaded module and execute again
dr = driver.Builder().with_modules(my_dataflow).build()
```

```
dr.display_all_functions("dag.png")
results = dr.execute(["C"])

# %%cell 3
# do something with results
print(results["C"])
```

Learn other Jupyter development tips on the page Jupyter notebooks.

Recap

- The Driver automatically assembles a dataflow from Python modules
- The Driver visualizes the dataflow created from your code
- Functions are executed by requesting nodes to driver `.execute()`

Next step

Now, you know the basics of authoring and executing Apache Hamilton dataflows! We encourage you to:

- Write some code with our interactive tutorials
- Kickstart your project with community resources

The next **Concepts** pages cover notions to write more expressive and powerful code. If you feel stuck or constrained with the basics, it's probably a good time to (re)visit them. They include:

- Materialization: interact with external data sources
- Function modifiers: write expressive dataflows without repeating code
- Builder: how to customize your Driver

Visualization

After assembling the dataflow, several visualization features become available to the Driver. Apache Hamilton dataflow visualizations are great for documentation because they are directly derived from the code.

On this page, you'll learn:

- the available visualization functions
- how to answer lineage questions
- how to apply a custom style to your visualization

For this page, we'll assume we have the following dataflow and Driver:

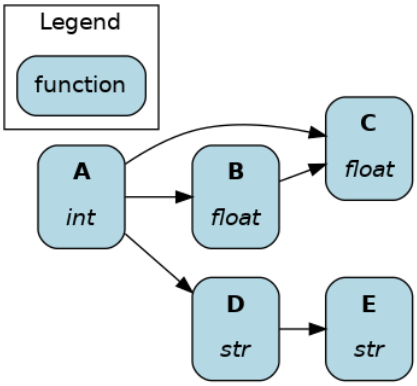
```
# my_dataflow.py
def A() -> int:
    """Constant value 35"""
    return 35

def B(A: int) -> float:
    """Divide A by 3"""
    return A / 3

def C(A: int, B: float) -> float:
    """Square A and multiply by B"""
    return A**2 * B

def D(A: int) -> str:
    """Say `hello` A times"""
    return "hello "

def E(D: str) -> str:
    """Say hello*A world"""
    return D + "world"

# run.py
from hamilton import driver
import my_dataflow

dr = driver.Builder().with_modules(my_dataflow).build()
```

Available visualizations

View full dataflow

During development and for documentation, it's most useful to view the full dataflow and all nodes.

```
dr.display_all_functions(...)
```

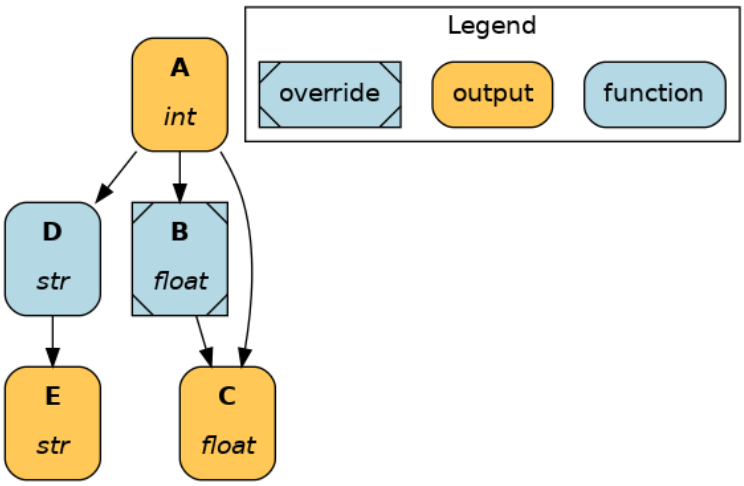


View executed dataflow

Visualizing exactly which nodes were executed is more helpful than viewing the full dataflow when logging driver execution (e.g., ML experiments).

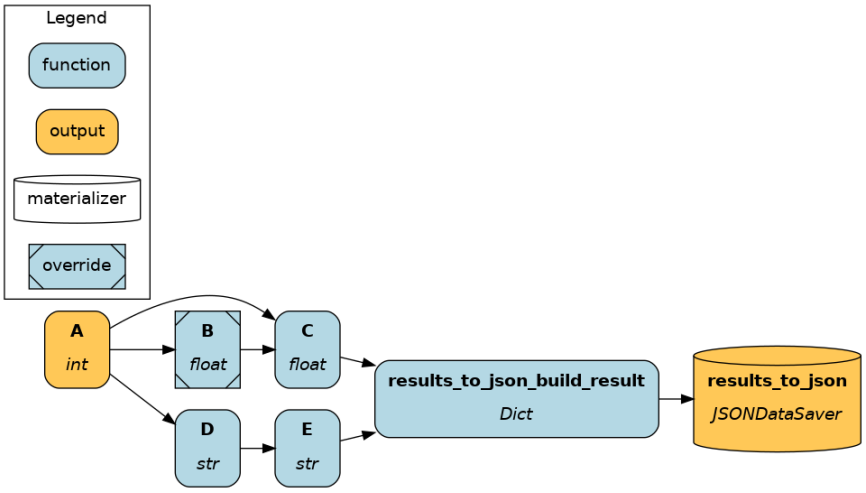
You should produce the visualization before executing the dataflow. Otherwise, the figure won't be generated if the execution fails first.

```
# pull variables to ensure .execute() and
# .visualize_execution() receive the same
# arguments
final_vars = ["A", "C", "E"]
inputs = dict()
overrides = dict(B=36.1)

dr.visualize_execution(
    final_vars=final_vars,
    inputs=inputs,
    overrides=overrides,
)
dr.execute(
    final_vars=final_vars,
    inputs=inputs,
    overrides=overrides,
)
```



An equivalent method is available if you're using materialization.

```
materializer =  to.json(
    path="./out.json",
    dependencies=["C", "E"],
    combine=base.DictResult(),
    id="results_to_json",
)
additional_vars = ["A"]
inputs = dict()
overrides = dict(B=36.1)

dr.visualize_materialization(
    materializer,
    additional_vars=additional_vars,
    inputs=inputs,
    overrides=dict(B=36.1),
    output_file_path="dag.png"
)
dr.materialize(
    materializer,
    additional_vars=additional_vars,
    inputs=inputs,
    overrides=dict(B=36.1),
)
```



Learn more about Materialization.

View node dependencies

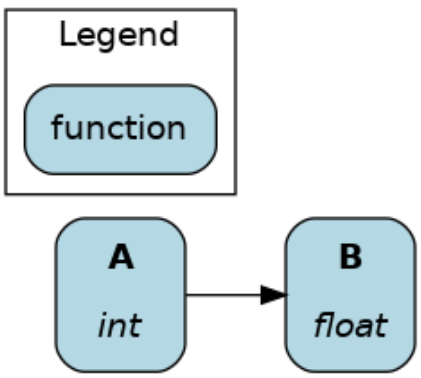
Representing data pipelines, ML experiments, or LLM applications as a dataflow helps reason about the dependencies between operations. The Hamilton Driver has the following utilities to select and return a list of nodes (to learn more Lineage + Apache Hamilton):

- `.what_is_upstream_of(*node_names: str)`
- `.what_is_downstream_of(*node_names: str)`
- `.what_is_the_path_between(upstream_node_name: str, downstream_node_name: str)`

These functions are wrapped into their visualization counterparts:

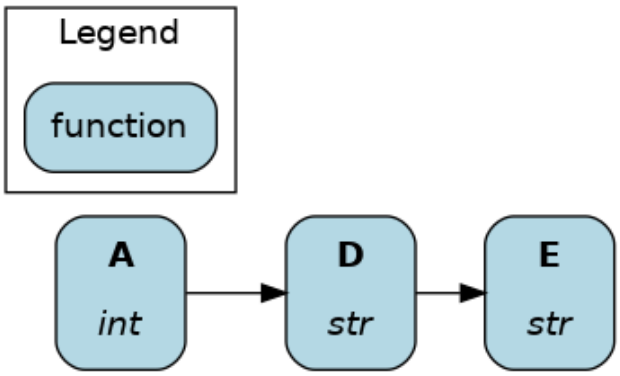
Display ancestors of `B`:

```
dr.display_upstream(["B"])
```



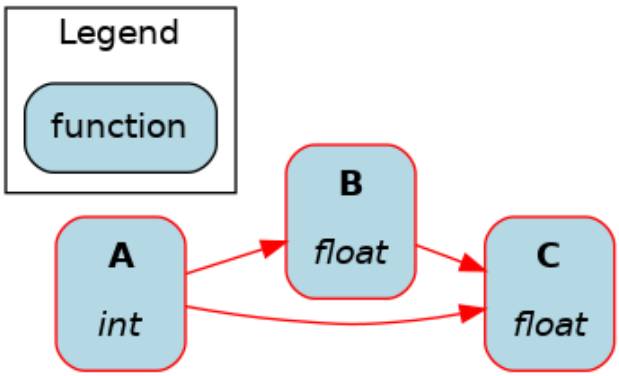
Display descendants of `D` and its immediate parents (`A` only).

```
dr.display_downstream(["D"])
```



Filter nodes to the necessary path:

```
dr.visualize_path-between("A", "C")
# dr.visualize_path-between("C", "D") would return
# ValueError: No path found between C and D.
```



Configure your visualization

All of the above visualization functions share parameters to customize the visualization (e.g., hide legend, hide inputs). Learn more by reviewing the API reference for Driver.display_all_functions(); parameters should apply to all other visualizations.

Custom node labels with display_name

Use the `@tag` decorator with `display_name` to show human-readable labels in visualizations while keeping valid Python identifiers as function names. This is useful for creating presentation-ready diagrams or adding business-friendly names:

```python
from hamilton.function_modifiers import tag

@tag(display_name="Parse Raw JSON")
def parse_raw_json(raw_data: str) -> dict:
    return json.loads(raw_data)

@tag(display_name="Transform to DataFrame")
def transform_to_df(parse_raw_json: dict) -> pd.DataFrame:
    return pd.DataFrame(parse_raw_json)
```

When visualized, nodes will display "Parse Raw JSON" and "Transform to DataFrame" instead of their function names. This keeps your code Pythonic while making visualizations more readable for stakeholders.

Note that `display_name` only affects visualization labels - the actual node names used in code and execution remain the function names.

Apply custom style

By default, each node is labeled with name and type, and stylized (shape, color, outline, etc.). By passing a function to the parameter `custom_style_function`, you can customize the node style based on its attributes. This pairs nicely with the `@tag` function modifier (learn more Add metadata to a node)

Your own custom style function must:

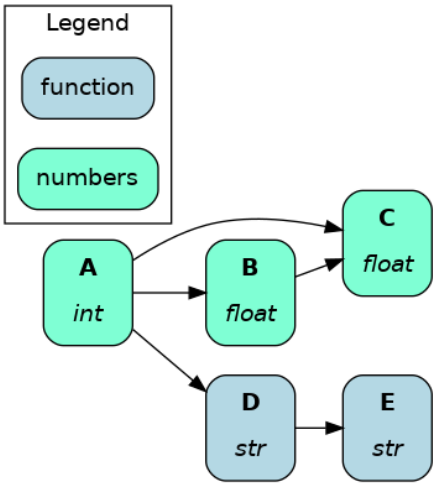
1. Use only keyword arguments, taking in `node` and `node_class`.
2. **Return a tuple `(style, node_class, legend_name)` where:**
    - `style`: dictionary of valid graphviz node style attributes.
    - `node_class`: class used to style the default visualization - we recommend returning the input `node_class`
    - `legend_name`: text to display in the legend. Return `None` for no legend entry.
3. For the execution-focused visualizations, your custom styles are applied before the modifiers for outputs and overrides are applied.

If you need more customization, we suggest getting the graphviz object produced, and modifying it directly.

This online graphviz editor can help you get started!

```python
def custom_style(
    *, node: graph_types.HamiltonNode, node_class: str
```

```
) -> Tuple[dict, Optional[str], Optional[str]]:
    """Custom style function for the visualization.

    :param node: node that Apache Hamilton is styling.
    :param node_class: class used to style the default visualization
    :return: a triple of (style, node_class, legend_name)
    """
    if node.type in [float, int]:
        style = ({"fillcolor": "aquamarine"}, node_class, "numbers")

    else:
        style = ({}, node_class, None)

    return style

dr.display_all_functions(custom_style_function=custom_style)
```



See the full code example for more details.

Materialization

So far, we executed our dataflow using the `Driver.execute()` method, which can receive an `inputs` dictionary and return a `results` dictionary (by default). However, you can also execute code with `Driver.materialize()` to directly read from / write to external data sources (file, database, cloud data store).

On this page, you'll learn:

- How to load and save data in Apache Hamilton
- Why use materialization
- What are `DataSaver` and `DataLoader` objects

- The difference between `.execute()` and `.materialize()`
- The basics to write your own materializer

Different ways to write the same dataflow

Below are 6 ways to write a dataflow that:

1. loads a dataframe from a parquet file
2. preprocesses the dataframe
3. trains a machine learning model
4. saves the trained model

The first two options don't use the concept of materialization and the next four do.

Without materialization

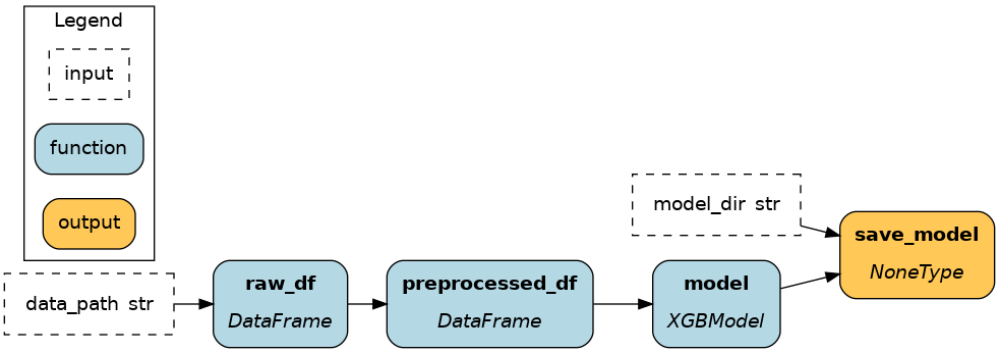
1. From nodes | **1. From `Driver`**

```python
import pandas as pd
import xgboost


def raw_df(data_path: str) -> pd.DataFrame:
    """Load raw data from parquet file"""
    return pd.read_parquet(data_path)


def preprocessed_df(raw_df: pd.DataFrame) ->
pd.DataFrame:
    """preprocess raw data"""
    return ...


def model(preprocessed_df: pd.DataFrame) ->
xgboost.XGBModel:
    """Train model on preprocessed data"""
    return ...
```

```python
import pandas as pd
import xgboost


def preprocessed_df(
pd.DataFrame:
    """preprocess ra
    return ...


def model(preprocess
xgboost.XGBModel:
    """Train model o
    return ...


if __name__ == "__ma
    import __main__

    from hamilton im
```

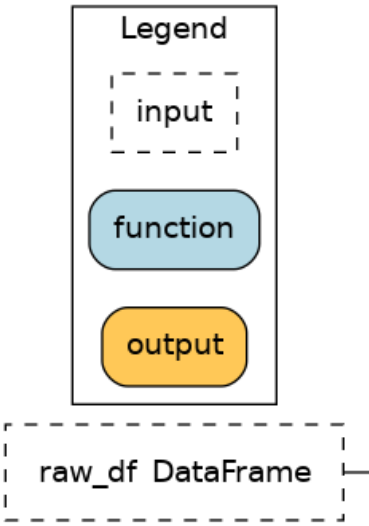
1. From nodes

```
def save_model(model: xgboost.XGBModel, model_dir:
str) -> None:
    """Save trained model to JSON format"""
    model.save_model(f"{model_dir}/model.json")


if __name__ == "__main__":
    import __main__

    from hamilton import driver

    dr =
driver.Builder().with_modules(__main__).build()

    data_path = "..."
    model_dir = "..."
    inputs = dict(data_path=data_path,
model_dir=model_dir)
    final_vars = ["save_model"]
    results = dr.execute(final_vars, inputs=inputs)
    # results["save_model"] == None
```



1. From `Driver`

```
    dr =
driver.Builder().wit

    data_path = "...
    model_dir = "...
    inputs = dict(ra
    final_vars = ["m

    results = dr.exe
    results["model"]
model.json")
```



Observations:

1. These two approaches load and save data using `pandas` and `xgboost` without any Apache Hamilton constructs. These methods are transparent and simple to get started, but as the number of node grows (or across projects) defining one node per parquet file to load introduces a lot of boilerplate.

2. Using **1) from nodes** improves visibility by including loading & saving in the dataflow (as illustrated).

3. Using **2) from ``Driver``** facilitates modifying loading & saving before code execution when executing the code, without modifying the dataflow itself. It is particularly useful when moving from development to production.

Limitations

Apache Hamilton's approach to "materializations" aims to solve 3 limitations:

1. **Redundancy**: deduplicate loading & saving code to improve maintainability and debugging

2. **Observability**: include loading & saving in the dataflow for full observability and allow hooks

3. **Flexibility**: change the loading & saving behavior without editing the dataflow

With materialization

1. Simple Materialization

```
import pandas as pd
import xgboost

from hamilton.function_modifiers import dataloader,
datasaver
from hamilton.io import utils


@dataloader()
def raw_df(data_path: str) -> tuple[pd.DataFrame,
dict]:
    """Load raw data from parquet file"""
    df = pd.read_parquet(data_path)
    return df,
utils.get_file_and_dataframe_metadata(data_path, df)


def preprocessed_df(raw_df: pd.DataFrame) ->
pd.DataFrame:
    """preprocess raw data"""
    return ...
```

1. Static materializers

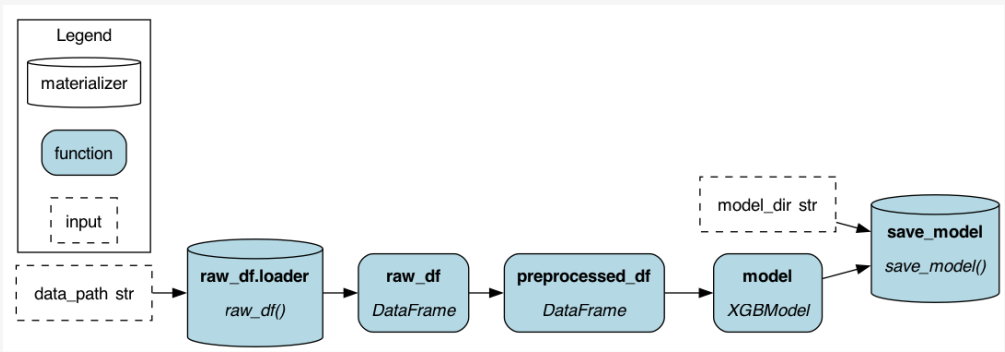
```
import pandas as pd
import xgboost


def preprocessed_df(
pd.DataFrame:
    """preprocess ra
    return ...


def model(preprocess
xgboost.XGBModel:
    """Train model
    return ...


if __name__ == "__ma
    import __main__

    from hamilton im
    from hamilton.io

    data_path = "...
```

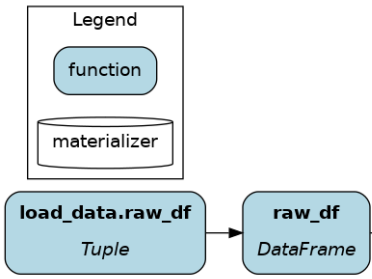
1. Simple Materialization

```python
def model(preprocessed_df: pd.DataFrame) ->
xgboost.XGBModel:
    """Train model on preprocessed data"""
    return ...


@datasaver()
def save_model(model: xgboost.XGBModel, model_dir:
str) -> dict:
    """Save trained model to JSON format"""
    model.save_model(f"{model_dir}/model.json")
    return utils.get_file_metadata(f"{model_dir}/
model.json")


if __name__ == "__main__":
    import __main__

    from hamilton import driver

    dr =
driver.Builder().with_modules(__main__).build()
    data_path = "..."
    model_dir = "..."
    inputs = dict(data_path=data_path,
model_dir=model_dir)
    final_vars = ["save_model"]
    results = dr.execute(final_vars, inputs=inputs)
    # results["save_model"] == None
```

1. Static materializers

```python
    model_dir = "...
    materializers =
        from_.parque
path=data_path),
        to.json(
            id="mode
DataSaver node
            depender
            path=f"
        ),
    ]
    dr = (
        driver.Build
        .with_module
        .with_materi
        .build()
    )

    results = dr.exe
    # results["model
    # results["model
the model
```





Simple Materialization

When you don't need to hide the implementation details of how you read and write, but you want to track what was read and written, you need to expose extra metadata. This is where the @datasaver() and @dataloader() decorators come in. They allow you to return metadata about what was read and written, and this metadata is then used to track what was read and written.

This is our recommended first step when you're starting to use materialization in Apache Hamilton.

Static materializers

Passing `from_` and `to` Apache Hamilton objects to `Builder().with_materializers()` injects into the dataflow standardized nodes to load and save data. It solves the 3 limitations highlighted in the previous section:

1. Redundancy ✅: Using the `from_` and `to` Apache Hamilton constructs reduces the boilerplate to load and save data from common formats (JSON, parquet, CSV, etc.) and to interact with 3rd party libraries (pandas, matplotlib, xgboost, dlt, etc.)
2. Observability ✅ Loaders and savers are part of the dataflow. You can view them with `Driver.display_all_functions()` and execute nodes by requesting them with `Driver.execute()`.
3. Flexibility ✅ The loading and saving behavior is decoupled from the dataflow and can modified easily when creating the `Driver` and executing code.

Dynamic materializers

The dataflow is executed by passing `from_` and `to` objects to `Driver.materialize()` instead of the regular `Driver.execute()`. This approach ressembles **2) from Driver**:

> **Note**
>
> `Driver.materialize()` can receive data savers (`from_`) and loaders (`to`) and will execute all `to` passed. Like `Driver.execute()`, it can receive `inputs`, and `overrides`, but instead of `final_vars` it receives `additional_vars`.

1. Redundancy ✅: Uses `from_` and `to` Apache Hamilton constructs.
2. Observability 🚸: Materializers are visible with `Driver.visualize_materialization()`, but can't be introspected otherwise. Also, you need to rely on `Driver.materialize()` which has a different call signature.

3. Flexibility ✅: Loading and saving is decoupled from the dataflow.

> **Note**
>
> Using static materializers is typically preferrable. Static and dynamic materializers can be used together with `dr = Builder.with_materializers().build()` and later `dr.materialize()`.

Function modifiers

By adding `@load_from` and `@save_to` function modifiers (Load and save external data) to Hamilton functions, materializers are generated when using `Builder.with_modules()`. This approach ressembles **1) from Driver**:

> **Note**
>
> Under the hood, the `@load_from` modifier uses the same code as `from_` to load data, same for `@save_to` and `to`.

1. Redundancy 🚸: Using `@load_from` and `@save_to` reduces redundancy. However, to make available to multiple nodes a loaded table, you would need to decorate each node with the same `@save_to`. Also, it might be impractical to decorate dynamically generated nodes (e.g., when using the `@parameterize` function modifier).

2. Observability ✅: Loaders and savers are part of the dataflow.

3. Flexibility 🚸: You can modify the path and materializer kwargs at runtime using `source()` in the decorator definition, but you can't change the format itself (e.g., from parquet to CSV).

> **Note**
>
> It can be desirable to couple loading and saving to the dataflow using function modifiers. It makes it clear when reading the dataflow definition which nodes should load or save data using external sources.

DataLoader and DataSaver

In Apache Hamilton, `DataLoader` and `DataSaver` are classes that define how to load or save a particular data format. Calling `Driver.materialize(DataLoader(), DataSaver())` adds nodes to the dataflow (see visualizations above).

Here are simplified snippets for saving and loading an XGBoost model to/from JSON.

DataLoader

```
import dataclasses
from os import PathLike
from typing import Any,
Collection, Dict, Tuple, Type,
Union

import xgboost

from hamilton.io import utils
from hamilton.io.data_adapters
import DataLoader


@dataclasses.dataclass
class
XGBoostJsonReader(DataLoader):
    path: Union[str, bytearray,
PathLike]

    @classmethod
    def applicable_types(cls) ->
Collection[Type]:
        return [xgboost.XGBModel]

    def load_data(self, type_:
Type) -> Tuple[xgboost.XGBModel,
Dict[str, Any]]:
        # uses the XGBoost library
        model = type_()

model.load_model(self.path)
        metadata =
utils.get_file_metadata(self.path)
        return model, metadata

    @classmethod
```

DataSaver

```
import dataclasses
from os import PathLike
from typing import Any,
Collection, Dict, Type, Union

import xgboost

from hamilton.io import utils
from hamilton.io.data_adapters
import DataSaver


@dataclasses.dataclass
class
XGBoostJsonWriter(DataSaver):
    path: Union[str, PathLike]

    @classmethod
    def applicable_types(cls) ->
Collection[Type]:
        return [xgboost.XGBModel]

    def save_data(self, data:
xgboost.XGBModel) -> Dict[str,
Any]:
        # uses the XGBoost library
        data.save_model(self.path)
        return
utils.get_file_metadata(self.path)

    @classmethod
    def name(cls) -> str:
        return "json"  # the name
for `to.{name}`
```

DataLoader	DataSaver

```
    def name(cls) -> str:
        return "json"  # the name
for `from_.{name}`
```

To define your own DataSaver and DataLoader, the Apache Hamilton XGBoost extension provides a good example

Function modifiers

In Functions, nodes & dataflow, we discussed how to write Python functions to define Hamilton nodes and dataflow. In the basic case, each function defines one node.

Yet, it's common to need nodes with similar purposes but different dependencies, such as preprocessing training and evaluation datasets. In that case, using a **function modifier** can help create both nodes from a single Hamilton function!

On this page, you'll learn:

- Python decorators basics
- Add metadata to node
- Validate node output
- Split node output into *n* nodes
- Define one function, create *n* nodes
- Select nodes to load from module

This page covers important conceptual notions but is not exhaustive. To find details about all function modifiers see API references Decorators.

Decorators

Python decorators are statements that begin with `@` located above function definitions. Apache Hamilton uses decorators to implement function modifiers and reduce the amount of code you have to write to make expressive dataflows.

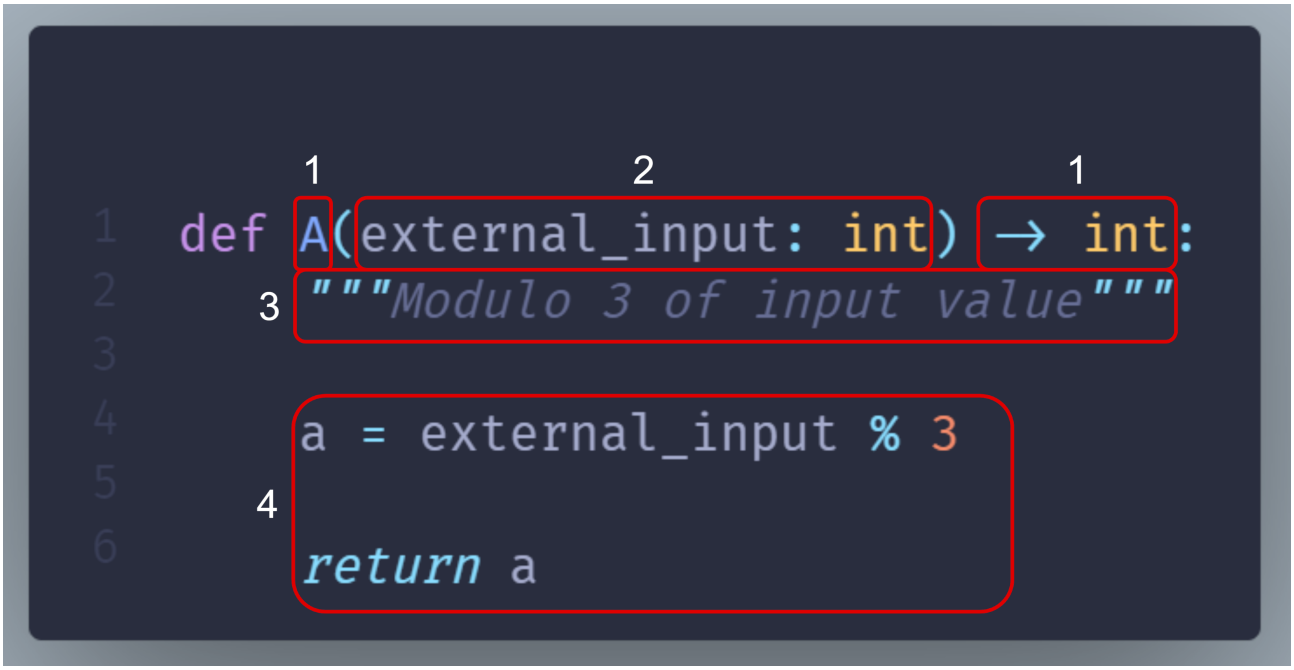
Multiple decorators can be stacked on a single function and are applied from bottom to top. Apache Hamilton decorators should be insensitive to ordering, but be careful with non-Apache Hamilton decorators (e.g., @retries, @time). See this decorator primer to learn more.

Function modifiers were designed to have clear semantics, so you should be able to figure out what they do from their name. For instance, the following code adds metadata using `@tag` and conducts some checks over the return value with `check_output`.

```
@tag(owner='Data-Science', pii='False')
@check_output(data_type=np.float64, range=(-5.0, 5.0),
allow_nans=False)
def height_zero_mean_unit_variance(
    height_zero_mean: pd.Series, height_std_dev: pd.Series
) -> pd.Series:
"""Zero mean unit variance value of height"""
return height_zero_mean / height_std_dev
```

Reminder: Anatomy of a node

This section from the page Functions, nodes & dataflow details how a Python function maps to a Hamilton node. We'll reuse these terms to explain the function modifiers.



id	Function components	Node components
1	Function name and return type annotation	Node name and type

id	Function components	Node components
2	Parameter names and type annotations	Node dependencies
3	Docstring	Description of the node return value
4	Function body	Implementation of the node

Add metadata to a node

@tag

The `@tag` decorator **doesn't modify the function/node**. It attaches metadata to the node that can be used by Apache Hamilton and you. It can help tag nodes by ownership, data source, version, infrastructure, and anything else.

For example, this tags the associated data product and the sensitivity of the data.

```
from hamilton.function_modifiers import tag

@tag(data_product='final', pii='true')
def final_column(
    intermediate_column: pd.Series
) -> pd.Series: ...
```

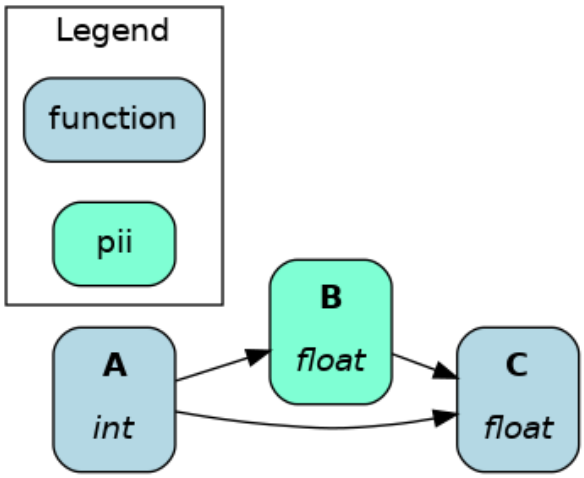
Query node by tag

Once you built your Driver, you can get all nodes with `Driver.list_available_variables()` and then filter them by tag. The following gets all the nodes for which `data_product="final"` and passes them to `driver.execute()`

```
dr = driver.Builder().with_modules(my_module).build()
tagged_nodes = [node.name for node in dr.list_available_variables()
                if 'final' == node.tags.get('data_product')]

results = dr.execute(tagged_nodes)
```

Customize visualization by tag

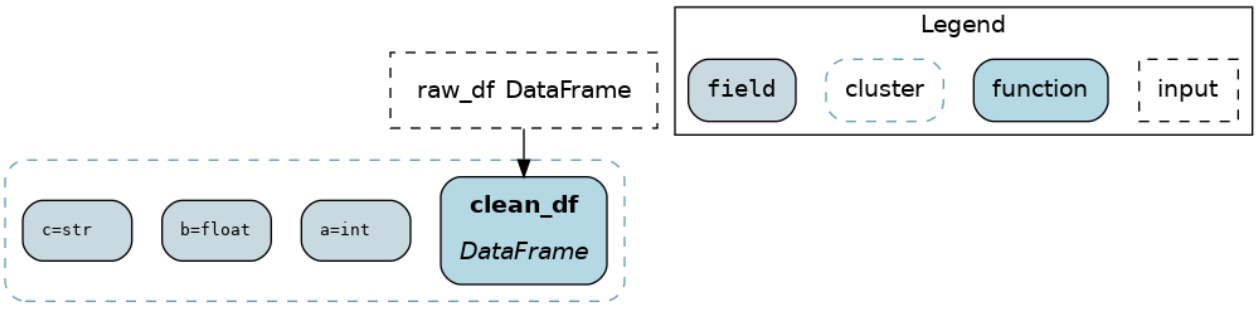
Tags are also accessible to the visualization styling feature, allowing you to highlight important nodes for your documentation. See Apply custom style for details.



@schema

The `@schema` function modifiers provides a lightweight way to add type metadata to dataframes. It works by specifying tuples of `(field_name, field_type)` with types as strings.

```
from hamilton.function_modifiers import schema

@schema.output(
    ("a", "int"),
    ("b", "float"),
    ("c", "str")
)
def clean_df(raw_df: pd.DataFrame) -> pd.DataFrame:
    return pd.DataFrame.from_records(
        {"a": [1], "b": [2.0], "c": ["3"]}
    )
```



Validate node output

The `@check_output` function modifiers are applied on the **node output / function return** and therefore don't directly affect node behavior. Decorators separate data validation from the function body where the core logic is. It improves function readability, and it helps reusing and maintaining standardized checks across multiple functions.

Note

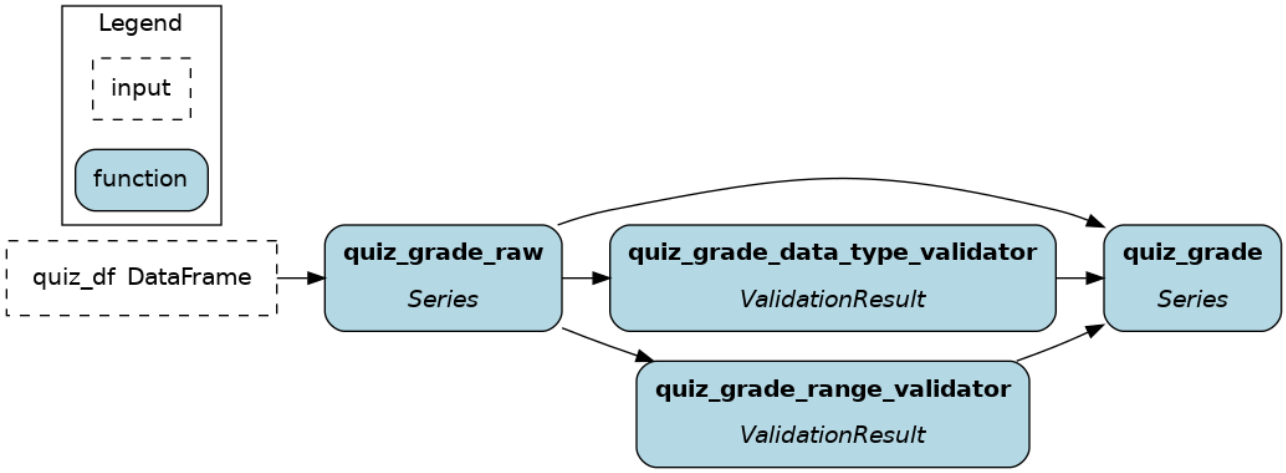
In the future, validation capabilities may be added to `@schema`. For now, it's only added metadata.

@check_output*

The `@check_output` implements many data checks for Python objects and DataFrame/Series including data type, min/max/between, count, fraction of null/nan values, and allow null/nan. Failed checks are either logged (`importance="warn"`) or make the dataflow fail (`importance="fail"`).

The next snippet checks if the returned Series is of type `np.int32`, which is non-nullable, and if its within the range 0-100, and logs failed checks. This allows us to manually review instances where data validation failed.

```
from hamilton.function_modifiers import check_output

@check_output(data_type=np.int32, range=(0,100), importance="warn")
def quiz_grade(quiz_df: pd.DataFrame) -> pd.Series:
    return ...
```



- To see all available validators, go to the file `hamilton/data_quality/default_validators.py` and view the variable `AVAILABLE_DEFAULT_VALIDATORS`.
- The function modifier `@check_output_custom` allows you to define your own validator. Validators inherit the `base.BaseDefaultValidator` class and are essentially standardized Hamilton node definitions (instead of functions). See `hamilton/data_quality/default_validators.py` or reach out on Slack for help!
- Note: `@check_output_custom` decorators cannot be stacked, but they instead can take multiple validators.

Note

As you see, validation steps effectively add nodes to the dataflow and the visualization. This helps trace which specific check failed for instance, but it can make visualizations harder to read.

You can hide these nodes using the custom visualization style feature (see Apply custom style) by applying the style `{"style": "invis"}` to nodes with the tag `hamilton.data_quality.source_node`. This will only keep the original nodes and their `_raw` variant.

pandera support

Apache Hamilton has a pandera plugin for data validation that you can install with `pip install sf-hamilton[pandera]`. Then, you can pass a pandera schema (for DataFrame or Series) to `@check_output(schema=...)`.

pydantic support

Apache Hamilton also supports data validation of pydantic models, which can be enabled with `pip install sf-hamilton[pydantic]`. With pydantic installed, you can pass any subclass of the pydantic base model to `@check_output(model=...)`. Pydantic validation is performed in strict mode, meaning that raw values will not be coerced to the model's types. For more information on strict mode see the pydantic docs.

Split node output into *n* nodes

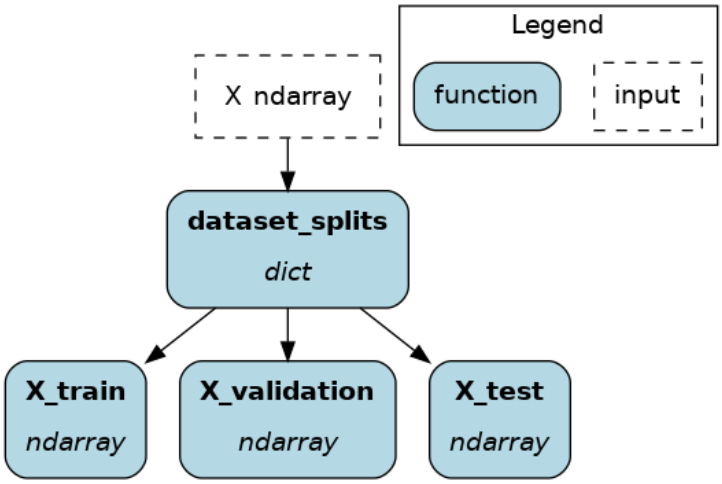
Sometimes, your node outputs multiple values that you would like to name and make available to other nodes. These function modifiers act on the **node output / function return**.

> **Note**
>
> To add metadata to extracted nodes, use `@tag_output`, which works just like `@tag`.

@unpack_fields

A good example is splitting a dataset into training, validation, and test splits. We use `@unpack_fields`, which requires specifying the names of the fields to extract. The function must return a tuple with at least as many elements as there are specified fields. Note that selecting a subset of the tuple or using an indeterminate tuple size is also possible.

```
from typing import Tuple
from hamilton.function_modifiers import unpack_fields

@unpack_fields("X_train", "X_validation", "X_test")
def dataset_splits(X: np.ndarray) -> Tuple[np.ndarray, np.ndarray,
np.ndarray]:
    """Randomly split data into train, validation, test"""
    X_train, X_validation, X_test = random_split(X)
    return X_train, X_validation, X_test
```

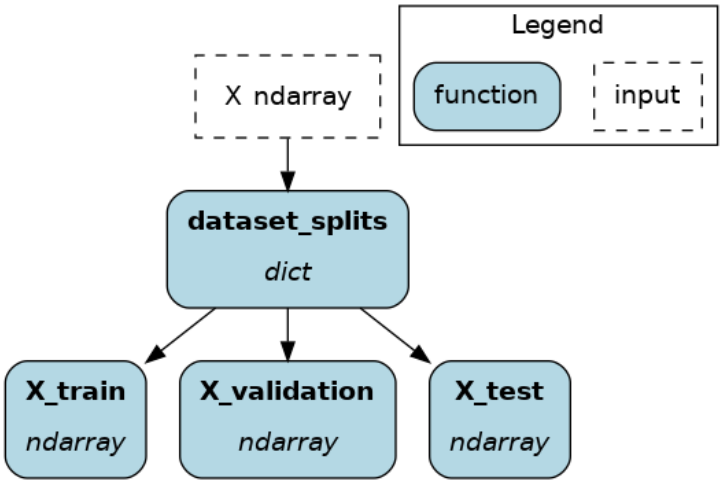


Now, `X_train`, `X_validation`, and `X_test` are available to other nodes and can be queried with `.execute()`. However, since `dataset_splits` is itself a node, you can query it to obtain all splits in a single tuple!

@extract_fields

Additionally, we can extract fields from an output dictionary using `@extract_fields`. The function must return a dictionary that contains, at a minimum, those keys specified in the decorator. In this case, you can specify a dictionary of fields and their types:

```
from typing import Dict
from hamilton.function_modifiers import extract_fields

@extract_fields(dict(  # fields specified as a dictionary
    X_train=np.ndarray,
    X_validation=np.ndarray,
    X_test=np.ndarray,
))
def dataset_splits(X: np.ndarray) -> Dict:
    """Randomly split data into train, validation, test"""
    X_train, X_validation, X_test = random_split(X)
    return dict(
        X_train=X_train,  # keys match those from @extract_fields
        X_validation=X_validation,
        X_test=X_test,
    )
```



Or if you are using a generic dictionary, you can specify solely the field names.

```
from typing import Dict
from hamilton.function_modifiers import extract_fields

@extract_fields("X_train", "X_validation", "X_test")  # field names
only
def dataset_splits(X: np.ndarray) -> Dict[str, np.ndarray]:  #
generic dict
    """Randomly split data into train, validation, test"""
    X_train, X_validation, X_test = random_split(X)
    return dict(
        X_train=X_train,
        X_validation=X_validation,
        X_test=X_test,
    )
```

If you are using a *TypedDict*, you can specify the just field names.

```
from typing import TypedDict
from hamilton.function_modifiers import extract_fields

class DatasetSplits(TypedDict):
    X_train: np.ndarray
    X_validation: np.ndarray
    X_test: np.ndarray

@extract_fields("X_train", "X_validation", "X_test")
def dataset_splits(X: np.ndarray) -> DatasetSplits:
    """Randomly split data into train, validation, test"""
    X_train, X_validation, X_test = random_split(X)
    return dict(
        X_train=X_train,
        X_validation=X_validation,
```

```
        X_test=X_test,
    )
```

Or you can leave the field names empty and extract all fields from the *TypedDict*.

```
from typing import TypedDict
from hamilton.function_modifiers import extract_fields

class DatasetSplits(TypedDict):
    X_train: np.ndarray
    X_validation: np.ndarray
    X_test: np.ndarray

@extract_fields(DatasetSplits)  # field names only
def dataset_splits(X: np.ndarray) -> DatasetSplits:
    """Randomly split data into train, validation, test"""
    X_train, X_validation, X_test = random_split(X)
    return dict(
        X_train=X_train,
        X_validation=X_validation,
        X_test=X_test,
    )
```

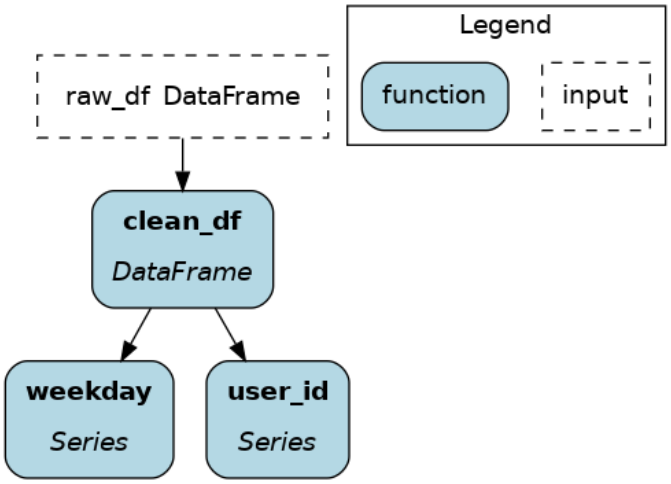
Again, `X_train`, `X_validation`, and `X_test` are now available to other nodes, or you can query the `dataset_splits` node to retrieve all splits in a dictionary.

@extract_columns

`@extract_columns` is a specialized version of `@extract_fields` to get individual columns of a dataframe (pandas, polars, Spark, etc.). It enables column-level lineage which improves visibility over data transformations and facilitates reusing feature transformations. Also, it can reduce memory usage by avoiding moving large dataframe through nodes.

Since it knows how to extract series from a dataframe, you just have to specify the column names.

```
from hamilton.function_modifiers import extract_columns

# assuming `user_id` and `weekday` are existing columns
# note that strings are passed directly, without a list
@extract_columns("user_id", "weekday")
def clean_df(raw_df: pd.DataFrame) -> pd.DataFrame:
    """Clean my data"""
    clean_df = clean_my_data(raw_df)
    return clean_df
```



Define one function, create *n* nodes

The family of `@parameterize` function modifiers allows the creation of multiple nodes with the same **node implementation / function body** (and therefore output type), but different **node inputs**.

This has many applications, such as producing the same performance plot for multiple models or computing groupby aggregates along different dimensions.

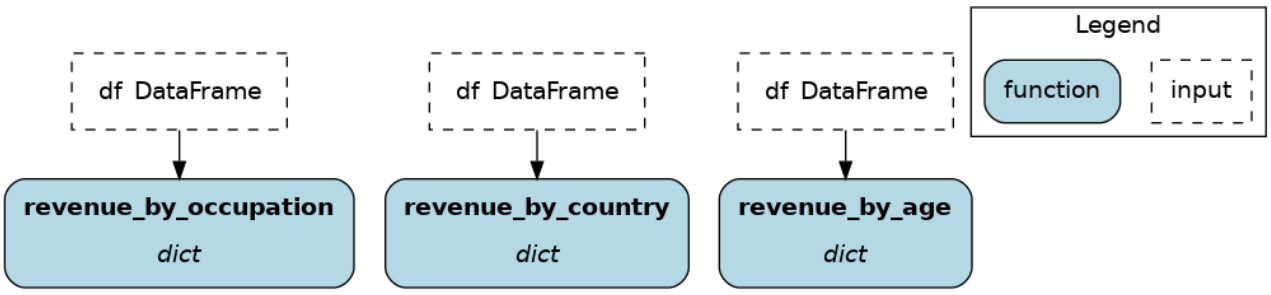
@parameterize

You need to specify the generated **node name**, a dictionary of dependencies, and optionally a docstring. For the dependencies, you can pass constants with `value()` or get them from the dataflow by passing a node name to `source()`. These notions are tricky at first, but let's look at an example:

We create 3 nodes: `revenue_by_age`, `revenue_by_country`, `revenue_by_occupation`. For each, we get the dataframe `df` from the dataflow using `source()` and specify a different `groupby_col` with `value()`. Also, the docstring uses `{groupby_col}` to have the value inserted.

```
from hamilton.function_modifiers import parameterize
from hamilton.function_modifiers import source, value

@parameterize(
    revenue_by_age=dict(df=source("df"), groupby_col=value("age")),
    revenue_by_country=dict(df=source("df"),
groupby_col=value("country")),
    revenue_by_occupation=dict(df=source("df"),
groupby_col=value("occupation")),
)
def population_metrics(df: pd.DataFrame, groupby_col: str) -> dict:
    """Compute df metrics aggregates over dimension {groupby_col}"""
```

```
    return df.groupby(groupby_col)["revenue"] \
             .agg(["mean", "min", "max"]) \
             .to_dict()
```



- The above example mixes constant `value()` and dataflow `source()` dependencies. The syntax is indeed verbose. Simplified syntaxes are available through `@parameterize_values` and `@parameterize_sources` if you only need one type of dependency.
- If you need to extract columns from the output of a generated node, use `@parameterize_extract_columns`

Select functions to include

The family of `@config` decorators doesn't modify the function. Rather, it tells the Driver which functions from the module (and therefore nodes) to include in the dataflow. This helps projects that need to run in different contexts (e.g., locally vs orchestrator) or need to swap different implementations of a node (e.g., ML experiments, code migration, A/B testing).

Note

At first, there can be confusion between `@config` and the `inputs` and `overrides` of the Driver's `.execute()` and `.materialize()` methods. In common language, people might refer to the `.execute(inputs=..., overrides=...)` as a configuration. However, these two affect the values passing through the dataflow **once the Driver is built** while `@config` determines **how the Driver is built**.

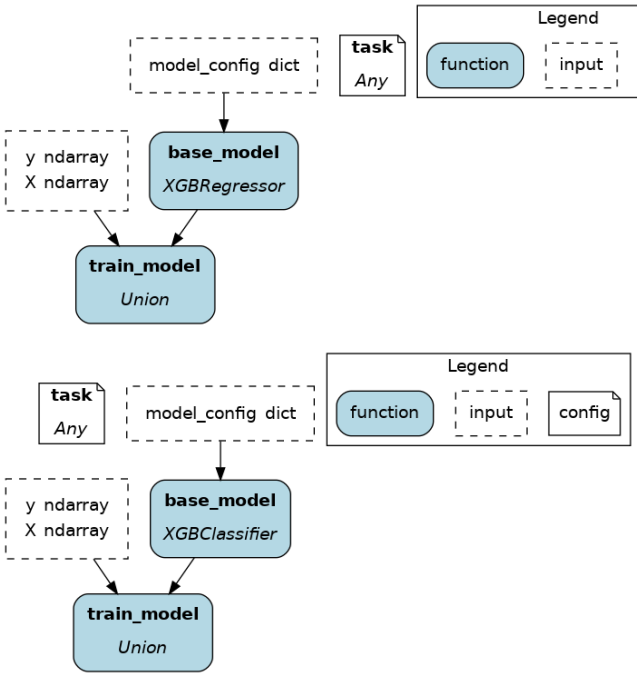
@config

For the decorator, you must specify one or more `key=value` pairs. Then, you need to add to the Builder `.with_config()` and give it a dictionary of `key=value` pairs. This will determine which functions to load.

This example uses `@config.when()` to select between a binary classifier and a regressor model. Notice a few elements:

- both functions have the same name `base_model` with a suffix `__binary` or `__regression`. This is required because Python enforces unique function names. After the config determines which function to load, Apache Hamilton will remove the suffix from the node name.
- the two functions have different return types, so `train_model` needs to annotate `base_model` as a `Union[]` type.

```
# model_training.py
from hamilton.function_modifiers import config

@config.when(task="binary_classification")
def base_model__binary() -> XGBClassifier:
    return XGBClassifier(...)

@config.when(task="continuous_regression")
def base_model__regression() -> XGBRegressor:
    return XGBRegressor(...)

def train_model(
    base_model: Union[XGBClassifier, XGBRegressor],
    X: np.ndarray,
    y: np.ndarray,
) -> Union[XGBClassifier, XGBRegressor]:
    return ...

# run.py
dr = (
    driver.Builder()
    .with_modules(model_training)
    .with_config(dict(task="continuous_regression"))
    .build()
)
```



In the above example, if the Driver receives no value for the key `task` or the value isn't `"binary_classification"` or `"continuous_regression"`, there would be no `base_model` node loaded and `train_model` would fail.

Using `@config.when_not()` can help set up a default case and ensure a `base_model` node is always loaded.

```
@config.when(library="xgboost")
def base_model__xgboost() -> XGBClassifier:
    return XGBClassifier(...)

@config.when_not(library="xgboost")
def base_model__default() -> sklearn.ensemble.RandomForestRegressor:
    return sklearn.ensemble.RandomForestRegressor(...)
```

There exists also `@config.when_in()` and `@config.when_not_in()` that accept a list of values to check. Expanding on the previous example:

```
@config.when(library="xgboost")
def base_model__xgboost() -> XGBClassifier:
    return XGBClassifier(...)

@config.when(library="lightgbm")
def base_model__lightgbm() -> LGBMClassifier:
    return LGBMClassifier(...)

@config.when_not_in(library=["xgboost", "lightgbm"])
def base_model__default() -> sklearn.ensemble.RandomForestRegressor:
    return sklearn.ensemble.RandomForestRegressor(...)
```

Load and save external data

Most dataflows require reading or writing data to external sources in some capacity. It's a good idea to conduct this step in a node separated from transformations to trace failures more easily.

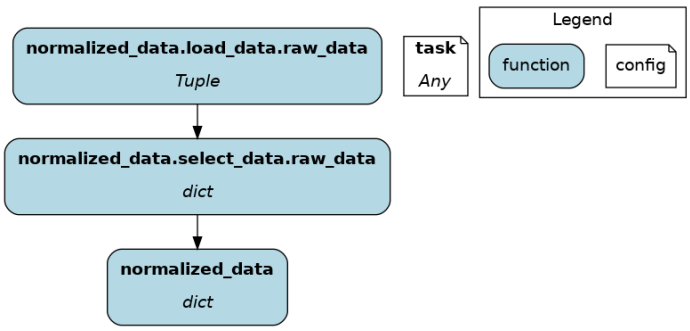
Nevertheless, adding one function per read/write becomes tedious and hard to maintain. Apache Hamilton provides well-tested implementations for common formats (JSON, CSV, Parquet, etc.) available through `@load_from` and `@save_to` decorators and materializers (see Materialization).

More formats are available through Apache Hamilton plugins, and you should be able to add your own custom loader/saver (reach out on Slack for help!)

@load_from

You can think of `@load_from` as adding an upstream node. The next example specifies the `path` of the file, which will be loaded in the variable `raw_data`. Note that the variable type should be compatible with the loaded file (`dict` for JSON here).

```
@load_from.json(path="/path/to/file.json")
def normalized_data(raw_data: dict) -> dict:
    return ...
```

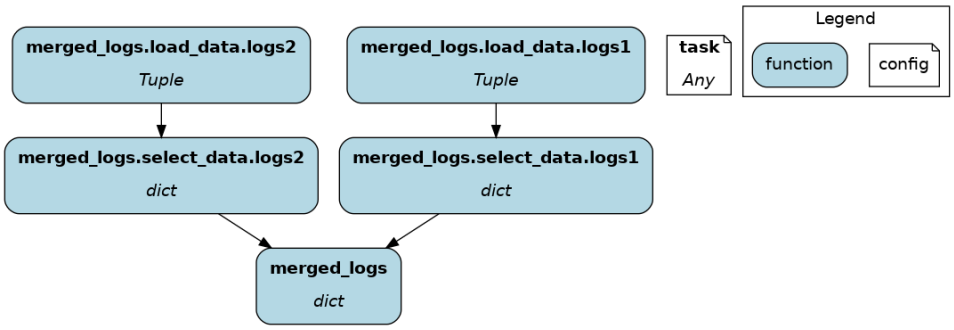


It is possible to use `source()` (like in `@parameterize`) to specify the file path from the driver code. See:

```
# functions.py
@load_from.json(path=source("raw_data_path"))
def normalized_data(raw_data: dict) -> dict:
    return ...

# run.py
dr = driver.Builder().with_modules(functions).build()
dr.execute(["normalized_data"], inputs=dict(raw_data_path="./this/
file.json"))
```

You will need to use the `inject_` keyword when you load multiple files into a node or your function has multiple parameters.

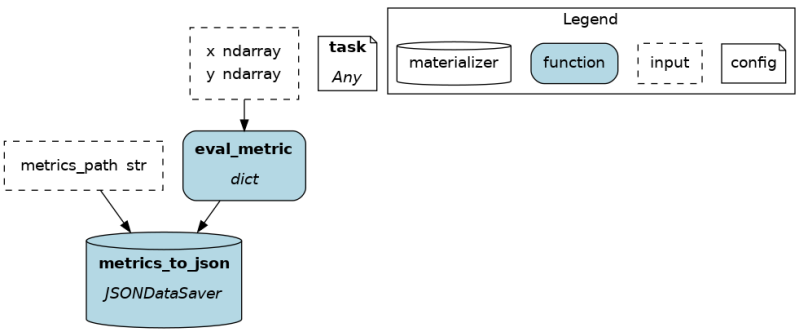
```
@load_from.json(path="/path/to/logs.json", inject_="logs1")
@load_from.json(path="/path/to/other/logs.json", inject_="logs2")
def merged_logs(logs1: dict, logs2: dict) -> dict:
    return ...
```



@save_to

The `@save_to` decorator works very similarly to `@load_from`. In this case, `path=...` specifies where the data will be saved, and an `output_name_` is required to be able to request the node from `Driver.execute()`. Here again, `source()` can be used.

```
# functions.py
@save_to.json(path=source("metrics_path"),
output_name_="metrics_to_json")
def eval_metric(x: np.ndarray, y: np.ndarray) -> dict:
    return dict(...)

# run.py
dr = driver.Builder().with_modules(functions).build()
dr.execute(["metrics_to_json"], inputs=dict(metrics_path="./out/
metrics.json"))
```



Builder

The Driver page covered the basics of building the Driver, visualizing the dataflow, and executing the dataflow. We learned how to create the dataflow by passing a Python module to `Builder().with_modules()`.

On this page, how to configure your Driver with the `driver.Builder()`. There will be mentions of advanced concepts, which are further explained on their respective page.

Note

As your Builder code grows complex, defining it over multiple lines can improve readability. This is possible by using parentheses after the assignment `=`

```
dr = (
    driver.Builder()
    .with_modules(my_dataflow)
    .build()
)
```

The order of Builder statements doesn't matter as long as `.build()` is last.

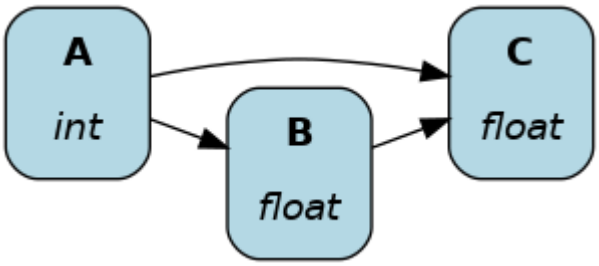
with_modules()

This passes dataflow modules to the Driver. When passing multiple modules, the Driver assembles them into a single dataflow.

```
# my_dataflow.py
def A() -> int:
    """Constant value 35"""
    return 35

def B(A: int) -> float:
    """Divide A by 3"""
    return A / 3
```

```
# my_other_dataflow.py
def C(A: int, B: float) -> float:
    """Square A and multiply by B"""
    return A**2 * B
```

```
# run.py
from hamilton import driver
import my_dataflow
import my_other_dataflow

dr = driver.Builder().with_modules(my_dataflow,
my_other_dataflow).build()
```



It encourages organizing code into logical modules (e.g., feature processing, model training, model evaluation). `features.py` might depend on PySpark and `model_training.py` on XGBoost. By organizing modules by dependencies, it's easier to reuse the XGBoost model training module in a project that doesn't use PySpark and avoid version conflicts.

```
# run.py
from hamilton import driver
import features
import model_training
import model_evaluation

dr = (
    driver.Builder()
    .with_modules(features, model_training, model_evaluation)
    .build()
)
```

Note

Your modules may have same named functions which will raise an error when using `.build()` since we cannot have two nodes with the same name. You can use the method `.allow_module_overrides()` and Apache Hamilton will choose the function from the later imported module.

```
dr = (
    driver.Builder()
    .with_modules(module_A, module_B)
    .allow_module_overrides()
```

```
        .build()
    )
```

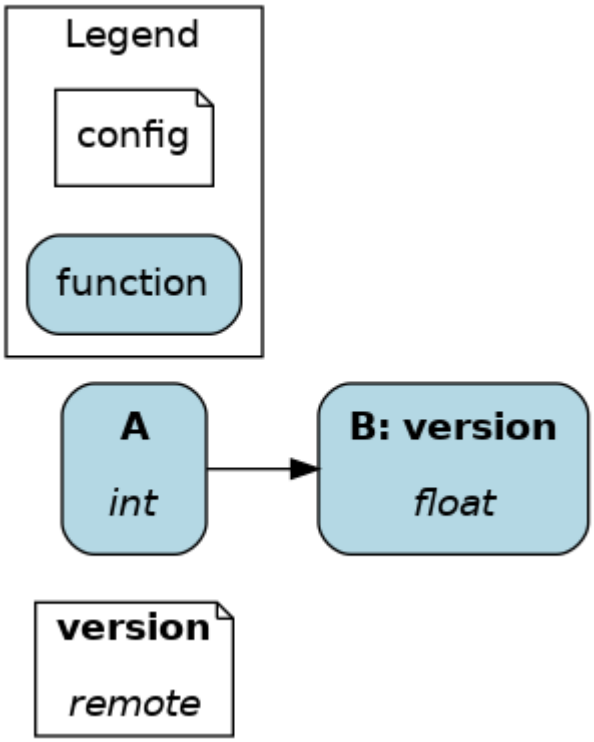
If `module_A` and `module_B` both have the function `foo()`, Apache Hamilton will use `module_B.foo()` when constructing the DAG. See https://github.com/apache/hamilton/tree/main/examples/module_overrides for more info.

with_config()

This is directly related to the `@config` function decorator (see Select functions to include) and doesn't have any effect in its absence. By passing a dictionary to `with_config()`, you configure which functions will be used to create the dataflow. You can't change the config after the Driver is created. Instead, you need to rebuild the Driver with the new config values.

```
# my_dataflow.py
from hamilton.function_modifiers import config

def A() -> int:
    """Constant value 35"""
    return 35

@config.when_not(version="remote")
def B__default(A: int) -> float:
    """Divide A by 3"""
    return A / 3

@config.when(version="remote")
def B__remote(A: int) -> float:
    """Divide A by 2"""
    return A / 2
```

```
# run.py
from hamilton import driver
import my_dataflow

dr = (
    driver.Builder()
    .with_modules(my_dataflow)
    .with_config(dict(version="remote"))
    .build()
)

dr.display_all_functions("dag.png")
```



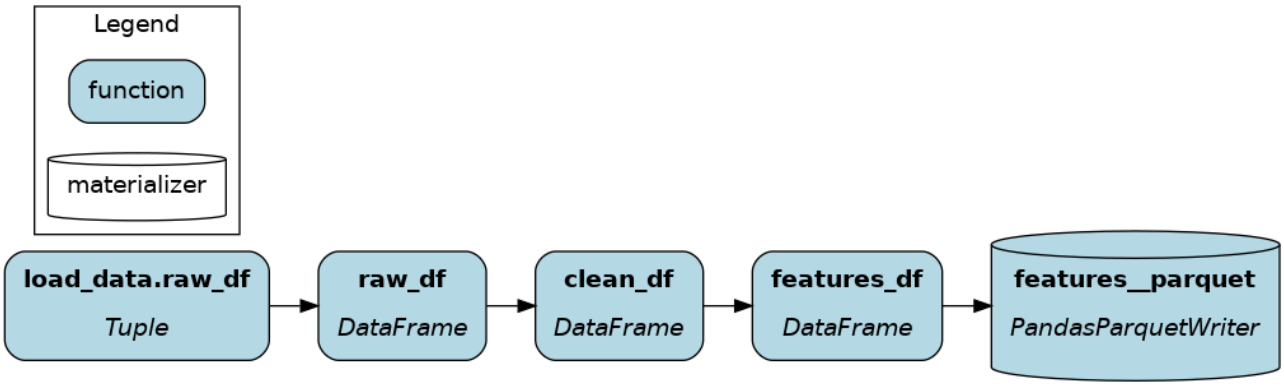
with_materializers()

Adds *DataSaver* and *DataLoader* nodes to your dataflow. This allows to visualize these nodes using `Driver.display_all_functions()` and be executed by name with `Driver.execute()`. More details on the Materialization documentation page.

```
# my_dataflow.py
import pandas as pd
from hamilton.function_modifiers import config

def clean_df(raw_df: pd.DataFrame) -> pd.DataFrame:
    return ...

def features_df(clean_df: pd.DataFrame) -> pd.DataFrame:
    return ...
```

```
# run.py
from hamilton import driver
from hamilton.io.materialization import from_, to
import my_dataflow

loader = from_.parquet(target="raw_df", path="/my/raw_file.parquet")
saver = to.parquet(
    id="features__parquet",
    dependencies=["features_df"],
    path="/my/feature_file.parquet"
```

```
)

dr = (
    driver.Builder()
    .with_modules(my_dataflow)
    .with_materializers(loader, saver)
    .build()
)
dr.display_all_functions("dag.png")

dr.execute(["features__parquet"])
```



with_cache()

This enables Apache Hamilton's caching feature, which allows to automatically store intermediary results and reuse them in subsequent executions to skip computations. Learn more in the Caching section.

```
from hamilton import driver
import my_dataflow

dr = (
    driver.Builder()
    .with_modules(my_dataflow)
    .with_cache()
    .build()
)
```

with_adapters()

This allows to add multiple Lifecycle hooks to the Driver. This is a very flexible abstraction to develop custom plugins to do logging, telemetry, alerts, and more. The following adds a hook to launch debugger when reaching the node `"B"`:

```
# run.py
from hamilton import driver, lifecycle
import my_dataflow

debug_hook = lifecycle.default.PDBDebugger(node_filter="B",
during=True)
dr = (
    driver.Builder()
    .with_modules(my_dataflow)
    .with_adapters(debug_hook)
    .build()
)
```

Other hooks are available to output a progress bar in the terminal, do experiment tracking for your Apache Hamilton runs, cache results to disk, send logs to DataDog, and more!

enable_dynamic_execution()

This directly relates to the Builder `with_local_executor()` and `with_remote_executor()` and the `Parallelizable/Collect` functions (see Dynamic DAGs/Parallel Execution). For the Driver to be able to parse them, you need to set `allow_experimental_mode=True` like the following:

```
# run.py
from hamilton import driver
import my_dataflow  # <- this contains Parallelizable/Collect nodes

dr = (
    driver.Builder()
    .enable_dynamic_execution(allow_experimental_mode=True)  # set
True
    .with_modules(my_dataflow)
    .build()
)
```

By enabling dynamic execution, reasonable defaults are used for local and remote executors. You also specify them explicitly as such:

```
# run.py
from hamilton import driver
from hamilton.execution import executors
import my_dataflow

dr = (
    driver.Builder()
    .with_modules(my_dataflow)
```

```
    .enable_dynamic_execution(allow_experimental_mode=True)
    .with_local_executor(executors.SynchronousLocalTaskExecutor())
    .with_remote_executor(executors.MultiProcessingExecutor(max_tasks=5))
    .build()
)
```

Caching

Caching enables storing execution results to be reused in later executions, effectively skipping redundant computations. This speeds up execution and saves resources (computation, API credits, GPU time, etc.), and has applications both for development and production.

To enable caching, add `.with_cache()` to your `Builder()`.

```
from hamilton import driver
import my_dataflow

dr = (
    driver.Builder()
    .with_module(my_dataflow)
    .with_cache()
    .build()
)

dr.execute([...])
dr.execute([...])
```

The first execution will store **metadata** and **results** next to the current directory under `./.hamilton_cache`. The next execution will retrieve results from cache when possible to skip execution.

Note

We highly suggest viewing the Caching tutorial for a practical introduction to caching.

How does it work?

Caching relies on multiple components:

- **Cache adapter**: decide to retrieve a result or execute the node

- **Metadata store**: store information about past node executions
- **Result store**: store results on disk, it is unaware of other cache components.

At a high-level, the cache adapter does the following for each node:

1. Before execution: determine the `cache_key`
2. **At execution:**
    1. if the `cache_key` finds a match in the metadata store (cache **hit**), retrieve the `data_version` of the `result`.
    2. If there's no match (cache **miss**), execute the node and store the `data_version` of the `result` in the metadata store.
3. After execution: if we had to execute the node, store the `result` in the result store.

The caching mechanism is highly performant because it can pass `data_version` (small strings) through the dataflow instead of the actual data until a node needs to be executed.

The result store is a mapping of `{data_version: result}`. While a `cache_key` is unique to determine retrieval or execution, multiple cache keys can point to the same `data_version`, which avoid storing duplicate results.

Cache key

Understanding the `cache_key` is important to understand why a node is recomputed or not. It is composed of:

- `node_name`: name of the node
- `code_version`: version of the node's code
- `dependencies_data_versions`: `data_version` of each dependency of the node

```
{
    "node_name": "processed_data",
    "code_version":
"c2ccafa54280fbc969870b6baa445211277d7e8cfa98a0821836c175603ffda2",
    "dependencies_data_versions": {
        "raw_data": "WgV5-4SfdKTfUY66x-msj_xXsKNPNTP2guRhfw==",
        "date": "ZWNhd-XNlIF0YV9-2ZXJzaW9u_YGAgKA==",
    }
}
```

By traversing the cache keys' `dependencies_data_versions`, we can actually reconstruct the dataflow structure!

⚠ Warning

Cache keys could be unstable across Python and Apache Hamilton versions (because of new features, bug fixes, etc.). Upgrading Python or Apache Hamilton could require starting with a new empty cache for reliable behavior.

Observing the cache

Caching is best understood throung interacting with it. Apache Hamilton offers many utilities to observe and introspect the cache manually.

Logging

To see how the cache works step-by-step, start your code (script, notebook, etc.) by getting the logger and setting the level to `DEBUG`. Using `INFO` will be less noisy and only log `GET_RESULT` and `EXECUTE_NODE` events.

```
import logging

logger = logging.getLogger("hamilton.caching")
logger.setLevel(logging.INFO)
logger.addHandler(logging.StreamHandler())
# this handler will print to the console
```

The logs follow the structure `{node_name}::{task_id}::{actor}::{event_type}::{message}`, omitting empty sections.

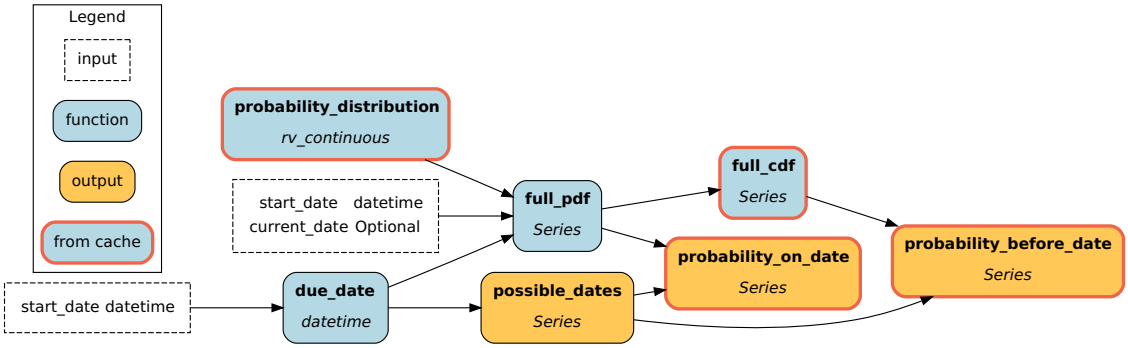
```
# example INFO logs for nodes foo, bar, and baz
foo::result_store::get_result::hit
bar::adapter::execute_node
baz::adapter::execute_node
```

Visualization

After `Driver` execution, calling `dr.cache.view_run()` will create a visualization of the dataflow with results retrieved from the cache highlighted.

By default, it shows the latest run, but it's possible to view previous runs by passing a `run_id`. Specify a `output_file_path` to save the visualization.

```
# ... define and execute a `Driver`

# select the 3rd unique run_id
run_id_3 = dr.cache.run_ids[2]
dr.cache.view_run(run_id=run_id_3,
output_file_path="cached_run_3.png")
```



Visualization produced by `dr.cache.view_run()`. Retrieved results are outlined.

Note

The method `.view_run()` doens't currently support task-based execution or `Parallelizable/Collect`.

Structured logs

Structured logs are stored on the `Driver.cache` and can be inspected programmatically. By setting `.with_cache(log_to_file=True)`, structured logs will also be appended to a `.jsonl` file as they happen; this is ideal for production usage.

To access log, use `Driver.cache.logs()`. You can `.logs(level=...)` to `"info"` or `"debug"` to view only `GET_RESULT` and `EXECUTE_NODE` or all events. Specifying `.logs(run_id=...)` will return logs from a given run, and leaving it empty will returns logs for all executions of this `Driver`.

```
dr.execute(...)
dr.cache.logs(level="info")
```

The shape of the returned object is slightly diffrent if specifying a `run_id` or not. Specifying a `run_id` will give `{node_name: List[CachingEvent]}`

Requesting `Driver.cache.logs()` will return a dictionary with `run_id` as key and list of `CachingEvent` as values `{run_id: List[CachingEvent]}`. This is useful for comparing run and verify nodes were properly executed or retrieved.

```
dr.cache.logs(level="debug", run_id=dr.cache.last_run_id)
# {
#     'raw_data': [CachingEvent(...), ...],
#     'processed_data': [CachingEvent(...), ...],
#     'amount_per_country': [CachingEvent(...), ...]
# }

dr.cache.logs(level="debug")
# {
#     'run_id_1': [CachingEvent(...), ...],
#     'run_id_2': [CachingEvent(...), ...]
# }
```

Note

When using `Parallelizable/Collect`, nodes part of the "parallel branches" will have a `task_id` key too `{node_name: {task_id: List[CachingEvent]}}` while nodes outside branches will remain `{node_name: List[CachingEvent]}`

Cached result format

By default, caching uses the `pickle` format because it can accomodate almost all Python objects. Although, it has caveats. The `cache` decorator allows you to use a different format for a given node (`JSON`, `CSV`, `Parquet`, etc.).

The next snippet caches `clean_dataset` as `parquet`, and `statistics` as `json`. These formats maybe more reliable, efficient, and easier to work with.

```
# my_dataflow.py
import pandas as pd
from hamilton.function_modifiers import cache

def raw_data(path: str) -> pd.DataFrame:
    return pd.read_csv(path)

@cache(format="parquet")
def clean_dataset(raw_data: pd.DataFrame) -> pd.DataFrame:
    raw_data = raw_data.fillna(0)
```

```
    return raw_data

@cache(format="json")
def statistics(clean_dataset: pd.DataFrame) -> dict:
    return ...
```

```
import driver
import my_dataflow

dr = (
    driver.Builder()
    .with_modules(my_dataflow)
    .with_cache()
    .buid()
)

# first execution will product a ``parquet`` file for
``clean_dataset``
# and a ``json`` file for ``statistics``
dr.execute(["statistics"])
# second execution will use these parquet and json files when loading
results
dr.execute(["statistics"])
```

Note

Internally, this uses Materializers

Caching behavior

The **caching behavior** refers to the caching logic used to: - version data - load and store metadata - load and store results - execute or not a node

The `DEFAULT` behavior aims to be easy to use and facilitate iterative development. However, other behavior may be desirble in particular scenarios or when going to production. The behavior can be set node-by-node.

1. `DEFAULT`: Try to retrieve results from cache instead of executing the node. Node result and metadata are stored.

2. `RECOMPUTE`: Always execute the node / never retrieve from cache. Result and metadata are stored. This can be useful to ensure external data is alawys reloaded.

3. `DISABLE`: Act as if caching isn't enabled for this node. Nodes depending on a disabled node will miss metadata for cache retrieval, forcing their re-execution. Useful for disabling caching in parts of the dataflow.

4. `IGNORE`: Similar to **Disable**, but downstream nodes will ignore the missing metadata and can successfully retrieve results. Useful to ignore "irrelevant" nodes that shouldn't impact the results (e.g., credentials, API clients, database connections).

See also

Learn more in the Caching logic reference section.

Note

There are other caching behaviors theoretically possible, but these four should cover most cases. Let us know if you have a use case that is not covered.

Setting caching behavior

The caching behavior can be specified at the node-level via the `@cache` function modifier or at the builder-level via `.with_cache(...)` arguments. Note that the behavior specified by the `Builder` will override the behavior from `@cache` since it's closer to execution.

via `@cache`

Below, we set `raw_data` to `RECOMPUTE` because the file it loads data from may change between executions. After executing and versioning the result of `raw_data`, if the data didn't change from previous execution, we'll be able to retrieve `clean_dataset` and `statistics` from cache.

```
# my_dataflow.py
import pandas as pd
from hamilton.function_modifiers import cache

@cache(behavior="recompute")
def raw_data(path: str) -> pd.DataFrame:
    return pd.read_csv(path)

def clean_dataset(raw_data: pd.DataFrame) -> pd.DataFrame:
    raw_data = raw_data.fillna(0)
    return raw_data
```

```
def statistics(clean_dataset: pd.DataFrame) -> dict:
    return ...
```

via `Builder().with_cache()`

Equivalently, we could set this behavior via the `Builder`. You can pass a list of node names to the keyword arguments `recompute`, `ignore`, and `disable`. Using `True` to enable that behavior for all nodes. For example, using `recompute=True` will force execution of all nodes and store their results in cache. Having `disable=True` is equivalent to not having the `.with_cache()` clause.

```
from hamilton import driver
import my_dataflow

dr = (
    driver.Builder()
    .with_modules(my_dataflow)
    .with_cache(recompute=["raw_data"])
    .build()
)
```

Set a default behavior

By default, caching is "opt-out" meaning all nodes are cached unless specified otherwise. To make it "opt-in", where only the specified nodes are cached, set `default_behavior="disable"`. You can also try different default behaviors.

```
from hamilton import driver
import my_dataflow

dr = (
    driver.Builder()
    .with_modules(my_dataflow)
    .with_cache(
        default=["raw_data", "statistics"],  # set behavior DEFAULT
        default_behavior="disable"  # all other nodes are DISABLE
    )
    .build()
)
```

Code version

The `code_version` of a node is determined by hashing its source code, ignoring docstring and comments.

Importantly, Apache Hamilton will not version nested function calls. If you edit utility functions or upgrade Python libraries, the cache might incorrectly assume the code to be the same.

For example, take the following function `foo`:

```
def _increment(x):
    return x + 1

def foo():
    return _increment(13)

# foo's code version:
129064d4496facc003686e0070967051ceb82c354508a58440910eb82af300db
```

Despite editing the nested `_increment()`, we get the same `code_version` because the content of `foo()` hasn't changed.

```
def _increment(x):
    return x + 2

def foo():
    return _increment(13)

# foo's code version:
129064d4496facc003686e0070967051ceb82c354508a58440910eb82af300db
```

In that case, `foo()` should return `13 + 2` instead of `13 + 1`. Unaware of the change in `_increment()`, the cache will find a `cache_key` match and return `13 + 1`.

A solution is to set the caching behavior to `RECOMPUTE` to force execute `foo()`. Another is to delete stored metadata or results to force re-execution.

Data version

Caching requires the ability to uniquely identify data (e.g., create a hash). By default, all Python primitive types (`int`, `str`, `dict`, etc.) are supported and more types can be added via extensions (e.g., `pandas`). For types not explicitly supported, caching can still function by versioning the object's internal `__dict__` instead. However, this could be expensive to compute or less reliable than alternatives.

Recursion depth

To version complex objects, we recursively hash its values. For example, versioning an object `List[Dict[str, float]]` involves hashing all keys and values of all dictionaries. Versioning complex objects with large `__dict__` state can become expensive.

In practice, we need to need a maximum recursion depth because there's a trade-off between the computational cost of hashing data and how accurately it uniquely identifies data (reduce hashing collisions).

Here's how to set the max depth:

```
from hamilton.io import fingerprinting
fingerprinting.set_max_depth(depth=3)
```

Support additional types

Additional types can be supported by registering a hashing function via the module `hamilton.io.fingerprinting`. It uses @functools.singledispatch to register the hashing function per Python type. The function must return a `str`. The code snippets shows how to support polars `DataFrame`:

```
import polars as pl
from hamilton.io import fingerprinting

# specify the type via the decorator
@fingerprinting.hash_value.register(pl.DataFrame)
def hash_polars_dataframe(obj, *args, **kwargs) -> str:
    """Convert a polars dataframe to a list of row hashes, then hash
the list.
    We consider that row order matters.
    """
    # obj is of type `pl.DataFrame`
    hash_per_row = obj.hash_rows(seed=0)
    # fingerprinting.hash_value(...) will automatically hash
primitive Python types
    return fingerprinting.hash_value(hash_per_row)
```

Alternatively, you can register functions without using decorators.

```
from hamilton.io import fingerprinting

def hash_polars_dataframe(obj, *args, **kwargs) -> str: ...

fingerprinting.hash_value.register(pl.DataFrame,
hash_polars_dataframe)
```

If you want to override the base case, the one defined by the function `hash_value()`, you can do so by registering a function for the type `object`.

```
@fingerprinting.hash_value.register(object)
def hash_object(obj, *args, **kwargs) -> str: ...
```

Storage

The caching feature is powered by two data storages:

- **Metadata store**: It contains information about past `Driver` executions (**code version**, **data version**, run id, etc.). From this metadata, Apache Hamilton determines if a node needs to be executed or not. This metadata is generally lightweight.
- **Result store**: It's a key-value store that maps a **data version** to a **result**. It's completely unaware of nodes, executions, etc. and simply holds the **results**. The result store can significantly grow in size depending on your usage. By default, all results are pickled, but other formats are possible.

Setting the cache path

By default, the **metadata** and **results** are stored under a new subdirectory `./.hamilton_cache/`, next to the current directory. Alternatively, you can set a path via `.with_cache(path=...)` that will be applied to both stores.

By project

Centralizing your cache by project is useful when you have nodes that are reused across multiple dataflows (e.g., training and inference ML pipelines, feature engineering).

```
# training_script.py
from hamilton import driver
import training

cache_path = "/path/to/project/hamilton_cache"
train_dr =
driver.Builder().with_modules(training).with_cache(path=cache_path).build()

# inference_script.py
from hamilton import driver
import inference

cache_path = "/path/to/project/hamilton_cache"
predict_dr =
driver.Builder().with_modules(inference).with_cache(path=cache_path).build()
```

Globally

Using a global cache is easier storage management. Since the metadata and the results for *all* your Apache Hamilton dataflows are in one place, it can be easier to cleanup disk space.

```
import pathlib
from hamilton import driver
import my_dataflow

# set the cache under the user's global directory for any operating
system
# The `Path` is converted to a string.
cache_path = str(pathlib.expanduser().joinpath("/.hamilton_cache"))
dr =
driver.Builder().with_module(my_dataflow).with_cache(path=cache_path).build()
```

Hint

It can be a good idea to store the cache path in an environment variable.

Separate locations

If you want the metadata and result stores to be at different location, you can instantiate and pass them to `.with_cache()`. In that case, `.with_cache()`'s `path` parameter will be ignored.

```
from hamilton import driver
from hamitlon.io.store import SQLiteMetadataStore, ShelveResultStore

metadata_store = SQLiteMetadataStore(path="~/.hamilton_cache")
result_store = ShelveResultStore(path="/path/to/my/project")

dr = (
    driver.Builder()
    .with_modules(dataflow)
    .with_cache(
        metadata_store=metadata_store,
        result_store=result_store,
    )
    .build()
)
```

Inspect storage

It is possible to directly interact with the metadata and result stores either by creating them or via `Driver.cache`.

```
from hamilton.caching.stores.sqlite import SQLiteMetadataStore
from hamilton.caching.stores.file import FileResultStore

metadata_store = SQLiteMetadataStore(path="~/.hamilton_cache")
result_store = FileResultStore(path="/path/to/my/project")

metadata_store.get(context_key=...)
result_store.get(data_version=...)
```

```
from hamilton import driver
import my_dataflow

dr = (
    driver.Builder()
    .with_modules(dataflow)
    .with_cache()
    .build()
)

dr.cache.metadata_store.get(context_key=...)
dr.cache.result_store.get(data_version=...)
```

A useful pattern is using the `Driver.cache` state or *structured logs <caching-structured-logs>* to retrieve a **data version** and query the **result store**.

```
from hamilton import driver
from hamilton.caching.adapter import CachingEventType
import my_dataflow

dr = (
    driver.Builder()
    .with_modules(dataflow)
    .with_cache()
    .build()
)

dr.execute(["amount_per_country"])

# via `cache.data_versions`; this points to the latest run
data_version = dr.cache.data_versions["amount_per_country"]
stored_result = dr.cache.result_store.get(data_version)

# via structured logs; this allows to query any run
run_id = ...
for event in dr.cache.logs(level="debug")[run_id]:
    if (
        event.event_type == CachingEventType.SET_RESULT
        and event.node_name == "amount_per_country"
```

```
    ):
        data_version = event.value
        break

stored_result = dr.cache.result_store(data_version)
```

In-memory

You can enable in-memory caching by using the `InMemoryMetadataStore` and `InMemoryResultStore`. Caching behaves the same, but metadata and results are never persisted to disk. This is useful in notebooks and interactive sessions where results are only temporary relevant (e.g., experimentating with new features).

⚠ Warning

In-memory caching can quickly fill memory. We suggest selectively caching results to limit this issue.

```
from hamilton import driver
from hamilton.caching.stores.memory import InMemoryMetadataStore,
InMemoryResultStore
import dataflow

dr = (
    driver.Builder()
    .with_modules(dataflow)
    .with_cache(
        metadata_store=InMemoryMetadataStore(),
        result_store=InMemoryResultStore(),
    )
    .build()
)
```

In-memory stores also allow you to persist your entire in-memory session to disk or start your in-memory session by loading an existing cache. This is compatible with most implementations.

Persist cache

This snippet shows how to persist an in-memory cache to an sqlite-backed metadata store and a file-based result store. Note that you should persist both the metadata and results stores for this to be useful. The `.persist_to()` method will repeatedly call `.set()` on the destination store. Persisting multiple times will add to the already cached data.

```
from hamilton import driver
from hamilton.caching.stores.sqlite import SQLiteMetadataStore
from hamilton.caching.stores.file import FileResultStore
from hamilton.caching.stores.memory import InMemoryMetadataStore,
InMemoryResultStore
import my_dataflow

dr = (
    driver.Builder()
    .with_modules(my_dataflow)
    .with_cache(
        metadata_store=InMemoryMetadataStore(),
        result_store=InMemoryResultStore(),
    )
    .build()
)

# execute the Driver several time. This will populate the in-memory
stores
dr.execute(...)

# persist to disk
dr.cache.metadata_store.persist_to(SQLiteMetadataStore(path="./.hamilton_cache")
dr.cache.result_store.persist_to(FileResultStore(path="./.hamilton_cache"))
```

Load cache

This snippet loads in-memory data from persisted metadata and result stores. The `.load_from()` is a classmethod and returns an instance of the in-memory store. The method `InMemoryResultStore.load_from(...)` must receive as argument a result store, but also a metadata store or a list of `data_version` to load. This is because `ResultStore` implementations don't have a registry of stored results.

```
from hamilton import driver
from hamilton.caching.stores.sqlite import SQLiteMetadataStore
from hamilton.caching.stores.file import FileResultStore
from hamilton.caching.stores.memory import InMemoryMetadataStore,
InMemoryResultStore
import my_dataflow

# create persisted stores
metadata_store = SQLiteMetadataStore(path="./.hamilton_cache")
result_store = FileResultStore(path="./.hamilton_cache")

dr = (
    driver.Builder()
    .with_modules(my_dataflow)
    .with_cache(
```

```
        # create in-memory stores by loading from persisted store

metadata_store=InMemoryMetadataStore.load_from(metadata_store),
        result_store=InMemoryResultStore.load_from(
            result_store=result_store,
            metadata_store=metadata_store,
        ),
    )
    .build()
)
```

Roadmap

Caching is a significant Apache Hamilton feature and there are plans to expand it. Here are some ideas and areas for development. Feel free comment on them or make other suggestions via Slack or GitHub!

- **Apache Hamilton UI integration**: caching introduces the concept of `data_version`. This metadata could be captured by the Apache Hamilton UI to show how different values are used across dataflow executions. This would be particularly useful for experiment tracking and lineage.
- **Distributed caching support**: the initial release supports multithreading and multiprocessing on a single machine. For distributed execution, we will need `ResultStore` and `MetadataStore` that can be remote and are safe for concurrent access.
- **Integrate with remote execution** (Ray, Skypilot, Modal, Runhouse): facilitate a pattern where the dataflow is executed locally, but some nodes can selectively be executed remotely and have their results cached locally.
- **async support**: Support caching with `AsyncDriver`. This requires a significant amount of code, but the core logic shouldn't change much.
- **cache eviction**: Allow to set up a max storage (in size or number of items) or time-based policy to delete data from the metadata and result stores. This would help with managing the cache size.
- **more store backends**: The initial release includes backend supported by the Python standard library (SQLite metadata and file-based results). Could support more backends via fsspec (AWS, Azure, GCP, Databricks, etc.)
- **support more types**: Include specialized hashing functions for complex objects from popular libraries. This can be done through Apache Hamilton extensions.

Function modifiers (Advanced)

> **⚠ Warning**
>
> This page is a work in progress. Refer to the API reference for more documentation and please ask a public question on Slack if you need help!

The page Function modifiers details how to use decorators to write expressive dataflows. The e presented function modifiers are highly expressive and should be sufficient in the large majority of cases.

Nonetheless, there exists higher level abstractions for power users that *may* be useful for integrations with your existing platform. If you want to use complex machinery instead of writing 1 additional function, comeback when it's your 10th manually addition 🛠️

This page assumes an advanced understanding of Apache Hamilton and will cover:

- @pipe
- @subdag
- @parameterize_subdag
- @resolve

Dynamic DAGs/Parallel Execution

There are two approaches to parallel execution in Apache Hamilton:

1. Using an adapter that submits each node/function to a system that handles execution, e.g. ray, dask, async, or a threadpool.
2. Using the *Parallelizable[]* and *Collect[]* types + delegating to an executor.

Using an Adapter

The adapter approach effectively farms out the execution of each node/function to a system that can handle resolving futures. That is, Apache Hamilton walks the DAG and submits each node to the adapter, which then submits the node for execution, and internally the execution resolves any Futures from prior submitted nodes.

To make use of this, the general pattern is you apply an adapter to the driver and don't need to touch your Hamilton functions!:

```
from hamilton import driver
from hamilton.execution import executors
from hamilton.plugins.h_threadpool import FutureAdapter
# from hamilton.plugins.h_ray import RayGraphAdapter
# from hamilton.plugins.h_dask import DaskGraphAdapter

dr = (
    driver.Builder()
    .with_modules(foo_module)
    .with_adapter(FutureAdapter())
    .build()
)

dr.execute(["my_variable"], inputs={...}, overrides={...})
```

The code above will execute the DAG submitting to a *ThreadPoolExecutor* (see h_threadpool.FutureAdapter), which is great if you're doing a lot of I/O bound work, e.g. making API calls, reading from a database, etc.

See this Threadpool based example for a complete example.

Other adapters, e.g. Ray h_ray.RayGraphAdapter, Dask h_dask.DaskGraphAdapter, etc... will submit to their respective executors, but will involve object serialization (see caveats below).

Using the *Parallelizable[]* and *Collect[]* types

Apache Hamilton now has pluggable execution, which allows for the following:

1. Grouping of nodes into "tasks" (discrete execution unit between serialization boundaries)

2. Executing the tasks in parallel, using any executor of your choice

You can run this executor using the *Builder*, a utility class that allows you to build a driver piece by piece. Note that you currently have to call *enable_dynamic_execution(allow_experimental_mode=True)* which will toggle it to use the *V2* executor. Then, you can:

1. Add task executors to specify how to run the tasks

2. Add node grouping strategies

3. Add modules to crawl for functions

4. Add a results builder to shape the results

Either constructing the driver, or using the builder and *not* calling *enable_dynamic_execution* will give you the standard executor. We highly recommend you use the builder pattern – while the constructor of the *Driver* will be fully backwards compatible according to the rules of semantic versioning, we may change it in the future (for 2.0).

Note that the new executor is required to handle dynamic creation of nodes (E.G. using *Parallelizable[]* and *Collect[]*.

Let's look at an example of the driver:

```
from my_code import foo_module, bar_module

from hamilton import driver
from hamilton.execution import executors

dr = (
    driver.Builder()
    .with_modules(foo_module)
    .enable_dynamic_execution(allow_experimental_mode=True)
    .with_config({"config_key": "config_value"})
    .with_local_executor(executors.SynchronousLocalTaskExecutor())
    .with_remote_executor(executors.MultiProcessingExecutor(max_tasks=5))
    .build()
)

dr.execute(["my_variable"], inputs={...}, overrides={...})
```

Note that we set a *remote* executor, and a local executor. While you can bypass this and instead set an *execution_manager* in the builder call (see Builder for documentation on the *Builder*),this goes along with the default grouping strategy, which is to place each node in its own group, except for dynamically generated (*Parallelizable[]*) blocks, which are each made into one group, and executed locally.

Thus, when you write a DAG like this (a simple map-reduce pattern):

```
from hamilton.htypes import Parallelizable, Collect

def url() -> Parallelizable[str]:
    for url_ in  _list_all_urls():
        yield url_

def url_loaded(url: str) -> str:
    return _load(urls)

def counts(url_loaded: str) -> int:
    return len(url_loaded.split(" "))
```

```
def total_words(counts: Collect[int]) -> int:
    return sum(counts)
```

The block containing *counts* and *url_loaded* will get marked as one task, repeated for each URL in url_loaded, and run on the remote executor (which in this case is the *ThreadPoolExecutor*).

Note that we currently have the following caveats:

1. No nested *Parallelizable[]*/*Collect[]* blocks – we only allow one level of parallelization
2. Serialization for *Multiprocessing* is suboptimal – we currently use the default *pickle* serializer, which breaks with certain cases. Ray, Dask, etc... all work well, and we plan to add support for joblib + cloudpickle serialization.
3. *Collect[]* input types are limited to one per function – this is another caveat that we intend to get rid of, but for now you'll want to concat/put into one function before collecting.

Known Caveats

If you're familiar with multi-processing then these caveats will be familiar to you. If not, then you should be aware of the following:

Serialization

Challenge:

- Objects are by default pickled and sent to the remote executor, and then unpickled.
- This can be slow, and can break with certain types of objects, e.g. OpenAI Client, DB Client, etc.

Solution:

- Make sure that your objects are serializable.
- If you're using a library that doesn't support serialization, then one option is to have Apache Hamilton instantiate the object in each parallel block. You can do this by making the code depend on something within the parallel block.
- Another option is write a custom wrapper function that uses *__set_state__* and *__get_state__* to serialize and deserialize the object.
- See this issue for details and possible features to make this simpler to deal with.

Multiple Collects

Currently, by design (see all limitations here), you can only have one "collect" downstream of "parallel".

So the following code WILL NOT WORK:

```
import logging

from hamilton import driver
from hamilton.execution.executors import SynchronousLocalTaskExecutor
from hamilton.htypes import Collect, Parallelizable
import pandas as pd


ANALYSIS_OB = tuple[tuple[str,...], pd.DataFrame]
ANALYSIS_RES = dict[str, str | float]


def split_by_cols(full_data: pd.DataFrame, columns: list[str]) ->
Parallelizable[ANALYSIS_OB]:
    for idx, grp in full_data.groupby(columns):
        yield (idx, grp)


def sub_metric_1(split_by_cols: ANALYSIS_OB, number: float=1.0) ->
ANALYSIS_RES:
    idx, grp = split_by_cols
    return {"key": idx, "mean": grp["spend"].mean() + number}


def sub_metric_2(split_by_cols: ANALYSIS_OB) -> ANALYSIS_RES:
    idx, grp = split_by_cols
    return {"key": idx, "mean": grp["signups"].mean()}


def metric_1(sub_metric_1: Collect[ANALYSIS_RES], columns:
list[str]) -> pd.DataFrame:
    data = [[k for k in d["key"]] + [d["mean"], "spend"] for d in
sub_metric_1]
    cols = list(columns) + ["mean", "metric"]
    return pd.DataFrame(data, columns=cols)


def metric_2(sub_metric_2: Collect[ANALYSIS_RES], columns:
list[str]) -> pd.DataFrame:
    data = [[k for k in d["key"]] + [d["mean"], "signups"] for d in
sub_metric_2]
    cols = list(columns) + ["mean", "metric"]
    return pd.DataFrame(data, columns=cols)
```

```
# this will not work because you can't have two Collect[] calls
downstream from a Parallelizable[] call
def all_agg(metric_1: pd.DataFrame, metric_2: pd.DataFrame) ->
pd.DataFrame:
    return pd.concat([metric_1, metric_2])


if __name__ == "__main__":
    from hamilton.execution import executors
    import __main__

    from hamilton.log_setup import setup_logging
    setup_logging(log_level=logging.DEBUG)

    local_executor = executors.SynchronousLocalTaskExecutor()

    dr = (
        driver.Builder()
        .enable_dynamic_execution(allow_experimental_mode=True)
        .with_modules(__main__)
        .with_remote_executor(local_executor)
        .build()
    )
    df = pd.DataFrame(
        index=pd.date_range('20230101', '20230110'),
        data={
            "signups": [1, 10, 50, 100, 200, 400, 700, 800, 1000,
1300],
            "spend": [10, 10, 20, 40, 40, 50, 100, 80, 90, 120],
            "region": ["A", "B", "C", "A", "B", "C", "A", "B", "C",
"X"],
        }
    )
    ans = dr.execute(
        ["all_agg"],
        inputs={
            "full_data": df,
            "number": 3.1,
            "columns": ["region"],
        }
    )
    print(ans["all_agg"])
```

To fix this, (this is documented in this issue) you can either create a new function that combines the two *Collect[]* calls that could be combined with @config.when.

```
def all_metrics(sub_metric_1: ANALYSIS_RES, sub_metric_2:
ANALYSIS_RES) -> ANALYSIS_RES:
    return ... # join the two dicts in whatever way you want
```

```
def all_agg(all_metrics: Collect[ANALYSIS_RES]) -> pd.DataFrame:
    return ... # join them all into a dataframe
```

Or you use @resolve, with @group (scroll down a little), @inject, to set what should be determined to be collected at DAG construction time:

```
@resolve(
    when=ResolveAt.CONFIG_AVAILABLE,
    decorate_with= lambda metric_names:
      inject( # this will annotate the function with @inject

# it will then inject a group of values corresponding to the sources
wanted
         sub_metrics=group(*[source(x) for x in metric_names])
      ),
)
def all_metrics(sub_metrics: list[ANALYSIS_RES], columns: list[str])
-> pd.DataFrame:
    frames = []
    for a in sub_metrics:
        frames.append(_to_frame(a, columns))
    return pd.concat(frames)

# then in your driver:
from hamilton import settings
_config = {settings.ENABLE_POWER_USER_MODE:True}
_config["metric_names"] = ["sub_metric_1", "sub_metric_2"]

# Then in the driver building pass in the configuration:
.with_config(_config)
```

UI Overview

Apache Hamilton comes with a fully open-source UI that can be run both for local deployment and on a remote server. The UI consists of the following features:

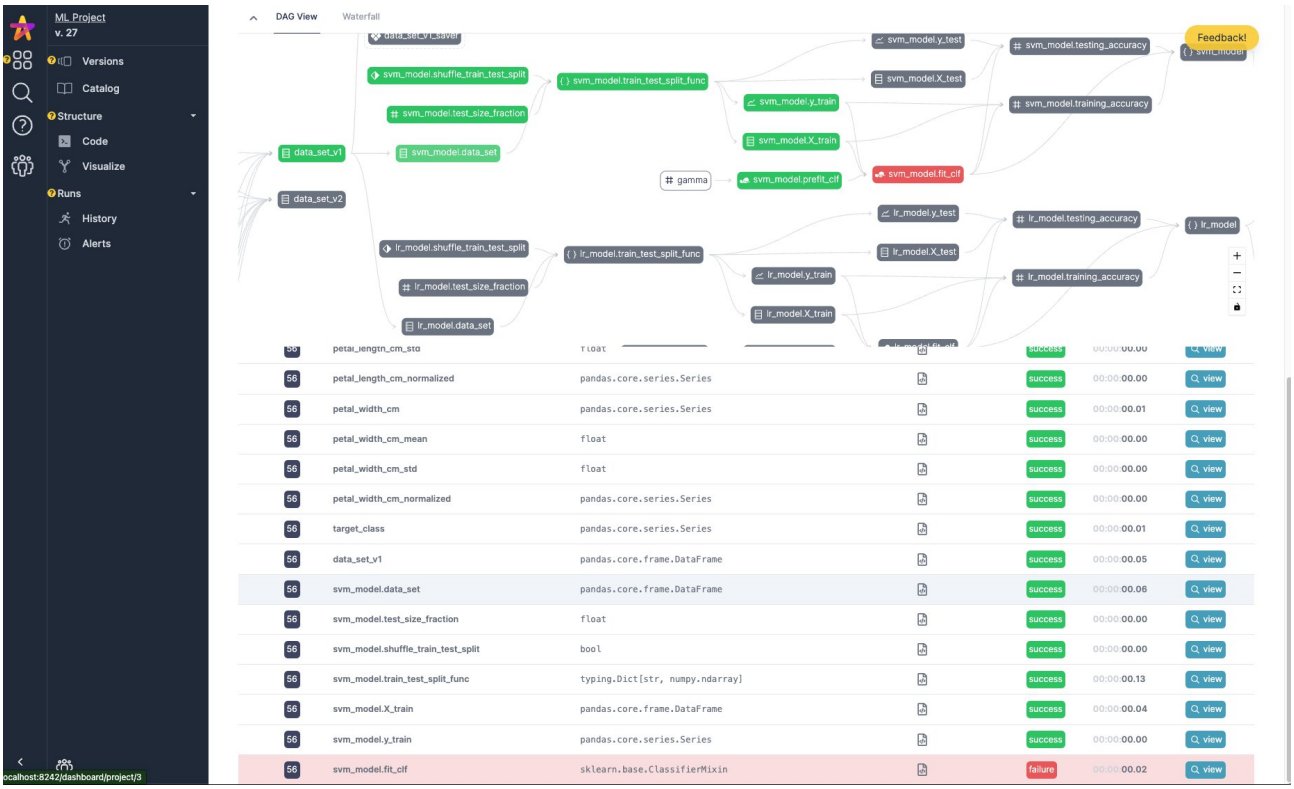
1. Telemetry for hamilton executions – both on the history of executions and the data itself.

2. A feature/artifact catalog for browsing/connecting executions of nodes -> results.

3. A dataflow (i.e. DAG) visualizer for exploring and looking at your code, and determining lineage.

4. A project explorer for viewing curating projects and viewing versions of your Apache Hamilton dataflows.

In short, the Apache Hamilton UI aims to combine a large swath of MLOps/data observability systems in one simple application.



—

The Apache Hamilton UI has two modes: 1. Run locally using sqlite3 2. Run on docker images with postgres (meant for deployment)

Local Mode

To run the hamilton UI in local mode, you can do the following:

```
pip install "sf-hamilton[ui,sdk]"
hamilton ui
# python -m hamilton.cli.__main__ ui # on windows
```

This will launch a browser window in localhost:8241. You can then navigate to the UI and start using it! While this can potentially handle a small production workflow, you may want to run on postgres with a separate frontend/backend/db for full scalability and a multi-read/write db.

Docker/Deployed Mode

The Apache Hamilton UI can be contained within a set of Docker images. You launch with docker-compose, and it will start up the UI, the backend server, and a Postgres database. If you'd like a quick overview of some of the features, you can watch the following:

Note: if you run into the "Invalid HTTP_HOST" error, then please set the environment variable *HAMILTON_ALLOWED_HOSTS="*"* (or comma separated list of domains of choice) for the backend docker container. You can inject this via *-e* or in the *docker-compose[-prod].yml* file itself.

Install

If you'd like a video walkthrough on getting set up, you can watch the following:

As prerequisites, you will need to have Docker installed – you can follow instructions here.

1. Clone the Apache Hamilton repository locally

```
git clone https://github.com/apache/hamilton
```

1. Navigate to the `hamilton/ui` directory

```
cd hamilton/ui
```

1. Execute the installation script with the following command

```
./run.sh
```

This will:

- Pull all Docker images from the Docker Hub
- Start a local Postgres database
- Start the backend server
- Start the frontend server

This takes a bit of time! So be patient. The server will be running on port 8242.

1. Then navigate to `http://localhost:8242` in your browser, and enter your email (this will be the username used within the app).

Building the Docker Images locally

If building the Docker containers from scratch, increase your Docker memory to 10gb or more – you can do this in the Docker Desktop settings.

To build the images locally, you can run the following command:

```
# from the hamilton/ui directory
./dev.sh --build
```

This will build the containers from scratch. If you just want to mount the local code, you can run just

```
./dev.sh
```

Self-Hosting

If you know docker, you should be good to go. The one environment variable to know is *HAMILTON_ALLOWED_HOSTS*, which you can set to * to allow all hosts, or a comma separated list of hosts you want to allow.

To host the UI on a subpath, set *REACT_APP_HAMILTON_SUB_PATH* to the subpath required. For example, to run on *https://domain.com/hamilton*:

```
- REACT_APP_HAMILTON_SUB_PATH=/hamilton
```

Make sure that the sub path environment variable begins with / if set.

Please reach out to us if you want to deploy on your own infrastructure and need help - join slack. More extensive self-hosting documentation is in the works, e.g. Snowflake, Databricks, AWS, GCP, Azure, etc.; we'd love a helm chart contribution!

Running on Snowflake

You can run the Apache Hamilton UI on Snowflake Container Services. For a detailed guide, see the blog post Observability of Python code and application logic with Apache Hamilton UI on Snowflake Container Services by Greg Kantyka and the Apache Hamilton Snowflake Example.

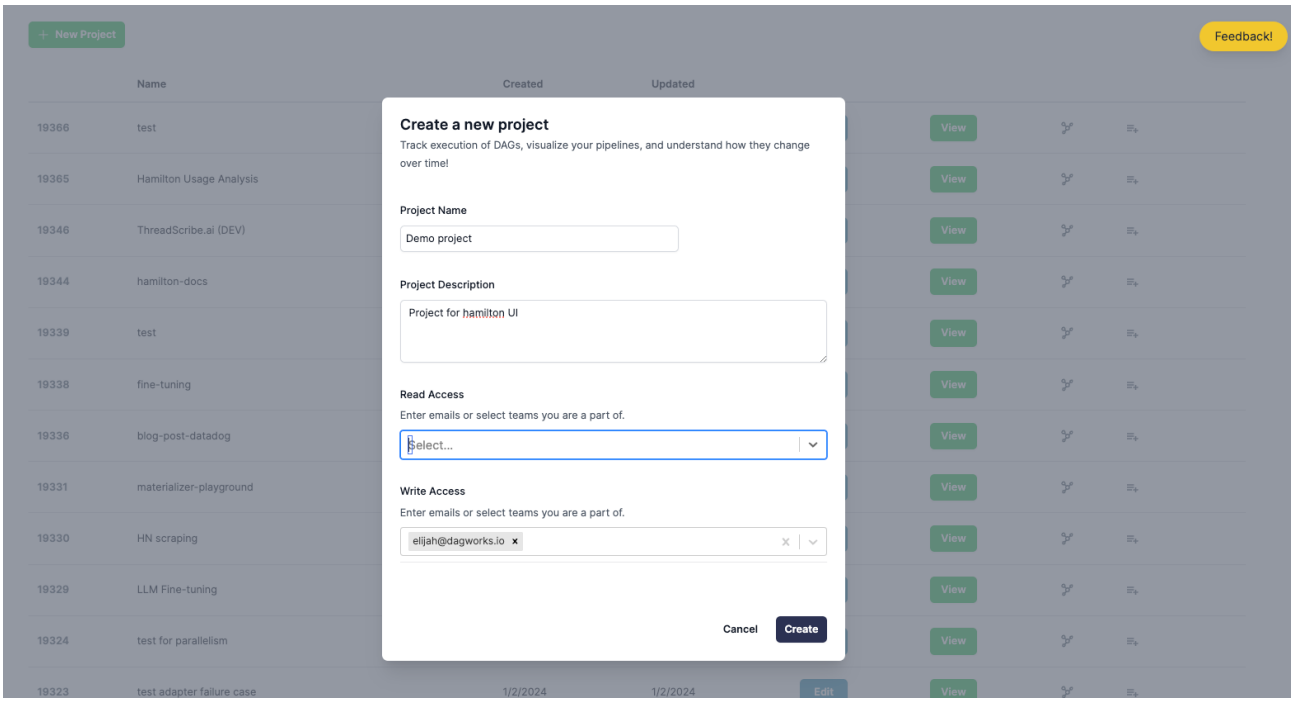
Get started

Now that you have your server running, you can run a simple dataflow and watch it in the UI! You can follow instructions in the UI when you create a new project, or follow the instructions here.

First, install the SDK:

```
pip install "sf-hamilton[sdk]"
```

Then, navigate to the project page (dashboard/projects), in the running UI, and click the green `+ New DAG` button.



Remember the project ID – you'll use it for the next steps.

Existing Apache Hamilton Code

Add the following adapter to your code if you have existing Apache Hamilton code:

```
from hamilton_sdk import adapters

tracker = adapters.HamiltonTracker(
   project_id=PROJECT_ID_FROM_ABOVE,
   username="USERNAME/EMAIL_YOU_PUT_IN_THE_UI",
   dag_name="my_version_of_the_dag",
   tags={"environment": "DEV", "team": "MY_TEAM", "version": "X"}
)

dr = (
  driver.Builder()
    .with_config(your_config)
    .with_modules(*your_modules)
    .with_adapters(tracker)
    .build()
)
```

Then run your DAG, and follow the links in the logs! Note that the link is correct if you're using the local mode – if you're on postgres it links to 8241 (but you'll want to follow it to 8241).

I need some Apache Hamilton code to run

If you don't have Apache Hamilton code to run this with, you can run Apache Hamilton UI example under examples/hamilton_ui:

```
# we assume you're in the Apache Hamilton repository root
cd examples/hamilton_ui
# make sure you have the right python packages installed
pip install -r requirements.txt
# run the pipeline providing the email and project_id you created in
the UI
python run.py --email <email> --project_id <project_id>
```

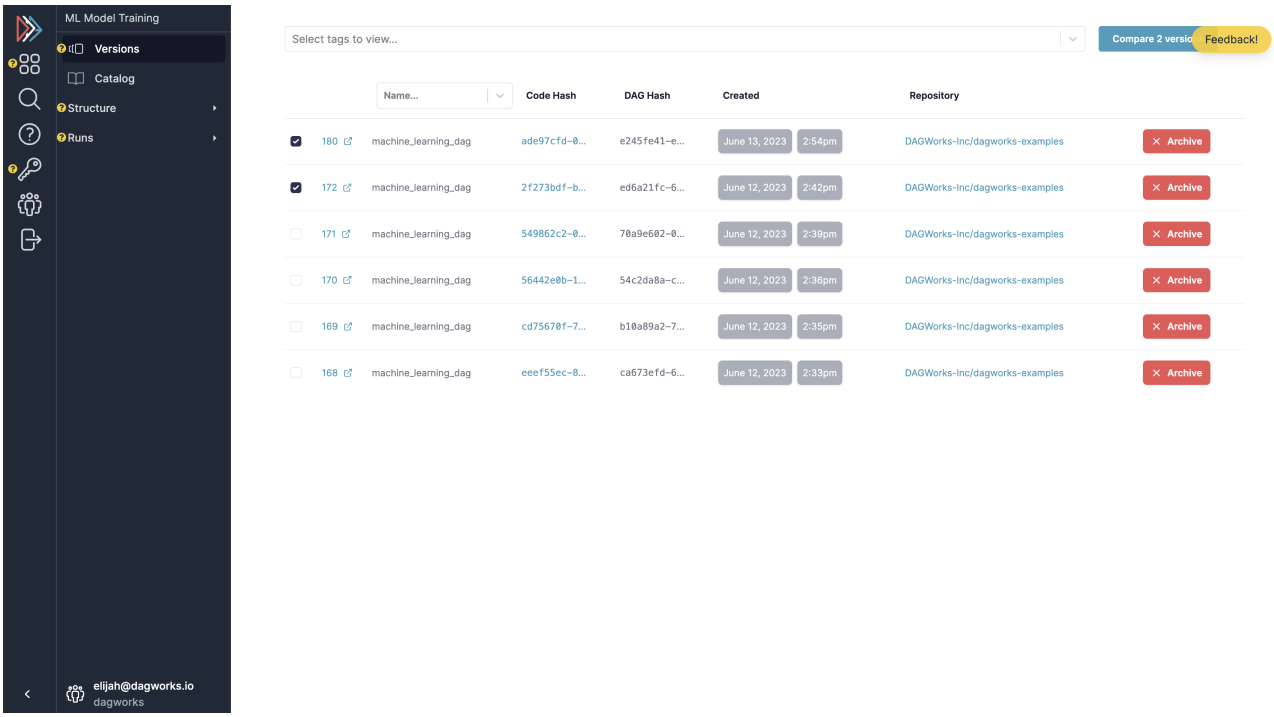
You should see links in the logs to the UI, where you can see the DAG run + the data summaries captured.

Features

Once you get to the UI, you can navigate to the projects page (left hand nav-bar). Assuming you have created a project and logged to it, you can then navigate to view it and then more details about it. E.g. versions, code, lineage, catalog, execution runs. See below for a few screenshots of the UI.

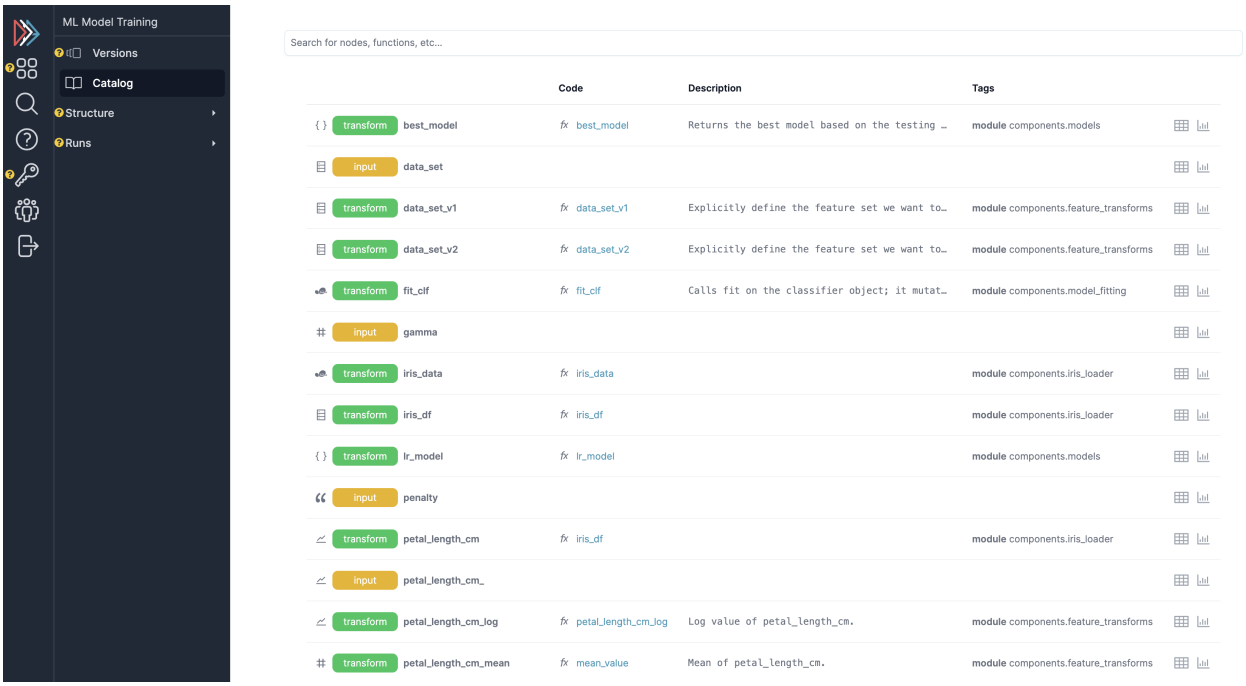
Dataflow versioning

Select a dataflow versions to compare and visualize.



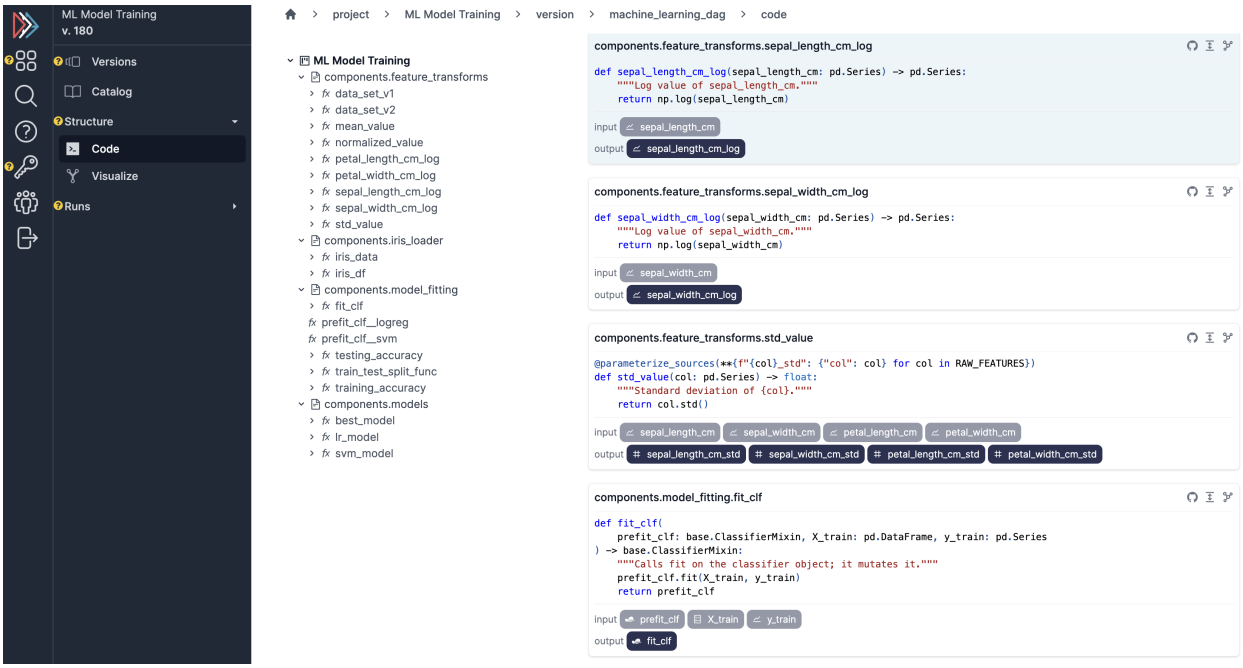
Assets/features catalog

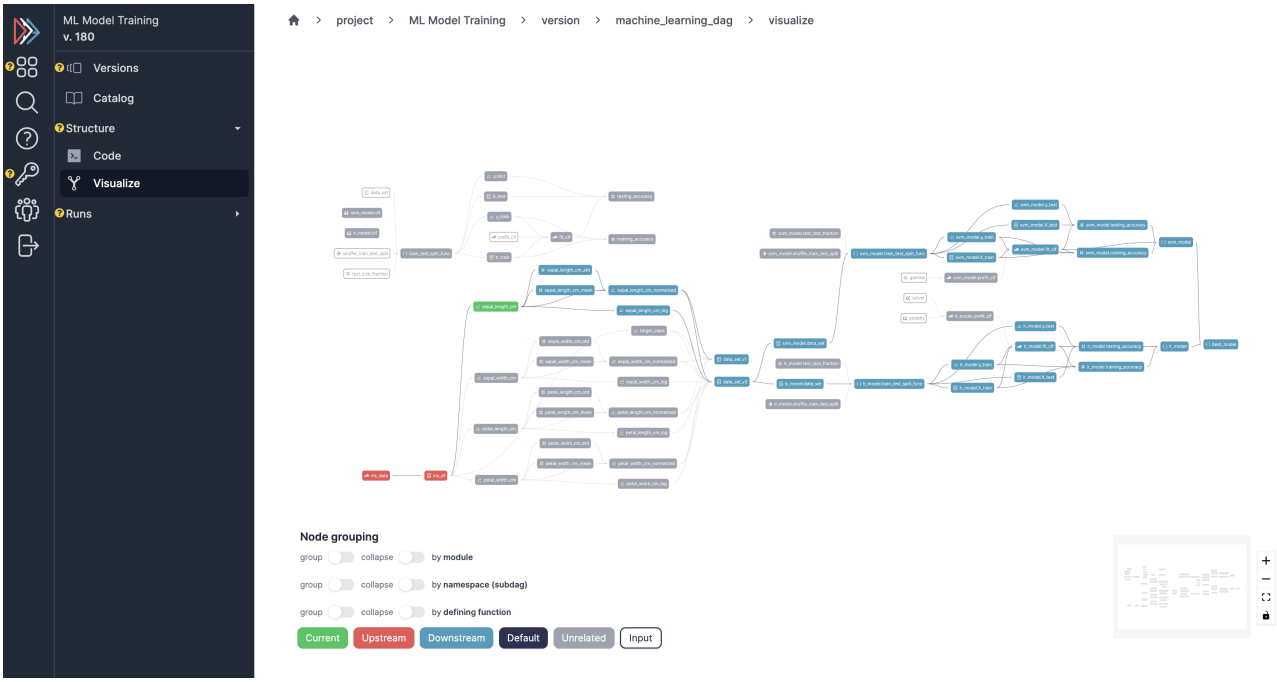
View functions, nodes, and assets across a history of runs.



Browser

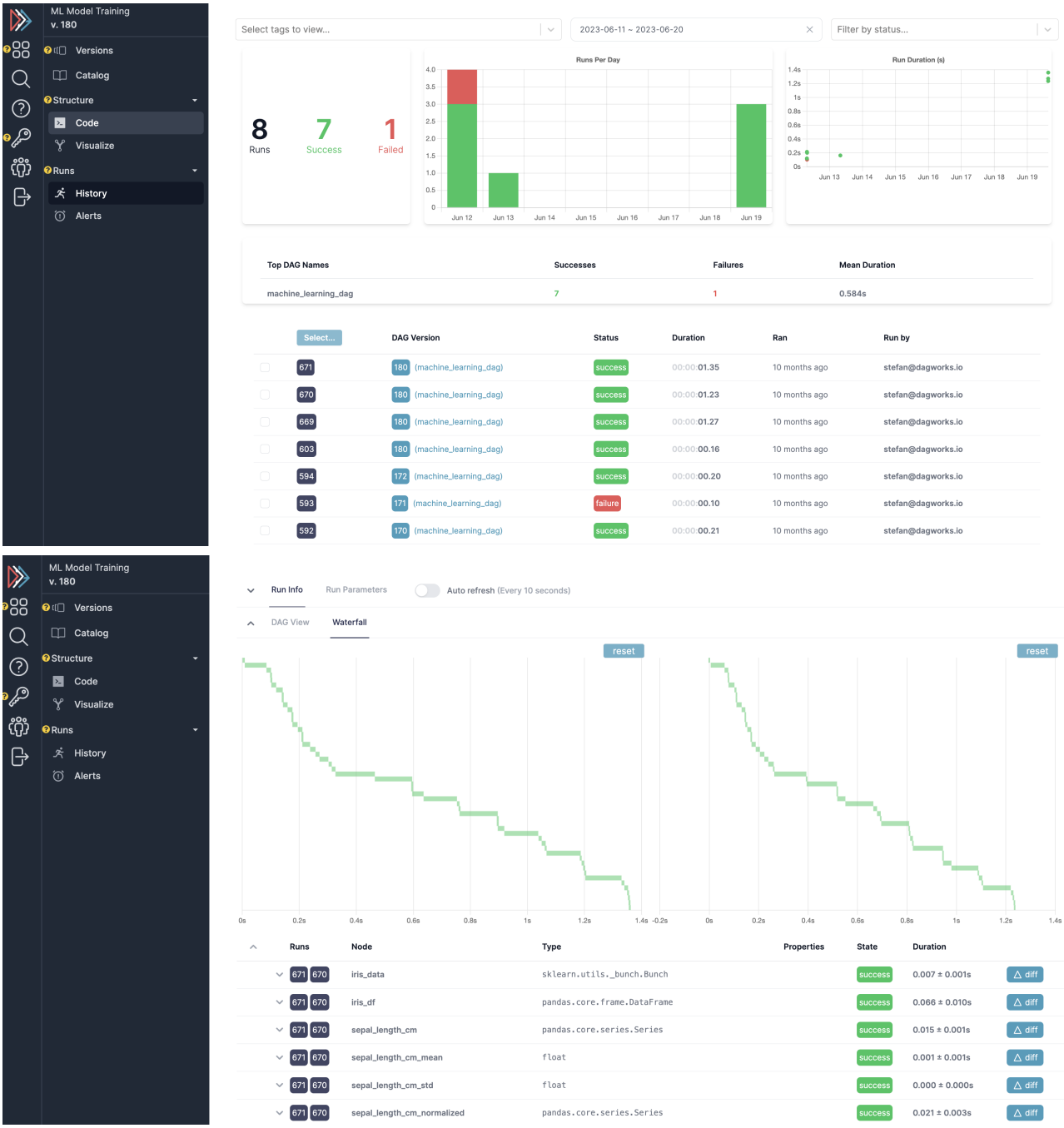
View dataflow structure and code.

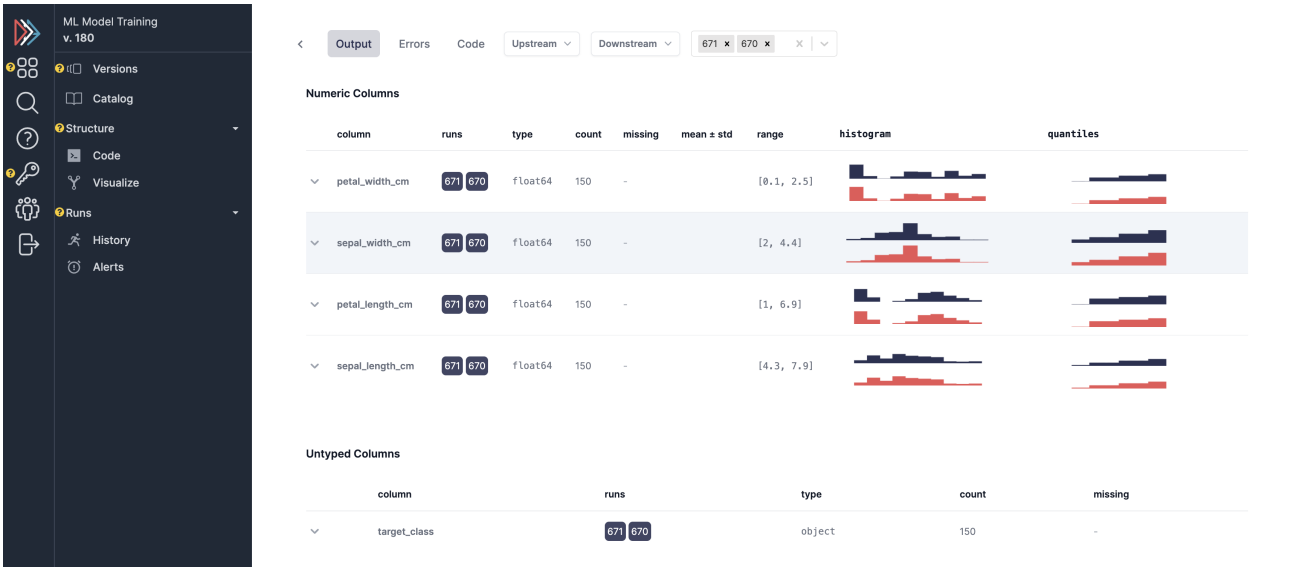




Run tracking + telemetry

View a history of runs, telemetry on runs/comparison, and data for specific runs:


ML Model Training
v. 180
Versions
Catalog
Structure
Code
Visualize
Runs
History
Alerts
Select tags to view...
2023-06-11 ~ 2023-06-20
Filter by status...
8
Runs
7
Success
1
Failed
Runs Per Day
Run Duration (s)
Top DAG Names
Successes
Failures
Mean Duration
machine_learning_dag
7
1
0.584s
Select...
DAG Version
Status
Duration
Ran
Run by
671 180 (machine_learning_dag) success 00:00:01.35 10 months ago stefan@dagworks.io
670 180 (machine_learning_dag) success 00:00:01.23 10 months ago stefan@dagworks.io
669 180 (machine_learning_dag) success 00:00:01.27 10 months ago stefan@dagworks.io
603 180 (machine_learning_dag) success 00:00:00.16 10 months ago stefan@dagworks.io
594 172 (machine_learning_dag) success 00:00:00.20 10 months ago stefan@dagworks.io
593 171 (machine_learning_dag) failure 00:00:00.10 10 months ago stefan@dagworks.io
592 170 (machine_learning_dag) success 00:00:00.21 10 months ago stefan@dagworks.io
Run Info
Run Parameters
Auto refresh (Every 10 seconds)
DAG View
Waterfall
reset
reset
Runs
Node
Type
Properties
State
Duration
671 670 iris_data sklearn.utils._bunch.Bunch success 0.007 ± 0.001s △ diff
671 670 iris_df pandas.core.frame.DataFrame success 0.066 ± 0.010s △ diff
671 670 sepal_length_cm pandas.core.series.Series success 0.015 ± 0.001s △ diff
671 670 sepal_length_cm_mean float success 0.001 ± 0.001s △ diff
671 670 sepal_length_cm_std float success 0.000 ± 0.000s △ diff
671 670 sepal_length_cm_normalized pandas.core.series.Series success 0.021 ± 0.003s △ diff



SDK Configuration

This section documents HamiltonTracker configuration options.

Changing where data is sent

You can change where telemetry is logged by passing in *hamilton_api_url* and/or *hamilton_ui_url* to the HamiltonTracker constructor. By default, these are set to *localhost:8241/8242*.

```
from hamilton_sdk import adapters

tracker = adapters.HamiltonTracker(
   project_id=PROJECT_ID_FROM_ABOVE,
   username="USERNAME/EMAIL_YOU_PUT_IN_THE_UI",
   dag_name="my_version_of_the_dag",
   tags={"environment": "DEV", "team": "MY_TEAM", "version": "X"},
   hamilton_api_url="http://YOUR_DOMAIN_HERE:8241",
   hamilton_ui_url="http://YOUR_DOMAIN_HERE:8242" # if using docker
the UI is on 8242.
)

dr = (
  driver.Builder()
    .with_config(your_config)
    .with_modules(*your_modules)
    .with_adapters(tracker)
    .build()
)
```

Changing behavior of what is captured

By default, a lot is captured and sent to the Apache Hamilton UI.

Here are a few options that can change that - these can be found in *hamilton_sdk.tracking.constants*. You can either change the defaults by directly changing the constants, by specifying them in a config file, or via environment variables.

Here we first explain the options:

Simple Invocation

Option	Default	Explanation
CAPTURE_DATA_STATISTICS	True	Whether to capture any data insights/ statistics
MAX_LIST_LENGTH_CAPTURE	50	Max length for list capture
MAX_DICT_LENGTH_CAPTURE	100	Max length for dict capture
DEFAULT_CONFIG_URI	~/.hamilton.conf	Default config file URI.

To change the defaults via a config file, you can do the following:

```
[SDK_CONSTANTS]
MAX_LIST_LENGTH_CAPTURE=100
MAX_DICT_LENGTH_CAPTURE=200

# save this to ~/.hamilton.conf
```

To change the defaults via environment variables, you can do the following, prefixing them with *HAMILTON_*:

```
export HAMILTON_MAX_LIST_LENGTH_CAPTURE=100
export HAMILTON_MAX_DICT_LENGTH_CAPTURE=200
python run_my_hamilton_code.py
```

To change the defaults directly, you can do the following:

```
from hamilton_sdk.tracking import constants

constants.MAX_LIST_LENGTH_CAPTURE = 100
constants.MAX_DICT_LENGTH_CAPTURE = 200

tracker = adapters.HamiltonTracker(
   project_id=PROJECT_ID_FROM_ABOVE,
   username="USERNAME/EMAIL_YOU_PUT_IN_THE_UI",
   dag_name="my_version_of_the_dag",
   tags={"environment": "DEV", "team": "MY_TEAM", "version": "X"}
)

dr = (
  driver.Builder()
    .with_config(your_config)
    .with_modules(*your_modules)
    .with_adapters(tracker)
    .build()
)
dr.execute(...)
```

In terms of precedence, the order is:

1. Module default.

2. Config file values.

3. Environment variables.

4. Directly set values.

Best Practices

A set of best-practices to help you get the most out of Apache Hamilton quickly and easily.

Function Naming

Here are three important points about function naming:

1. It enables you to define your Apache Hamilton dataflow.

2. It drives collaboration & code reuse.

3. It serves as documentation itself.

You don't need to get this right the first time – search and replace is really easy with Apache Hamilton code bases – but it is something to converge thinking on!

It enables you to define your Apache Hamilton dataflow

As the name of a hamilton function defines the name of the created artifact, naming is vital to a readable, extensible hamilton codebase. Names must mean something:

```
def foo_bar(input1: int, input2: pd.Series) -> pd.Series:
    """docs..."""
    ...
```

In this case, `foo_bar` is not helpful - it's unclear what this function produces at all. Remember you want function names to mean something, since that will enable clarity when using Apache Hamilton, what is being requested, and will help document what the function itself is doing.

It drives collaboration and reuse

When people come to encounter your code, they'll need to understand it, add to it, modify it, etc.

You'll want to ensure some standardization to enable:

1. Mapping business concepts to function names. E.g. That will help people to find things in the code that map to things that happen within your business.

2. Ensuring naming uniformity across the code base. People usually follow the precedent of the code around them, so if everything in a particular module for say, date features, has a `D_` prefix, then they will likely follow that naming convention. This is likely something you will iterate on – and it's best to try to converge on a team naming convention once you have a feel for the Hamilton functions being written by the team.

We suggest that long functions names that are separated by `_` aren't a bad thing. E.g. if you were to come across a function named `life_time_value` versus `ltv` versus `l_t_v`, which one is more obvious as to what it is and what it represents?

It serves as documentation itself

Remember your code usually lives a lot longer that you ever think it will. So our suggestion is to always err to the more obvious way of naming to ensure it's clear what a function represents.

Again, if you were to come across a function named `life_time_value` versus `ltv` versus ``l_t_v`, which one is more obvious as to what it is and what it represents?

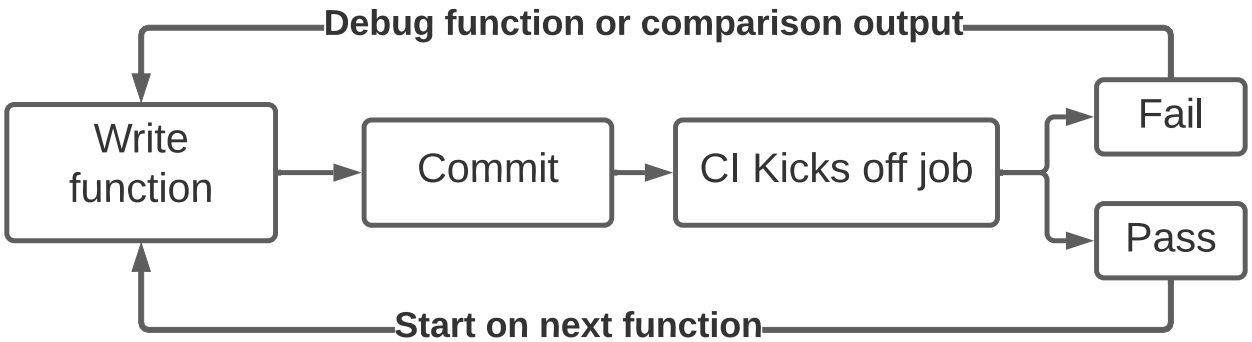
Migrating to Apache Hamilton

Here are two suggestions for helping you migrate to Apache Hamilton

Continuous Integration for Comparisons

Create a way to easily & frequently compare results.

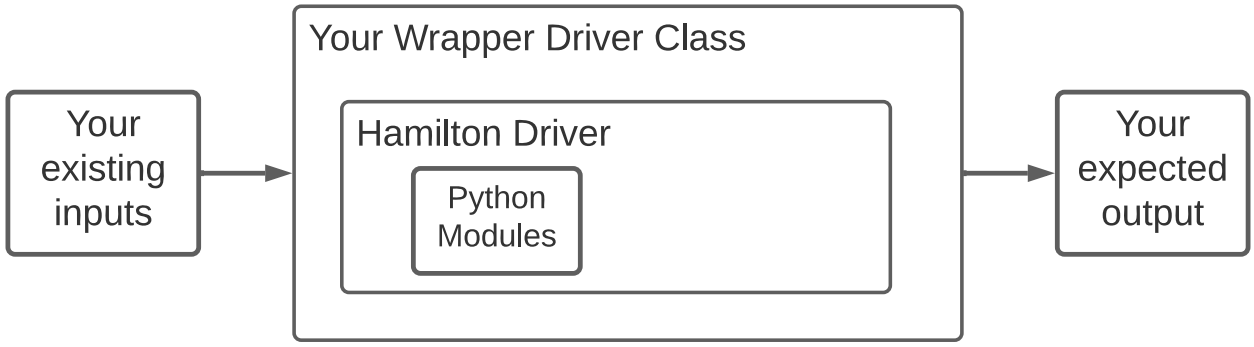
1. Integrate with continuous integration (CI) system if you can.
2. 🔎🐛 Having a means that tests code early & often will helps diagnose bugs in your old code (most likely) or your new implementation (less likely).
3. Specifically, have a system to compare the output of your Apache Hamilton code, to compare to the output of your existing system.



Integrate into your code base via a *"custom wrapper object"*

If you have existing systems that you want to integrate Apache Hamilton into, it might require non-trivial effort for you to change those systems to be able to use Apache Hamilton. If that's the case, then we suggest creating a "custom object" to "wrap" Apache Hamilton, so that it's easier to migrate to it.

Specifically, this custom wrapper object class's purpose is to match your existing API expectations. It will act as the translation layer from your existing API expectations, to what running Apache Hamilton requires, and back. In Apache Hamilton terminology, this is a *Custom Driver Wrapper*, since it wraps around the Hamilton Driver class.



This is a best practice because:

1. When migrating, it's best to avoid making too many changes. So don't change your API expectations if you can.
2. It allows you to easily insert Apache Hamilton into any context. Thereby minimizing potential migration problems.

Code Organization

Apache Hamilton will force you to organize your code! Here's some tips.

Apache Hamilton forces you to put your code into modules that are distinct from where you run your code.

You'll soon find that a single python module does not make sense, and so you'll organically start to (very likely) put like functions with like functions, i.e. thus creating domain specific modules –> *use this to your development advantage!*

At Stitch Fix we:

1. Use modules to model team thinking, e.g. date_features.py.
2. Use modules to helps isolate what you're working on.
3. Use modules to replace parts of your Apache Hamilton dataflow very easily for different contexts.

Team thinking

You'll need to curate your modules. We suggest orienting this around how teams think about the business.

E.g. marketing spend features should be in the same module, or in separate modules but in the same directory/package.

This will then make it easy for people to browse the code base and discover what is available.

Helps isolate what you're working on

Grouping functions into modules then helps set the tone for what you're working on. It helps set the "namespace", if you will, for that function. Thus you can have the same function name used in multiple modules, as long as only one of those modules is imported to build the DAG.

Thus modules help you create boundaries in your code base to isolate functions that you'll want to change inputs to.

Enables you to replace parts of your DAG easily for different contexts

The names you provide as inputs to functions form a defined "interface", to borrow a computer science term, so if you want to swap/change/augment an input, having a function that would map to it defined in another module(s) provides a lot of flexibility. Rather than having a single module with all functions defined in it, separating the functions into different modules could be a productivity win.

Why? That's because when you come to tell Apache Hamilton what functions constitute your dataflow (i.e. DAG), you'll be able to simply replace/add/change the module being passed. So if you want to compute inputs for certain functions differently, this composability of including/excluding modules, when building the DAG provides a lot of flexibility that you can exploit to make your development cycle faster.

Common Indices

If you're creating dataframes, then this will apply to you!

While Apache Hamilton is a general-purpose framework, we've found a common pattern is to manipulate datasets that have shared indices (spines) for creating dataframes.

Although this might not apply towards every use-case (E.G. more complex joins with spark dataframes), a large selection of use-cases can be enabled if every dataframe in your pipeline shares an index. This is particularly pertinent when writing transformations over (non-event-based) time-series data.

While Apache Hamilton currently has no means of enforcing shared-spine, it is up to the writer of the function to validate input data as necessary. Thus we recommend the following if you are creating a dataframe as output:

Best practice:

1. Load data via functions, defined in their own specific module.

2. Take that loaded data, and transform/ensure indexes match the output you want to create.

3. Continue with transformations.

For time-series modeling, this will mean you provide a common time-series index. Or, if you're creating features for input to a classification model, e.g. over clients, then ensure the index is client_ids.

Output Immutability

In Apache Hamilton, functions are only called once!

Immutability means, that once a "data structure", e.g. a column is created, and output by a function, the values in the column are not changeable.

When Apache Hamilton figures out the execution call path, it walks it and calls functions only once. This means, that if the output of a function is immutable, then there's only one place it was created; it's not modified anywhere else. This provides a great debugging experience if there are ever issues in your dataflow. We believe that by default, one should always strive for immutability of outputs.

However, it is up to you, the Hamilton function writer, to ensure that immutability is something that is adhered to.

Best practice:

1. To preserve "immutability" of outputs, don't mutate passed in data structures. e.g. if you get passed in a pandas series, don't mutate it.
    1. Test for this in your unit tests if this is something important to you!
2. Otherwise YMMV with debugging:
    1. Clearly document mutating inputs in your function documentation if you do mutate inputs provided. That will make debugging your code that much simpler!

Using within your ETL System

Conceptually you can integrate Apache Hamilton within your existing ETL system quite easily:

Compatibility Matrix

Title

Framework / Scheduler	Compatibility
Airflow	✅ (see [airflow example](https://github.com/apache/hamilton/tree/main/examples/airflow))
Dagster	✅
Prefect	✅ (see [prefect example](https://github.com/apache/hamilton/tree/main/examples/prefect))
Kubeflow Pipelines	✅
CRON	✅
dbt	✅(see dbt example)
kubernetes	✅ but you need to setup kubernetes to run an image that can run python code - e.g. see Running a python application on kubernetes
docker	✅ but you need to setup a docker image that can execute python code.
... in general if it runs python 3.7+ ...	✅

ETL Recipe

1. Write Hamilton functions & *"driver"* code.
2. Publish your Hamilton functions in a package, or import via other means (e.g. checkout a repository & include in python path).
3. Include *sf-hamilton* as a python dependency
4. Have your ETL system execute your "driver" code.
5. Profit.

Loading Data

In Apache Hamilton, data loaders are just the same as other functions in the DAG. They take in configuration parameters, and output datasets in the desired form. Following up on the marketing spend dataset, you might write a data loader that reads a dataframe saved in csv format on s3 like this:

```
import boto3
import urllib
import pandas as pd

from hamilton.function_modifiers import extract_columns

client = boto3.client("s3")

@extract_columns('col1', 'col2', 'col3', ...)
def marketing_spend(marketing_spend_data_path: str) -> pd.DataFrame:
    """Loads marketing spend from specified path on s3
    """
    if not marketing_spend_data_path.startswith("s3://"):
        raise ValueError(f"Invalid s3 URI
{marketing_spend_data_path}")
    return pd.read_csv(
        marketing_spend_data_path,
        storage_options = {...}) # See https://pandas.pydata.org/
docs/reference/api/pandas.read_csv.html#pandas-read-csv for more info
```

Loading data is as easy as that! Run your driver with `marketing_spend_data_path` as a parameter, and you're good to go. However, there are a few considerations you might have prior to productionalizing this dataflow...

Plugging in new Data Sources

An advantage of Apache Hamilton is that it allows for rapid plug-and play for various components of your pipeline. This is particularly important for data loading, where you might want to load your data from different sources depending on some context. For instance – if you're running your pipeline in production, you may want to use the production data sources. If you're running it in QA, you might want to use the staging data sources. Or, if you're running it locally, you might want to use abbreviated, in-memory data sources for testing. While Apache Hamilton is not opinionated on exactly _how_ you make this switch, it presents a variety of tooling that can make it more manageable. Some options. To demonstrate some techniques, let's continue on the example of loading marketing spend...

Modules as Interfaces

Say you have multiple data-loading nodes in your DAG. One strategy is to put them all in a single module. That way, if you want to load them up from different sources, you can simply switch the module your driver utilizes. Taking the example from above, you might have the following modules:

```
@extract_columns('col1', 'col2', 'col3', ...)
def marketing_spend(marketing_spend_data_path: str) -> pd.DataFrame:
    """Loads marketing spend from specified path on s3
    """
    if not marketing_spend_data_path.startswith("s3://"):
        raise ValueError(f"Invalid s3 URI
{marketing_spend_data_path}")
    return pd.read_csv(
        marketing_spend_data_path,
        storage_options = {...}) # See https://pandas.pydata.org/
docs/reference/api/pandas.read_csv.html#pandas-read-csv for more info
```

```
@extract_columns('col1', 'col2', 'col3', ...)
def marketing_spend(marketing_spend_data_path: str) -> pd.DataFrame:
    """Loads marketing spend from specified path on s3
    """
    if not marketing_spend_data_path.endswith("csv"):
        raise ValueError(f"Invalid local data loading target
{marketing_spend_data_path}")
    if not os.path.exists(marketing_spend_data_path):
        raise ValueError(f"Path does not exists")
    return pd.read_csv(marketing_spend_data_path)
```

Then, in your driver, you can choose between which module you want to use:

```
local_data_driver = Driver(config, local_data_loaders, ...)
prod_data_driver = Driver(config, prod_data_loaders, ...)
```

Using the Config to Decide Sources

Note that we can utilize the config to determine where the data comes from as well. By using `config.when` you can arrive at the same effect as above, while making it entirely config driven. If you combine the two functions into the same module with *@config.when* it will look as follows:

```
@config.when(data_source='local')
@extract_columns('col1', 'col2', 'col3', ...)
def marketing_spend__local(marketing_spend_data_path: str) ->
pd.DataFrame:
    ...

@config.when(data_source='prod')
@extract_columns('col1', 'col2', 'col3', ...)
def marketing_spend__prod(marketing_spend_data_path: str) ->
pd.DataFrame:
    ...
```

Then you can invoke your driver but set the config differently:

```
driver = Driver(
    {'data_source' : 'prod', 'marketing_spend_data_path' :
's3://...'},
    data_loaders, ...)
```

Note that there are a variety of other ways you can organize your code – at this point its entirely use-case dependent. Apache Hamilton is a language for declaring dataflows that's applicable towards a multitude of use-cases. It's not going to dictate how to write your functions or where you put them.

User Guide

This portion of the documentation goes over the set of common examples for Apache Hamilton usage, so you can apply it to your day-to-day work. Each one corresponds to an example in the examples directory. If there's an example you want but don't see, reach out or open an issue on github – we're always looking to add more.

Jupyter notebooks

There are two main ways to use Apache Hamilton in a notebook.

1. Dynamically create modules within the notebook.

2. Import modules into the notebook.

1 - Dynamically create modules within your notebook

There's two main ways, using the Hamilton Jupyter magic, or using `ad_hoc_utils` to create a temporary module.

Use Hamilton Jupyter Magic

The Hamilton Jupyter magic allows you to dynamically create a module from a cell in your notebook. This is useful for quick iteration and development. Once you're then happy, it's easy to then write out a module with the functions you've developed using `%%writefile` magic.

To load the magic:

```
# load some extensions / magic...
%load_ext hamilton.plugins.jupyter_magic
```

Then to use it:

```
%%cell_to_module -m MODULE_NAME # more args
```

To see help on the magic, you can run `%%cell_to_module --help`, or just `?%%cell_to_module` in a cell.

It should output information similar to the following:

-m, –module_name: Module name to provide. Default is jupyter_module. -c, –config: JSON config string, or variable name containing config to use. -r, –rebuild-drivers: Flag to rebuild drivers. -d, –display: Flag to visualize dataflow. -v, –verbosity: of standard output. 0 to hide. 1 is normal, default.

Example use:

```
%%cell_to_module -m MODULE_NAME --display --rebuild-drivers

def hello() -> str:
    return "hello"

def world(hello: str) -> str:
    return f"{hello} world"
```

Once you're happy with the functions you've developed, you can then write them out to a module using the `%%writefile` magic:

```
%%writefile hello_world.py
```

Importing specific functions into cell modules

If you import parts of modules in a Hamilton Jupyter Magic cell, these will need to be reloaded when changes are made to their source. This can be done either by restarting the kernel or with the help of importlib.reload:

```
%%cell_to_module MODULE_NAME

# first import the module itself, so it can be reloaded
import my_common_functions

# reload the module
import importlib
importlib.reload(my_common_functions)
# now import the specific function from the module
from my_common_functions import commonfunction

# use the imported function
commonfunction()
```

Using ad_hoc_utils to create a temporary module (e.g. use in google colab)

You have the ability to inline define functions with your driver that can be used to build a DAG. *We strongly recommend only using this approach when absolutely necessary* — it's very easy to build spaghetti code this way.

For example, say we want to add a function to compute the logarithm of `avg_3wk_spend` and not add it to `some_functions.py`, we can do the following steps directly in our notebook:

```
# Step 1 - define function
import numpy as np

def log_avg_3wk_spend(avg_3wk_spend: pd.Series) -> pd.Series:
    """Simple function taking the logarithm of spend over signups."""
    return np.log(avg_3wk_spend)
```

We then have to create a "temporary python module" to house it in. We do this by importing `ad_hoc_utils` and then calling the `create_temporary_module` function, passing in the functions we want, and providing a name for the module we're creating.

```
# Step 2 - create a temporary modeul to house all notebook functions
from hamilton import ad_hoc_utils
temp_module = ad_hoc_utils.create_temporary_module(
     log_avg_3wk_spend, module_name='function_example')
```

You can now treat `temp_module` like a python module and pass it to your driver and use Hamilton like normal:

```
# Step 3 - add the module to the driver and continue as usual
dr = driver.Driver(config, some_functions, temp_module)
df = dr.execute(['avg_3wk_spend', 'log_avg_3wk_spend'],
inputs=input_data)
```

Caveat with this approach:

Using a "temporary python module" will not enable scaling of computation by using Ray, Dask, or Pandas on Spark. So we suggest only using this approach for development purposes only.

2 - Importing modules into your notebook

This tutorial can also be found published on TDS.

Step 1 — Install Jupyter & Apache Hamilton

I assume you already have this step set up. But just in case you don't:

```
pip install jupyterlab
pip install sf-hamilton
```

Then to start the notebook server it should just be:

Step 2— Set up the files

1. Start up your Jupyter notebook.

2. Go to the directory where you want your notebook and Hamilton function module(s) to live.

3. Create a python file(s). Do that by going to "New > text file". It'll open a "file" editor view. Name the file and give it a `.py` extension. Once you save it, you'll see that jupyter now provides python syntax highlighting. Keep this tab open, so you can flip back to it to edit this file.

4. Start up a notebook that you will use in another browser tab.

Step 3— The basic process of iteration

At a high level, you will be switching back and forth between your tabs. You will add functions to your Hamilton function python module, and then import/reimport that module into your notebook to get the changes. From there you will then use Apache Hamilton as usual to run and execute things and the notebook for all the standard things you use notebooks for.

Let's walk through an example.

Here's a function I added to our Hamilton function module. I named the module `some_functions.py` (obviously choose a better name for your situation).

```
import pandas as pd

def avg_3wk_spend(spend: pd.Series) -> pd.Series:
    """Rolling 3 week average spend."""
    print("foo") # will use this to prove it reloaded!
    return spend.rolling(3).mean()
```

And here's what I set up in my notebook to be able to use Hamilton and import this module:

Cell 1: This just imports the base things we need; see the pro-tip at the bottom of this page for how to automatically reload changes.

```
import importlib
import pandas as pd
from hamilton import driver
```

Cell 2: Import your Hamilton function module(s)

```
# import your hamilton function module(s) here
import some_functions
```

Cell 3: Run this cell anytime you make and save changes to `some_functions.py`

```
# use this to reload the module after making changes to it.
importlib.reload(some_functions)
```

What this will do is reload the module, and therefore make sure the code is up to date for you to use.

Cell 4: Use Hamilton

```
config = {}
dr = driver.Driver(config, some_functions)
input_data = {'spend': pd.Series([0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10])}
df = dr.execute(['avg_3wk_spend'], inputs=input_data)
```

You should see `foo` printed as an output after running this cell.

Okay, so let's now say we're iterating on our Hamilton functions. Go to your Hamilton function module (`some_functions.py` in this example) in your other browser tab, and change the `print("foo")` to something else, e.g. `print("foo-bar").` Save the file — it should look something like this:

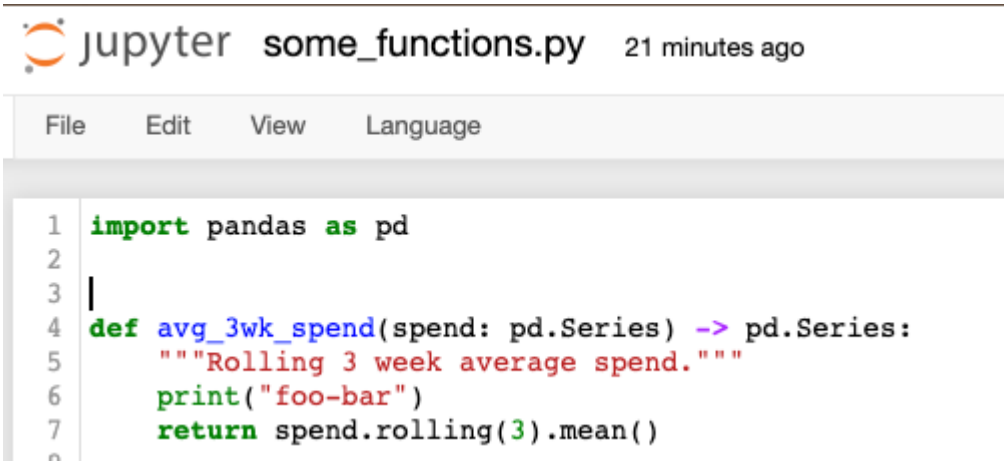
```
def avg_3wk_spend(spend: pd.Series) -> pd.Series:
    """Rolling 3 week average spend."""
    print("foo-bar")
    return spend.rolling(3).mean()
```

Go back to your notebook, and re-run Cell 3 & Cell 4. You should now see a different output printed, e.g. `foo-bar` .

Congratulations! You just managed to iterate on Apache Hamilton using a Jupyter notebook!

To summarize this is how things ended up looking on my end:

- Here's what my `some_functions.py` file looks like:



- Here's what my notebook looks like:

```
In [ ]: import importlib

        import pandas as pd

        from hamilton import driver
```

```
In [ ]: # import your hamilton function module(s) here
        import some_functions
```

```
In [ ]: # use this to reload the module after making changes to it.
        importlib.reload(some_functions)
```

```
In [ ]: # use hamilton -- re-run this if you changed the functions
        config = {}
        dr = driver.Driver(config, some_functions)
        input_data = {'spend': pd.Series([0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10])}
        df = dr.execute(['avg_3wk_spend'], inputs=input_data)
```

```
In [ ]: print(df)
```

```
In [ ]: # carry on with your notebook
```

Pro-tip: You can use ipython magic to autoreload code

Open a Python module and a Jupyter notebook side-to-side, and then add %autoreload ipython magic to the notebook to auto-reload the cell:

```
from hamilton.driver import Driver

# load extension
%load_ext autoreload
# configure autoreload to only affect specified files
```

```
%autoreload 1
# import & specify my_module to be reloaded
# i.e. this is the data transformation module that I have open in
other tab
%aimport my_module

hamilton_driver = Driver({}, my_module)
hamilton_driver.execute(['desired_output1', 'desired_output2'])
```

You'd then follow the following process:

1. Write your data transformation in the open python module

2. In the notebook, instantiate a Hamilton Driver and test the DAG with a small subset of data.

3. Because of %autoreload, the module is reimported with the latest changes each time the Hamilton DAG is executed. This approach prevents out-of-order notebook executions, and functions always reside in clean .py files.

Credit: Thierry Jean's blog post.

Pro-tip: You can import functions directly

The nice thing about forcing Hamilton functions into a module, is that it's very easy to re-use in another context. E.g. another notebook, or directly.

For example, it is easy to directly use the functions in the notebook, like so:

```
some_functions.avg_3wk_spend(pd.Series([0, 1, 2, 3, 4, 5, 6, 7, 8,
9, 10]))
```

Which calls the `avg_3wk_spend` function we defined in the `some_functions.py` module.

Loading data

While we've been injecting data in from the driver in previous examples, Hamilton functions are fully capable of loading their own data. In the following example, we'll show how to use Apache Hamilton to:

1. Load data from an external source (CSV file and duckdb database)

2. Alter the source of data depending on how the DAG is parameterized/created

3. Mock data for a test-setting (so you can quickly execute your DAG without having to wait for data to load)

See the full tutorial here.

Caching

In Hamilton, **caching** broadly refers to "reusing results from previous executions to skip redundant computation". If you change code or pass new data, it will automatically determine which results can be reused and which nodes need to be re-executed. This improves execution speed and reduces resource usage (computation, API credits, etc.).

Note

Open the notebook in Google Colab for an interactive version and better syntax highlighting.

Throughout this tutorial, we'll be using the Hamilton notebook extension to define dataflows directly in the notebook (see tutorial).

```
from hamilton import driver

# load the notebook extension
%reload_ext hamilton.plugins.jupyter_magic
```

We import the `logging` module and get the logger from `hamilton.caching`. With the level set to `INFO`, we'll see `GET_RESULT` and `EXECUTE_NODE` cache events as they happen.

```
import logging
```

```
logger = logging.getLogger("hamilton.caching")
logger.setLevel(logging.INFO)
logger.addHandler(logging.StreamHandler())
```

The next cell deletes the cached data to ensure this notebook can be run from top to bottom without any issues.

```
import shutil

shutil.rmtree("./.hamilton_cache", ignore_errors=True)
```

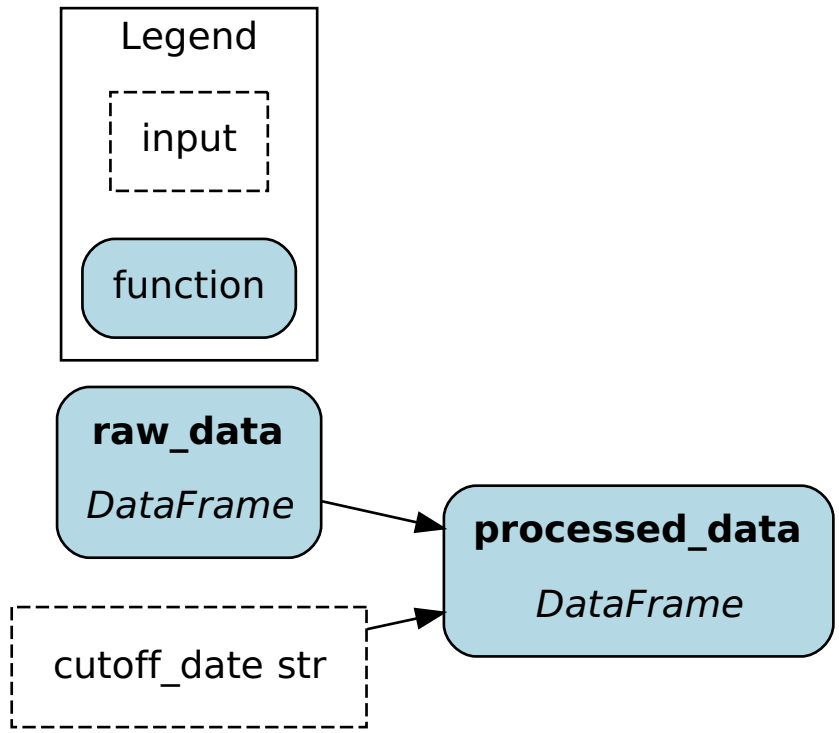
Basics

Throughout this notebook, we'll use the same simple dataflow that processes transactions in various locations and currencies.

We use the cell magic `%%cell_to_module` from the Hamilton notebook extension. It will convert the content of the cell into a Python module that can be loaded by Hamilton. The `--display` flag allows to visualize the dataflow.

```
%%cell_to_module basics_module --display
import pandas as pd

DATA = {
    "cities": ["New York", "Los Angeles", "Chicago", "Montréal",
"Vancouver"],
    "date": ["2024-09-13", "2024-09-12", "2024-09-11", "2024-09-11",
"2024-09-09"],
    "amount": [478.23, 251.67, 989.34, 742.14, 584.56],
    "country": ["USA", "USA", "USA", "Canada", "Canada"],
    "currency": ["USD", "USD", "USD", "CAD", "CAD"],
}

def raw_data() -> pd.DataFrame:
    """Loading raw data. This simulates loading from a file,
database, or external service."""
    return pd.DataFrame(DATA)

def processed_data(raw_data: pd.DataFrame, cutoff_date: str) ->
pd.DataFrame:
    """Filter out rows before cutoff date and convert currency to
USD."""
    df = raw_data.loc[raw_data.date > cutoff_date].copy()
    df["amound_in_usd"] = df["amount"]
```

```
    df.loc[df.country == "Canada", "amound_in_usd"] *= 0.73
    return df
```



Then, we build the `Driver` with caching enabled and execute the dataflow.

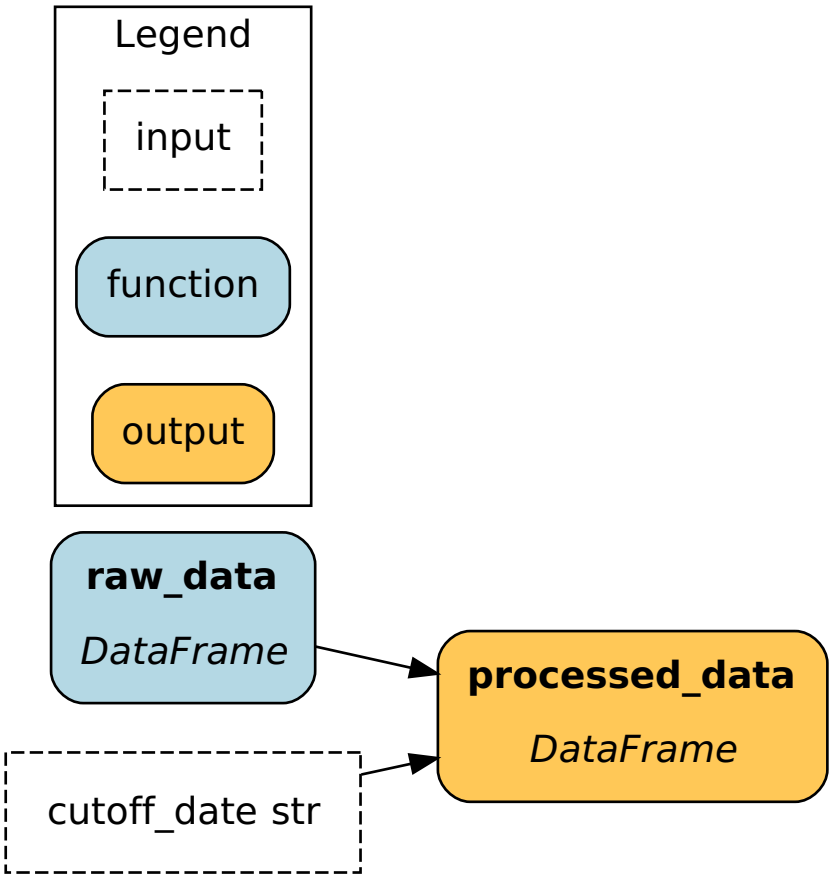
```
basics_dr =
driver.Builder().with_modules(basics_module).with_cache().build()

basics_results_1 = basics_dr.execute(["processed_data"],
inputs={"cutoff_date": "2024-09-01"})
print()
print(basics_results_1["processed_data"].head())
```

```
raw_data::adapter::execute_node
processed_data::adapter::execute_node
```

```
        cities        date  amount country currency  amound_in_usd
0     New York  2024-09-13  478.23     USA      USD       478.2300
1  Los Angeles  2024-09-12  251.67     USA      USD       251.6700
2      Chicago  2024-09-11  989.34     USA      USD       989.3400
3     Montréal  2024-09-11  742.14  Canada      CAD       541.7622
4    Vancouver  2024-09-09  584.56  Canada      CAD       426.7288
```

We can view what values were retrieved from the cache using `dr.cache.view_run()`. Since this was the first execution, nothing is retrieved.

```
basics_dr.cache.view_run()
```

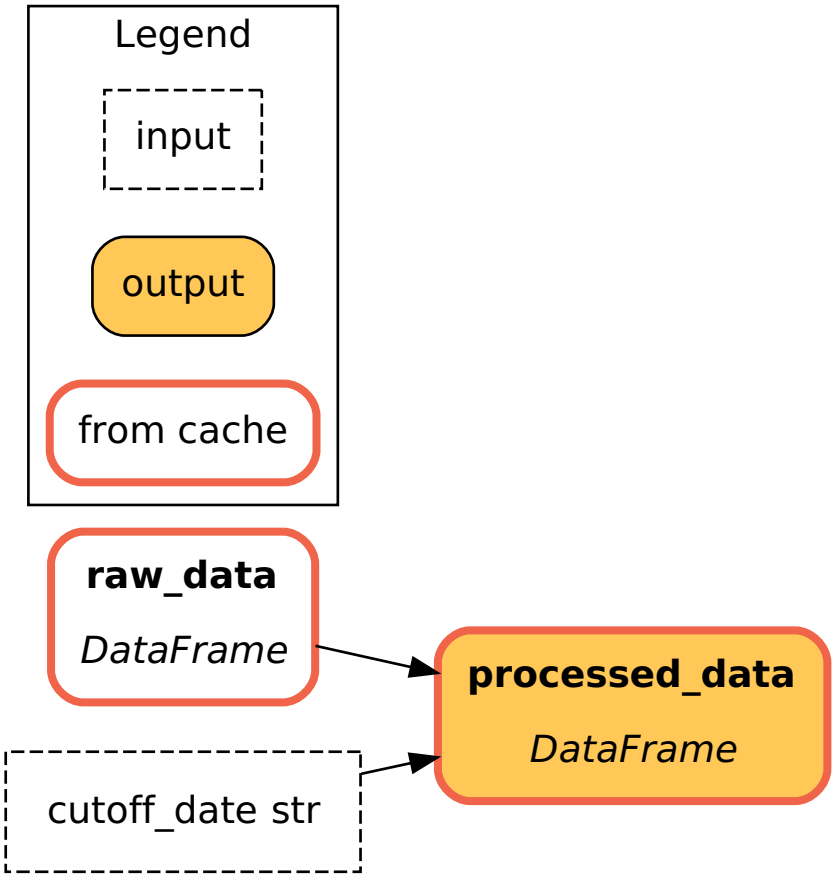


On the second execution, `processed_data` is retrieved from cache as reported in the logs and highlighted in the visualization

```
basics_results_2 = basics_dr.execute(["processed_data"],
inputs={"cutoff_date": "2024-09-01"})
print()
print(basics_results_2["processed_data"].head())
print()
basics_dr.cache.view_run()
```

```
raw_data::result_store::get_result::hit
processed_data::result_store::get_result::hit
```

```
        cities        date  amount country currency  amound_in_usd
0     New York  2024-09-13  478.23     USA      USD       478.2300
1  Los Angeles  2024-09-12  251.67     USA      USD       251.6700
2      Chicago  2024-09-11  989.34     USA      USD       989.3400
3     Montréal  2024-09-11  742.14  Canada      CAD       541.7622
4    Vancouver  2024-09-09  584.56  Canada      CAD       426.7288
```



Understanding the `cache_key`

The Hamilton cache stores results using a `cache_key`. It is composed of the node's name (`node_name`), the code that defines it (`code_version`), and its data inputs (`data_version` of its dependencies).

For example, the cache keys for the previous cells are:

```
{
    "node_name": "raw_data",
    "code_version":
"9d727859b9fd883247c3379d4d25a35af4a56df9d9fde20c75c6375dde631c68",
    "dependencies_data_versions": {}  // it has no dependencies
}
{
    "node_name": "processed_data",
    "code_version":
"c9e3377d6c5044944bd89eeb7073c730ee8707627c39906b4156c6411f056f00",
    "dependencies_data_versions": {
        "cutoff_date": "WkGjJythLWYAIj2Qr8T_ug==",  // input value
        "raw_data": "t-BDcMLikFSNdn4piUKy1mBcKPoEsnsYjUNzWg=="  //
raw_data's result
```

```
    }
}
```

Results could be successfully retrieved because nodes in the first execution and second execution shared the same `cache_key`.

The `cache_key` objects are internal and you won't have to interact with them directly. However, keep that concept in mind throughout this tutorial. Towards the end, we show how to manually handle the `cache_key` for debugging.

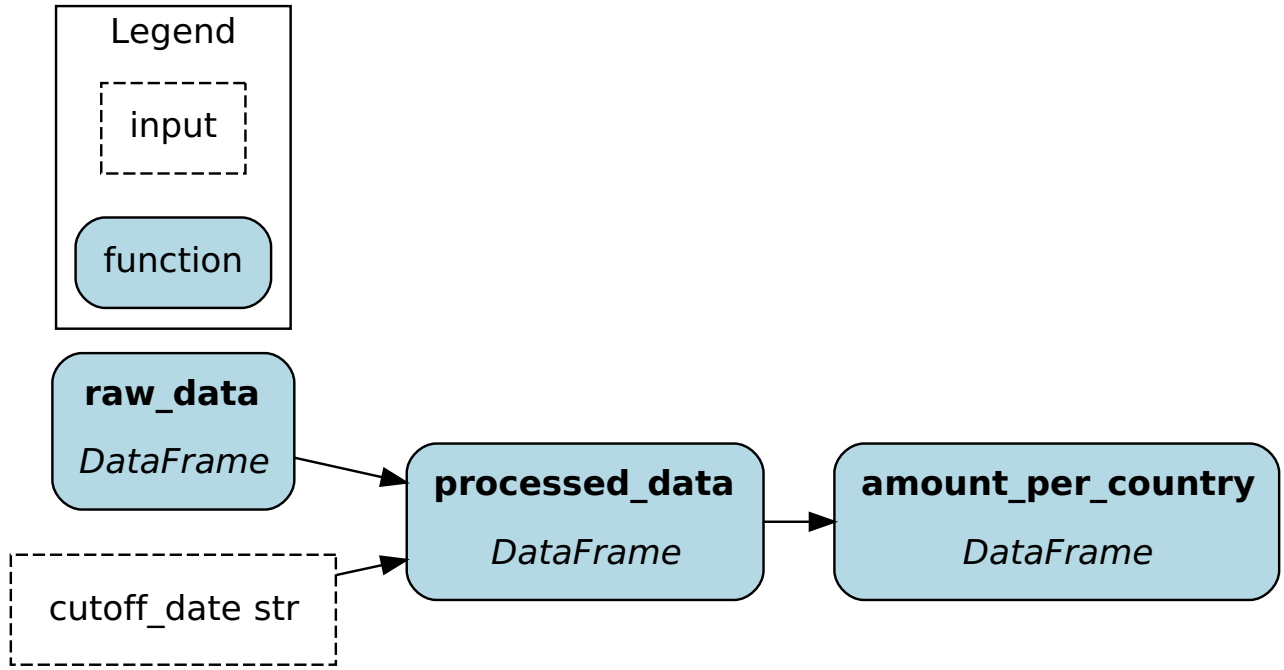
Adding a node

Let's say you're iteratively developing your dataflow and you add a new node. Here, we copy the previous module into a new module named `adding_node_module` and define the node `amount_per_country`.

> In practice, you would edit the cell directly, but this makes the notebook easier to read and maintain

```
%%cell_to_module adding_node_module --display
import pandas as pd

DATA = {
    "cities": ["New York", "Los Angeles", "Chicago", "Montréal",
"Vancouver"],
    "date": ["2024-09-13", "2024-09-12", "2024-09-11", "2024-09-11",
"2024-09-09"],
    "amount": [478.23, 251.67, 989.34, 742.14, 584.56],
    "country": ["USA", "USA", "USA", "Canada", "Canada"],
    "currency": ["USD", "USD", "USD", "CAD", "CAD"],
}

def raw_data() -> pd.DataFrame:
    """Loading raw data. This simulates loading from a file,
database, or external service."""
    return pd.DataFrame(DATA)

def processed_data(raw_data: pd.DataFrame, cutoff_date: str) ->
pd.DataFrame:
    """Filter out rows before cutoff date and convert currency to
USD."""
    df = raw_data.loc[raw_data.date > cutoff_date].copy()
    df["amound_in_usd"] = df["amount"]
    df.loc[df.country == "Canada", "amound_in_usd"] *= 0.73
    return df

def amount_per_country(processed_data: pd.DataFrame) -> pd.DataFrame:
```

```
    """Sum the amount in USD per country"""
    return processed_data.groupby("country")
["amound_in_usd"].sum().to_frame()
```

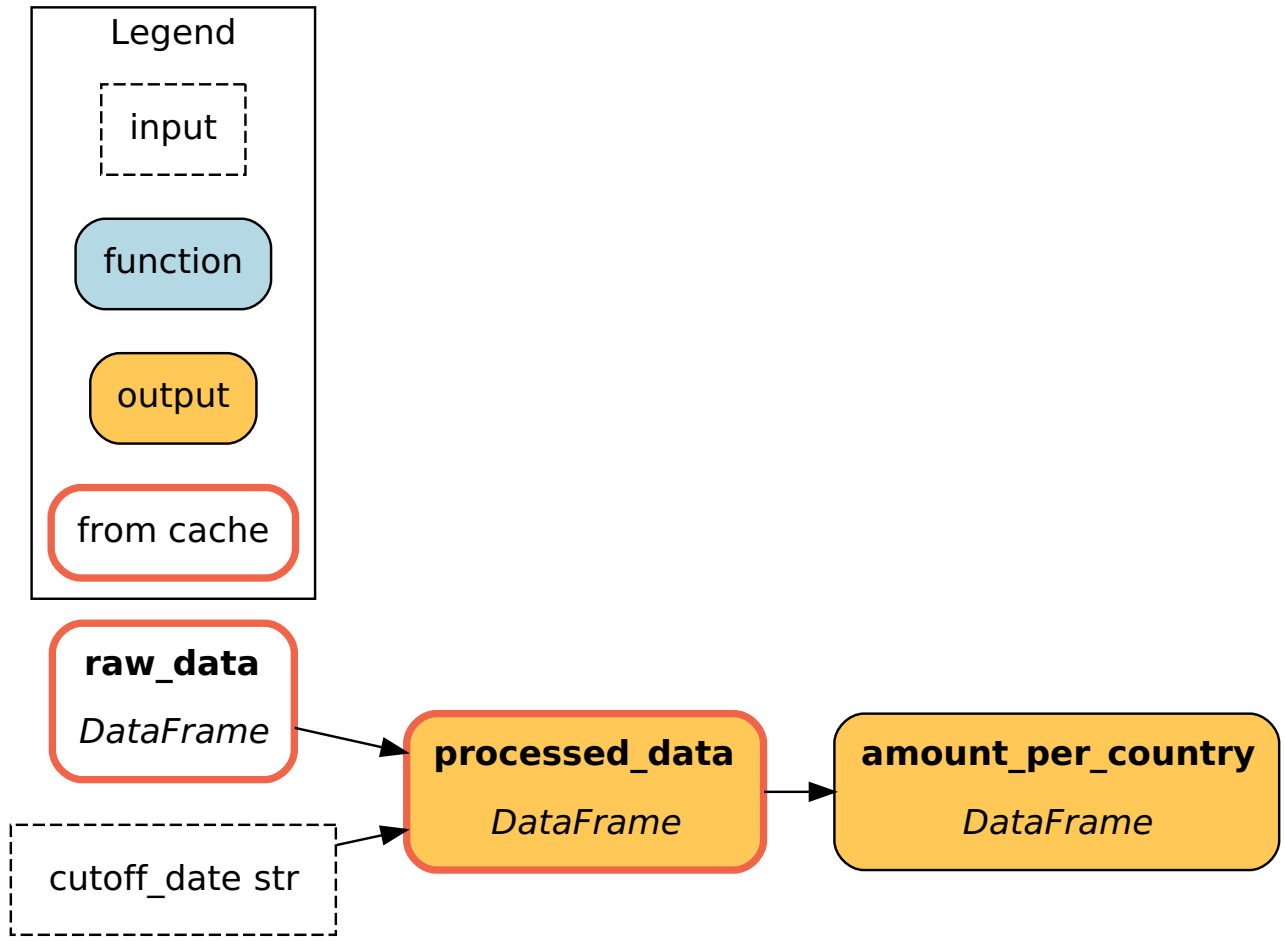


We build a new `Driver` with `adding_node_module` and execute the dataflow. You'll notice that `raw_data` and `processed_data` are retrieved and only `amount_per_country` is executed.

```
adding_node_dr =
driver.Builder().with_modules(adding_node_module).with_cache().build()

adding_node_results = adding_node_dr.execute(
    ["processed_data", "amount_per_country"],
    inputs={"cutoff_date": "2024-09-01"}
)
print()
print(adding_node_results["amount_per_country"].head())
print()
adding_node_dr.cache.view_run()
```

```
raw_data::result_store::get_result::hit
processed_data::result_store::get_result::hit
amount_per_country::adapter::execute_node
```

```
         amound_in_usd
country
Canada         968.491
USA           1719.240
```



Even though this is the first execution of `adding_node_dr` and the module `adding_node_module`, the cache contains results for `raw_data` and `processed_data`. We're able to retrieve values because they have the same cache keys (code version and dependencies data versions).

This means you can reuse cached results across dataflows. This is particularly useful with training and inference machine learning pipelines.

Changing inputs

We reuse the same dataflow `adding_node_module`, but change the input `cutoff_date` from `"2024-09-01"` to `"2024-09-11"`.

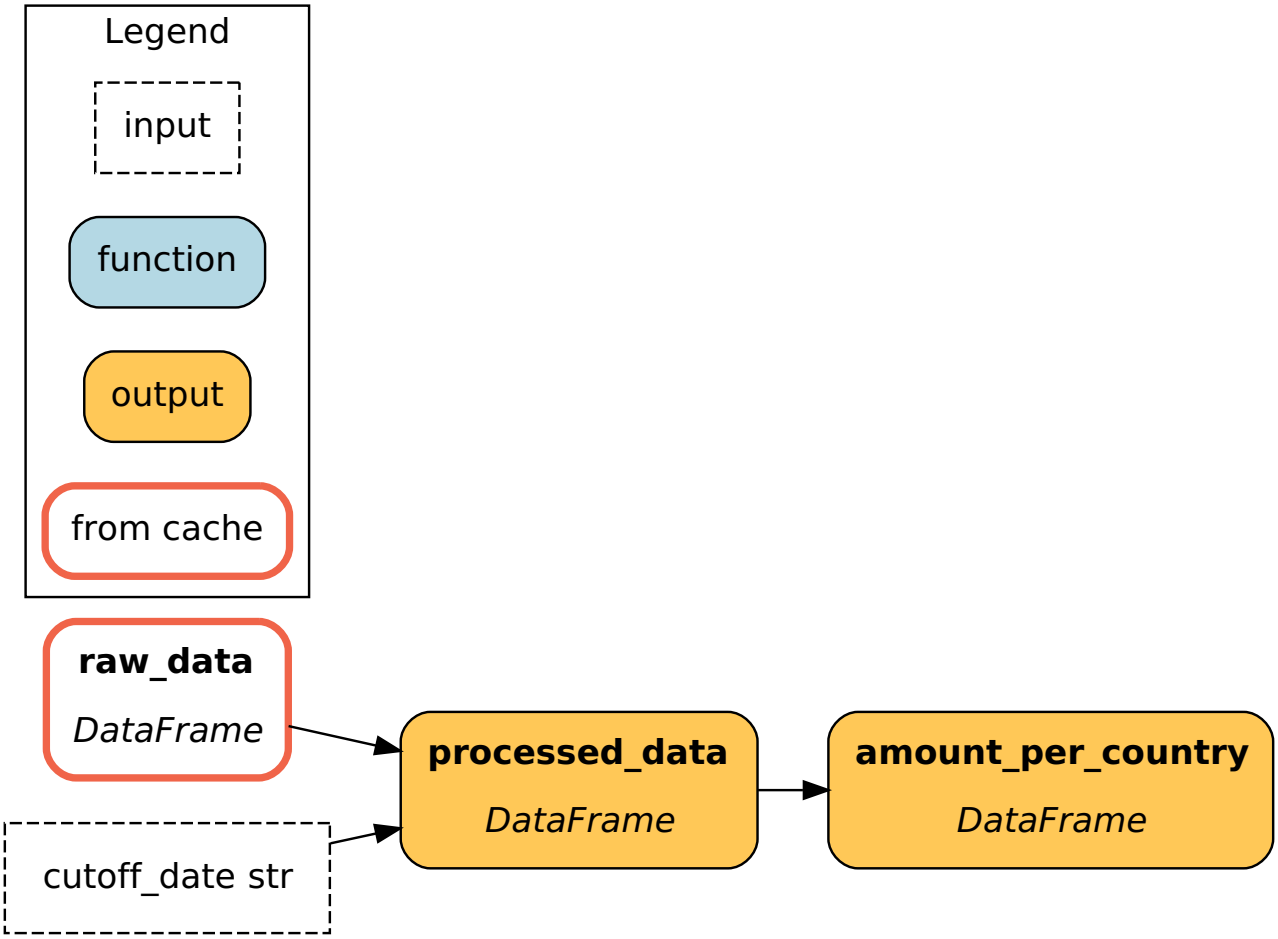
This new input forces `processed_data` to be re-executed. This produces a new result for `processed_data`, which cascades and also forced `amount_per_country` to be re-executed.

```
changing_inputs_dr =
driver.Builder().with_modules(adding_node_module).with_cache().build()

changing_inputs_results_1 = changing_inputs_dr.execute(
    ["processed_data", "amount_per_country"],
    inputs={"cutoff_date": "2024-09-11"}
```

```
)
print()
print(changing_inputs_results_1["amount_per_country"].head())
print()
changing_inputs_dr.cache.view_run()
```

```
raw_data::result_store::get_result::hit
processed_data::adapter::execute_node
amount_per_country::adapter::execute_node
```

```
         amound_in_usd
country
USA              729.9
```



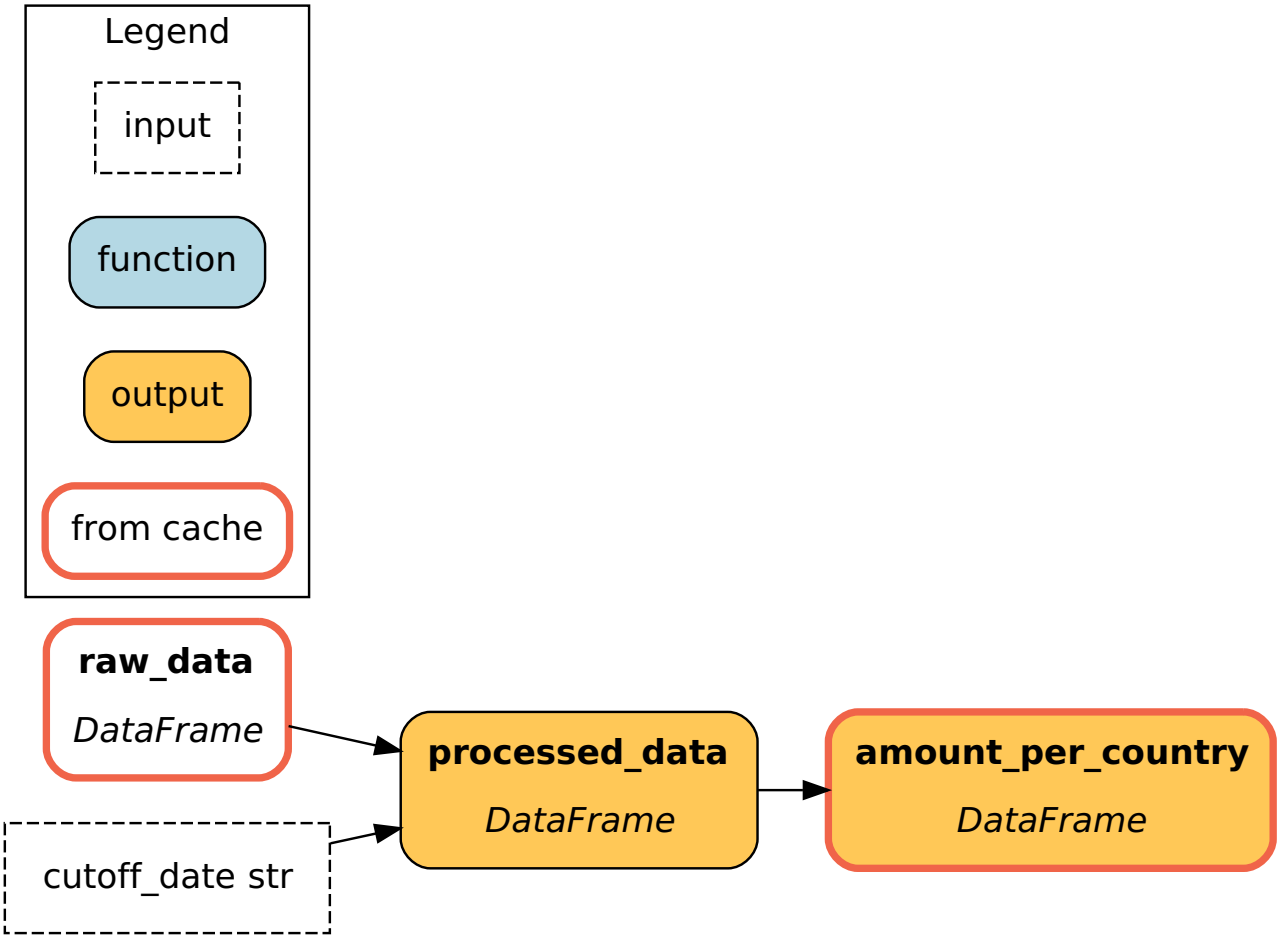
Now, we execute with the `cutoff_date` value `"2024-09-05"`, which forces `processed_data` to be executed.

```
changing_inputs_results_2 = changing_inputs_dr.execute(
    ["processed_data", "amount_per_country"],
    inputs={"cutoff_date": "2024-09-05"}
)
```

```
print()
print(changing_inputs_results_2["amount_per_country"].head())
print()
changing_inputs_dr.cache.view_run()
```

```
raw_data::result_store::get_result::hit
processed_data::adapter::execute_node
amount_per_country::result_store::get_result::hit
```

```
         amound_in_usd
country
Canada         968.491
USA           1719.240
```



Notice that the cache could still retrieve `amount_per_country`. This is because `processed_data` return a value that had been cached previously (in the `Adding a node` section).

In concrete terms, filtering rows by the date `"2024-09-05"` or `"2024-09-01"` includes the same rows and produces the same dataframe.

```
print(adding_node_results["processed_data"])
print()
print(changing_inputs_results_2["processed_data"])
```

```
        cities        date  amount country currency  amound_in_usd
0     New York  2024-09-13  478.23     USA      USD       478.2300
1  Los Angeles  2024-09-12  251.67     USA      USD       251.6700
2      Chicago  2024-09-11  989.34     USA      USD       989.3400
3     Montréal  2024-09-11  742.14  Canada      CAD       541.7622
4    Vancouver  2024-09-09  584.56  Canada      CAD       426.7288

        cities        date  amount country currency  amound_in_usd
0     New York  2024-09-13  478.23     USA      USD       478.2300
1  Los Angeles  2024-09-12  251.67     USA      USD       251.6700
2      Chicago  2024-09-11  989.34     USA      USD       989.3400
3     Montréal  2024-09-11  742.14  Canada      CAD       541.7622
4    Vancouver  2024-09-09  584.56  Canada      CAD       426.7288
```

Changing code

As you develop your dataflow, you will need to edit upstream nodes. Caching will automatically detect code changes and determine which node needs to be re-executed. In `processed_data()`, we'll change the conversation rate from `0.73` to `0.71`.

> NOTE. changes to docstrings and comments `#` are ignored when versioning a node.

```
%%cell_to_module changing_code_module
import pandas as pd

DATA = {
    "cities": ["New York", "Los Angeles", "Chicago", "Montréal",
"Vancouver"],
    "date": ["2024-09-13", "2024-09-12", "2024-09-11", "2024-09-11",
"2024-09-09"],
    "amount": [478.23, 251.67, 989.34, 742.14, 584.56],
    "country": ["USA", "USA", "USA", "Canada", "Canada"],
    "currency": ["USD", "USD", "USD", "CAD", "CAD"],
}

def raw_data() -> pd.DataFrame:
    """Loading raw data. This simulates loading from a file,
database, or external service."""
    return pd.DataFrame(DATA)

def processed_data(raw_data: pd.DataFrame, cutoff_date: str) ->
pd.DataFrame:
```

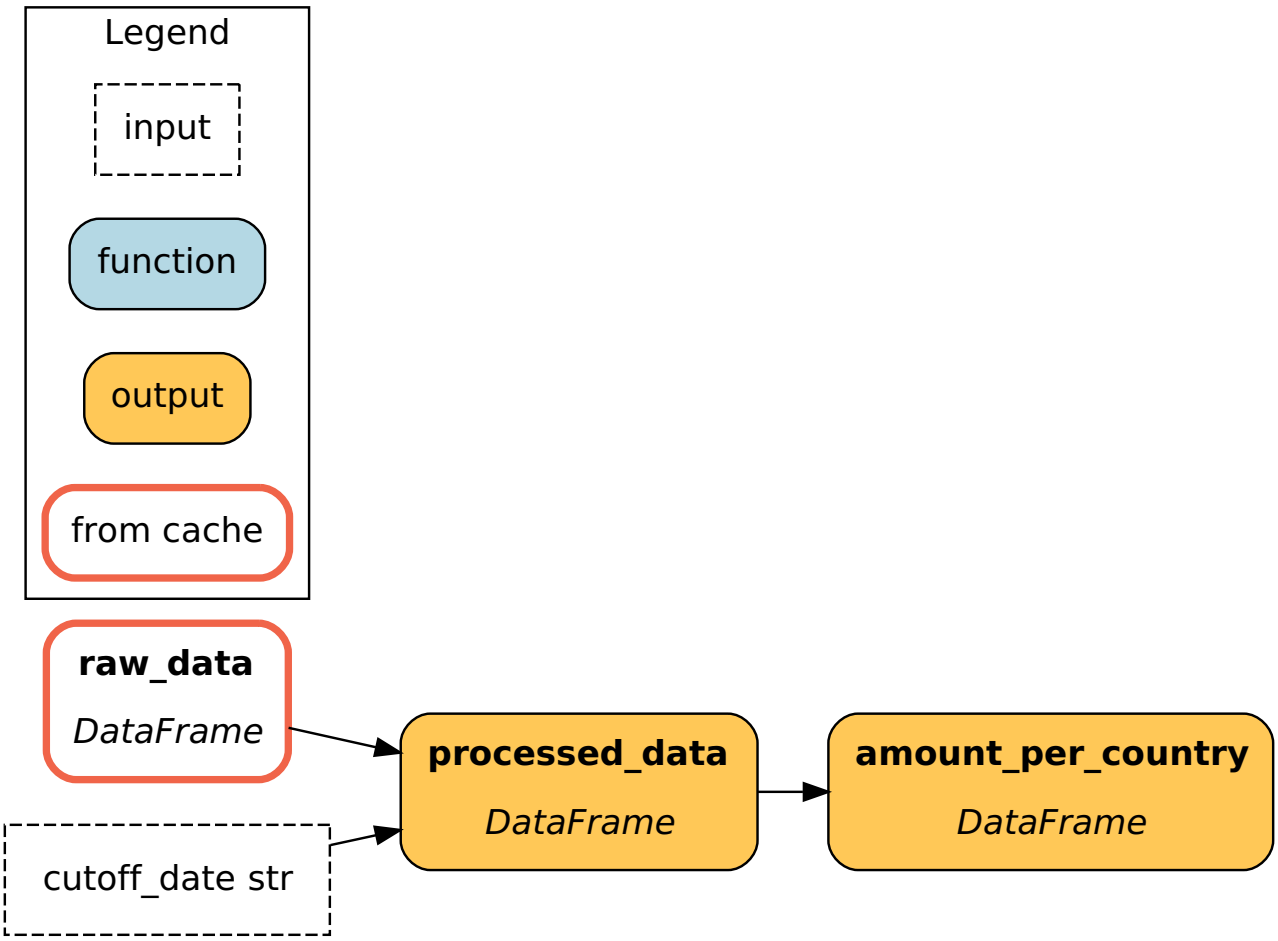
```
    """Filter out rows before cutoff date and convert currency to
USD."""
    df = raw_data.loc[raw_data.date > cutoff_date].copy()
    df["amound_in_usd"] = df["amount"]
    df.loc[df.country == "Canada", "amound_in_usd"] *= 0.71  # <-
VALUE CHANGED FROM module_2
    return df

def amount_per_country(processed_data: pd.DataFrame) -> pd.DataFrame:
    """Sum the amount in USD per country"""
    return processed_data.groupby("country")
["amound_in_usd"].sum().to_frame()
```

We need to execute `processed_data` because the code change created a new `cache_key` and led to a cache miss. Then, `processed_data` returns a previously unseen value, forcing `amount_per_country` to also be re-executed

```
changing_code_dr_1 =
driver.Builder().with_modules(changing_code_module).with_cache().build()

changing_code_results_1 = changing_code_dr_1.execute(
    ["processed_data", "amount_per_country"],
    inputs={"cutoff_date": "2024-09-01"}
)
print()
print(changing_code_results_1["amount_per_country"].head())
print()
changing_code_dr_1.cache.view_run()
```

```
raw_data::result_store::get_result::hit
processed_data::adapter::execute_node
amount_per_country::adapter::execute_node
```

```
         amound_in_usd
country
Canada         941.957
USA           1719.240
```



We make another code change to `processed_data` to accomodate currency conversion for Brazil and Mexico.

```
%%cell_to_module changing_code_module_2
import pandas as pd

DATA = {
    "cities": ["New York", "Los Angeles", "Chicago", "Montréal",
"Vancouver"],
    "date": ["2024-09-13", "2024-09-12", "2024-09-11", "2024-09-11",
"2024-09-09"],
    "amount": [478.23, 251.67, 989.34, 742.14, 584.56],
    "country": ["USA", "USA", "USA", "Canada", "Canada"],
    "currency": ["USD", "USD", "USD", "CAD", "CAD"],
}

def raw_data() -> pd.DataFrame:
    """Loading raw data. This simulates loading from a file,
database, or external service."""
    return pd.DataFrame(DATA)

def processed_data(raw_data: pd.DataFrame, cutoff_date: str) ->
pd.DataFrame:
    """Filter out rows before cutoff date and convert currency to
```

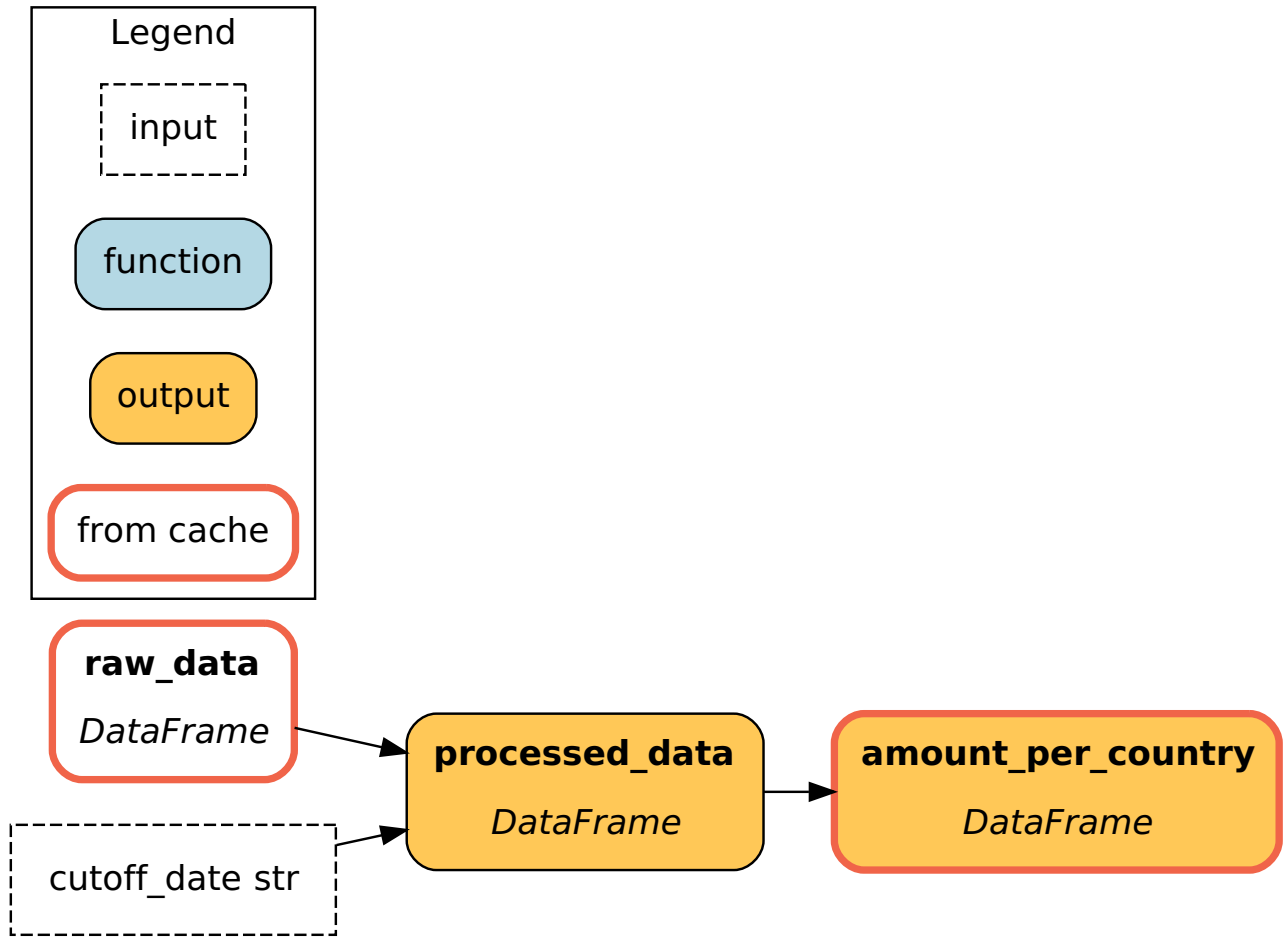
```
USD."""
    df = raw_data.loc[raw_data.date > cutoff_date].copy()
    df["amound_in_usd"] = df["amount"]
    df.loc[df.country == "Canada", "amound_in_usd"] *= 0.71
    df.loc[df.country == "Brazil", "amound_in_usd"] *= 0.18  # <-
LINE ADDED
    df.loc[df.country == "Mexico", "amound_in_usd"] *= 0.05  # <-
LINE ADDED
    return df

def amount_per_country(processed_data: pd.DataFrame) -> pd.DataFrame:
    """Sum the amount in USD per country"""
    return processed_data.groupby("country")
["amound_in_usd"].sum().to_frame()
```

Again, the code change forces `processed_data` to be executed.

```
changing_code_dr_2 =
driver.Builder().with_modules(changing_code_module_2).with_cache().build()

changing_code_results_2 =
changing_code_dr_2.execute(["processed_data","amount_per_country"],
inputs={"cutoff_date": "2024-09-01"})
print()
print(changing_code_results_2["amount_per_country"].head())
print()
changing_code_dr_2.cache.view_run()
```

```
raw_data::result_store::get_result::hit
processed_data::adapter::execute_node
amount_per_country::result_store::get_result::hit
```

```
         amound_in_usd
country
Canada         941.957
USA           1719.240
```



However, `amount_per_country` can be retrieved because `processed_data` returned a previously seen value.

In concrete terms, adding code to process currency from Brazil and Mexico didn't change the `processed_data` result because it only includes data from the USA and Canada.

NOTE. This is similar to what happened at the end of the section **Changing inputs**.

```
print(changing_code_results_1["processed_data"])
print()
print(changing_code_results_2["processed_data"])
```

```
        cities        date  amount country currency  amound_in_usd
0     New York  2024-09-13  478.23     USA      USD       478.2300
1  Los Angeles  2024-09-12  251.67     USA      USD       251.6700
2      Chicago  2024-09-11  989.34     USA      USD       989.3400
3     Montréal  2024-09-11  742.14  Canada      CAD       526.9194
4    Vancouver  2024-09-09  584.56  Canada      CAD       415.0376

        cities        date  amount country currency  amound_in_usd
0     New York  2024-09-13  478.23     USA      USD       478.2300
1  Los Angeles  2024-09-12  251.67     USA      USD       251.6700
```

```
2      Chicago  2024-09-11  989.34     USA      USD       989.3400
3     Montréal  2024-09-11  742.14  Canada      CAD       526.9194
4    Vancouver  2024-09-09  584.56  Canada      CAD       415.0376
```

Changing external data

Hamilton's caching mechanism uses the node's `code_version` and its dependencies `data_version` to determine if the node needs to be executed or the result can be retrieved from cache. By default, it assumes idempotency of operations.

This section covers how to handle node with external effects, such as reading or writing external data.

Idempotency

To illustrate idempotency, let's use this minimal dataflow which has a single node that returns the current date and time:

```python
import datetime

def current_datetime() -> datetime.datetime:
    return datetime.datetime.now()
```

The first execution will execute the node and store the resulting date and time. On the second execution, the cache will read the stored result instead of re-executing. Why? Because the `code_version` is the same and the dependencies `data_version` (it has no dependencies) haven't changed.

A similar situation occurs when reading from external data, as shown here:

```python
import pandas as pd

def dataset(file_path: str) -> pd.DataFrame:
    return pd.read_csv(file_path)
```

Here, the code of `dataset()` and the value for `file_path` can stay the same, but the file itself could be updated (e.g., new rows added).

The next sections show how to always re-execute a node and ensure the latest data is used. The `DATA` constant is modified with transactions in Brazil and Mexico to simulate `raw_data` loading a new dataset.

```
%%cell_to_module changing_external_module
import pandas as pd

DATA = {
    "cities": ["New York", "Los Angeles", "Chicago", "Montréal",
"Vancouver", "Houston", "Phoenix", "Mexico City", "Chihuahua City",
"Rio de Janeiro"],
    "date": ["2024-09-13", "2024-09-12", "2024-09-11", "2024-09-11",
"2024-09-09", "2024-09-08", "2024-09-07", "2024-09-06", "2024-09-05",
"2024-09-04"],
    "amount": [478.23, 251.67, 989.34, 742.14, 584.56, 321.85,
918.67, 135.22, 789.12, 432.78],
    "country": ["USA", "USA", "USA", "Canada", "Canada", "USA",
"USA", "Mexico", "Mexico", "Brazil"],
    "currency": ["USD", "USD", "USD", "CAD", "CAD", "USD", "USD",
"MXN", "MXN", "BRL"],
}

def raw_data() -> pd.DataFrame:
    """Loading raw data. This simulates loading from a file,
database, or external service."""
    return pd.DataFrame(DATA)

def processed_data(raw_data: pd.DataFrame, cutoff_date: str) ->
pd.DataFrame:
    """Filter out rows before cutoff date and convert currency to
USD."""
    df = raw_data.loc[raw_data.date > cutoff_date].copy()
    df["amound_in_usd"] = df["amount"]
    df.loc[df.country == "Canada", "amound_in_usd"] *= 0.71
    df.loc[df.country == "Brazil", "amound_in_usd"] *= 0.18
    df.loc[df.country == "Mexico", "amound_in_usd"] *= 0.05
    return df

def amount_per_country(processed_data: pd.DataFrame) -> pd.DataFrame:
    """Sum the amount in USD per country"""
    return processed_data.groupby("country")
["amound_in_usd"].sum().to_frame()
```

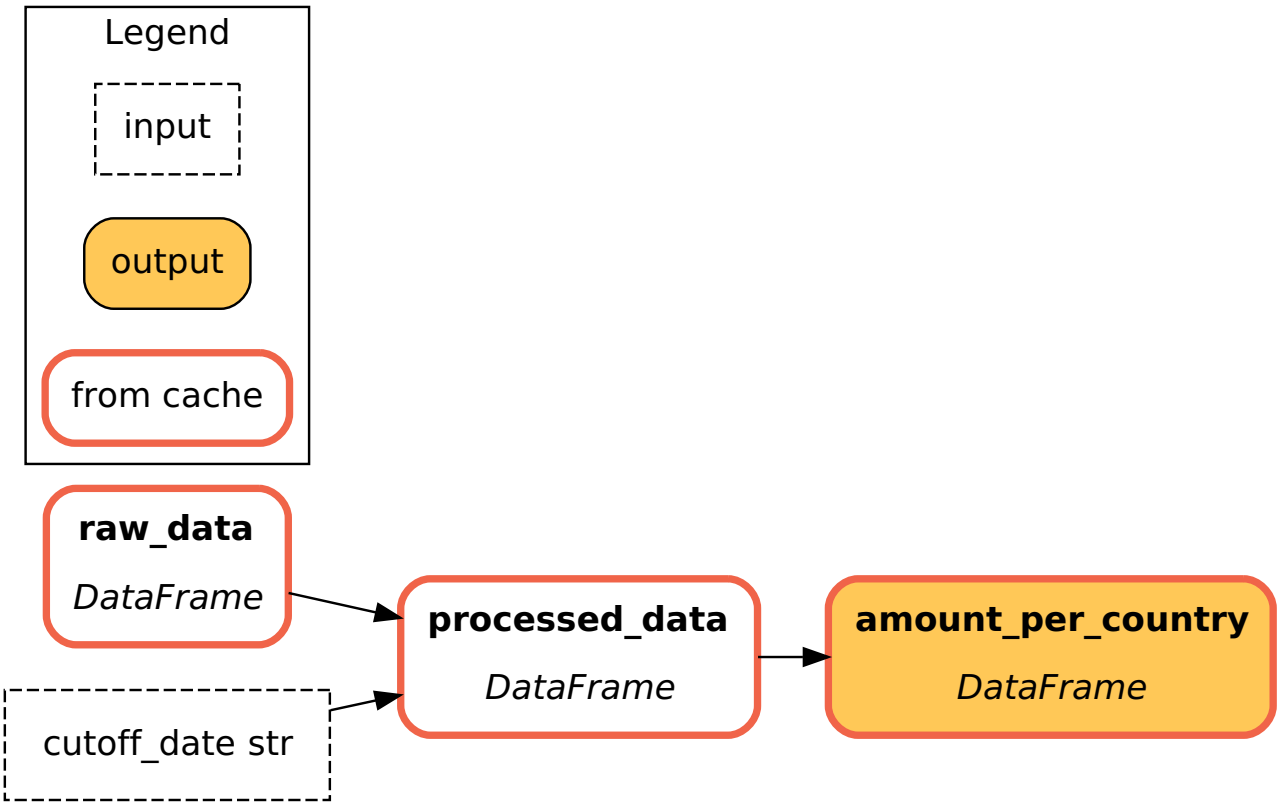
At execution, we see `raw_data` being retrieved along with all downstream nodes. Also, we note that the printed results don't include Brazil nor Mexico.

```
changing_external_dr =
driver.Builder().with_modules(changing_external_module).with_cache().build()

changing_external_results =
changing_external_dr.execute(["amount_per_country"],
inputs={"cutoff_date": "2024-09-01"})
print()
```

```
print(changing_external_results["amount_per_country"].head())
print()
changing_external_dr.cache.view_run()
```

```
raw_data::result_store::get_result::hit
processed_data::result_store::get_result::hit
amount_per_country::result_store::get_result::hit
```

```
         amound_in_usd
country
Canada          941.957
USA            1719.240
```



`.with_cache()` to specify caching behavior

Here, we build a new `Driver` with the same `changing_external_module`, but we specify in `.with_cache()` to always recompute `raw_data`.

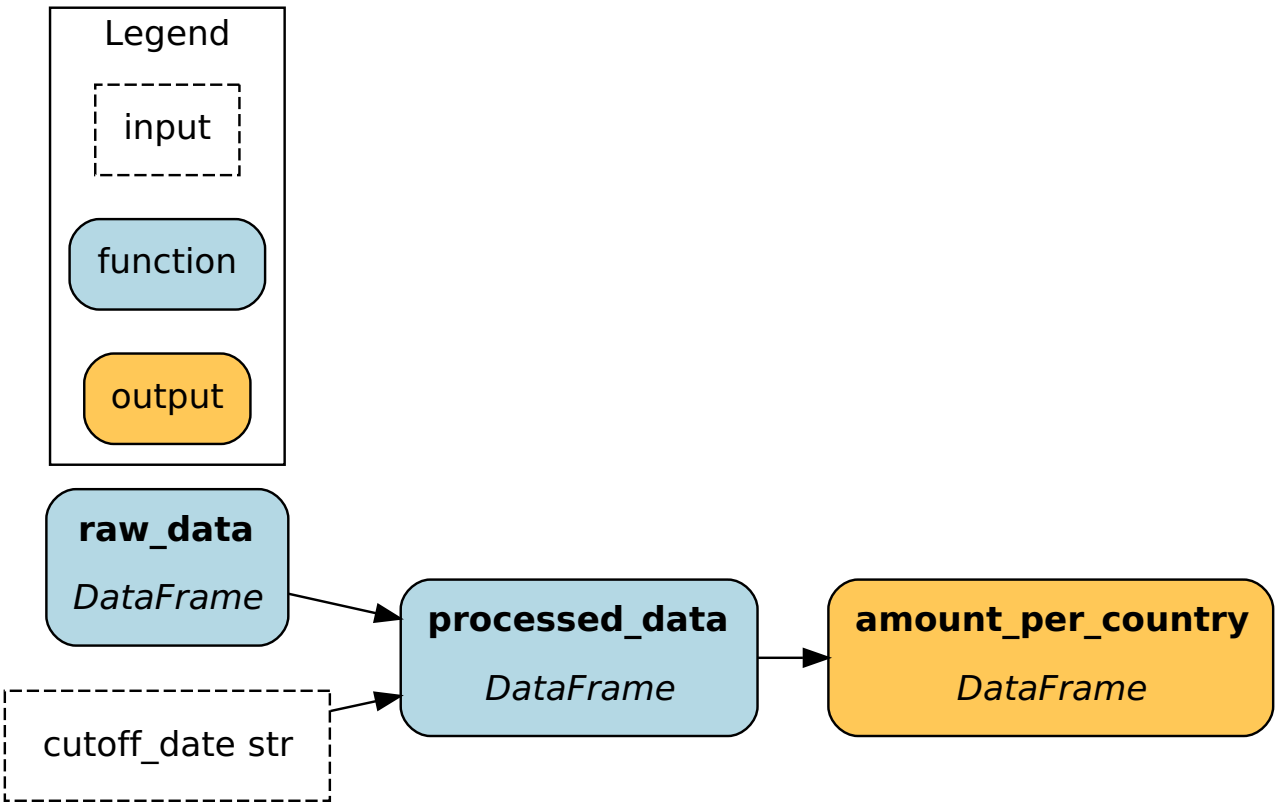
The visualization shows that `raw_data` was executed, and because of the new data, all downstream nodes also need to be executed. The results now include Brazil and Mexico.

```
changing_external_with_cache_dr =
driver.Builder().with_modules(changing_external_module).with_cache(recompute=["r
```

```
changing_external_with_cache_results =
changing_external_with_cache_dr.execute(["amount_per_country"],
inputs={"cutoff_date": "2024-09-01"})
print()
print(changing_external_with_cache_results["amount_per_country"].head())
print()
changing_external_with_cache_dr.cache.view_run()
```

```
raw_data::adapter::execute_node
processed_data::adapter::execute_node
amount_per_country::adapter::execute_node
```

```
         amound_in_usd
country
Brazil         77.9004
Canada        941.9570
Mexico         46.2170
USA          2959.7600
```



`@cache` to specify caching behavior

Another way to specify the `RECOMPUTE` behavior is to use the `@cache` decorator.

```
%%cell_to_module changing_external_decorator_module
import pandas as pd
from hamilton.function_modifiers import cache

DATA = {
    "cities": ["New York", "Los Angeles", "Chicago", "Montréal",
"Vancouver", "Houston", "Phoenix", "Mexico City", "Chihuahua City",
"Rio de Janeiro"],
    "date": ["2024-09-13", "2024-09-12", "2024-09-11", "2024-09-11",
"2024-09-09", "2024-09-08", "2024-09-07", "2024-09-06", "2024-09-05",
"2024-09-04"],
    "amount": [478.23, 251.67, 989.34, 742.14, 584.56, 321.85,
918.67, 135.22, 789.12, 432.78],
    "country": ["USA", "USA", "USA", "Canada", "Canada", "USA",
"USA", "Mexico", "Mexico", "Brazil"],
    "currency": ["USD", "USD", "USD", "CAD", "CAD", "USD", "USD",
"MXN", "MXN", "BRL"],
}

@cache(behavior="recompute")
def raw_data() -> pd.DataFrame:
    """Loading raw data. This simulates loading from a file,
database, or external service."""
    return pd.DataFrame(DATA)

def processed_data(raw_data: pd.DataFrame, cutoff_date: str) ->
pd.DataFrame:
    """Filter out rows before cutoff date and convert currency to
USD."""
    df = raw_data.loc[raw_data.date > cutoff_date].copy()
    df["amound_in_usd"] = df["amount"]
    df.loc[df.country == "Canada", "amound_in_usd"] *= 0.71
    df.loc[df.country == "Brazil", "amound_in_usd"] *= 0.18
    df.loc[df.country == "Mexico", "amound_in_usd"] *= 0.05
    return df

def amount_per_country(processed_data: pd.DataFrame) -> pd.DataFrame:
    """Sum the amount in USD per country"""
    return processed_data.groupby("country")
["amound_in_usd"].sum().to_frame()
```

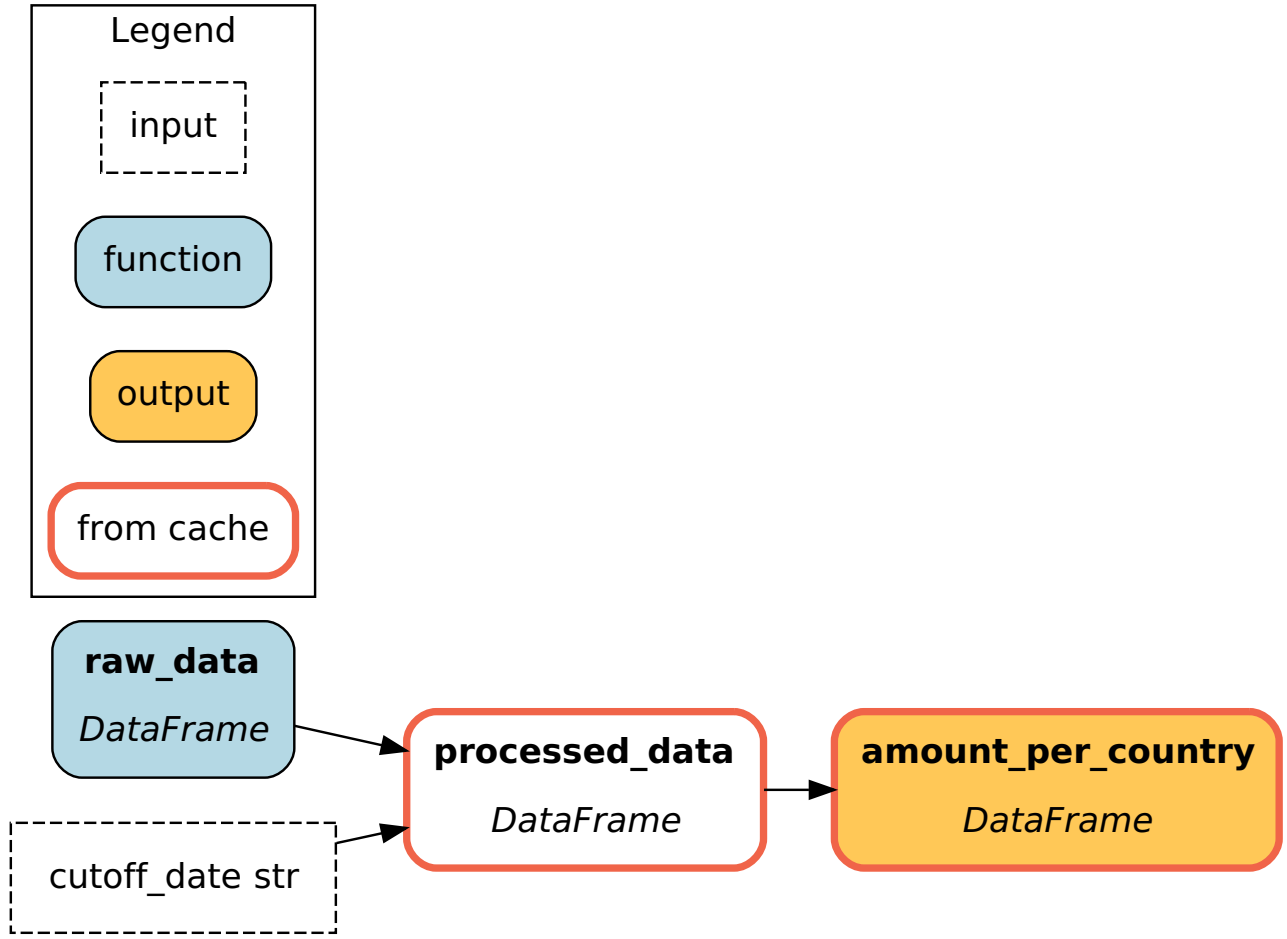
We build a new `Driver` with `changing_external_cache_decorator_module`, which includes the `@cache` decorator. Note that we don't specify anything in `.with_cache()`.

```
changing_external_decorator_dr = (
    driver.Builder()
    .with_modules(changing_external_decorator_module)
    .with_cache()
    .build()
```

```
)

changing_external_decorator_results =
changing_external_decorator_dr.execute(
    ["amount_per_country"],
    inputs={"cutoff_date": "2024-09-01"}
)
print()
print(changing_external_decorator_results["amount_per_country"].head())
print()
changing_external_decorator_dr.cache.view_run()
```

```
raw_data::adapter::execute_node
processed_data::result_store::get_result::hit
amount_per_country::result_store::get_result::hit
```

```
         amound_in_usd
country
Brazil         77.9004
Canada        941.9570
Mexico         46.2170
USA          2959.7600
```



We see that `raw_data` was re-executed. Then, `processed_data` and `amount_per_country` can be retrieved because they were produced just before by the `changing_external_with_cache_dr`

When to use `@cache` vs. `.with_cache()`?

Specifying the caching behavior via `.with_cache()` or `@cache` is entirely equivalent. There are benefits to either approach:

- `@cache`: specify behavior at the dataflow-level. The behavior is tied to the node and will be picked up by all `Driver` loading the module. This can prevent errors or unexpected behaviors for users of that dataflow.
- `.with_cache()`: specify behavior at the `Driver`-level. Gives the flexiblity to change the behavior without modifying the dataflow code and committing changes. You might be ok with `DEFAULT` during development, but want to ensure `RECOMPUTE` in production.

Importantly, the behavior specified in `.with_cache(...)` overrides whatever is in `@cache` because it is closer to execution. For example, having `.with_cache(default=["raw_data"])` `@cache(behavior="recompute")` would force `DEFAULT` behavior.

⛔ **Important**: Using the `@cache` decorator alone doesn't enable caching; adding `.with_cache()` to the `Builder` does. The decorator is only a mean to specify special behaviors for a node.

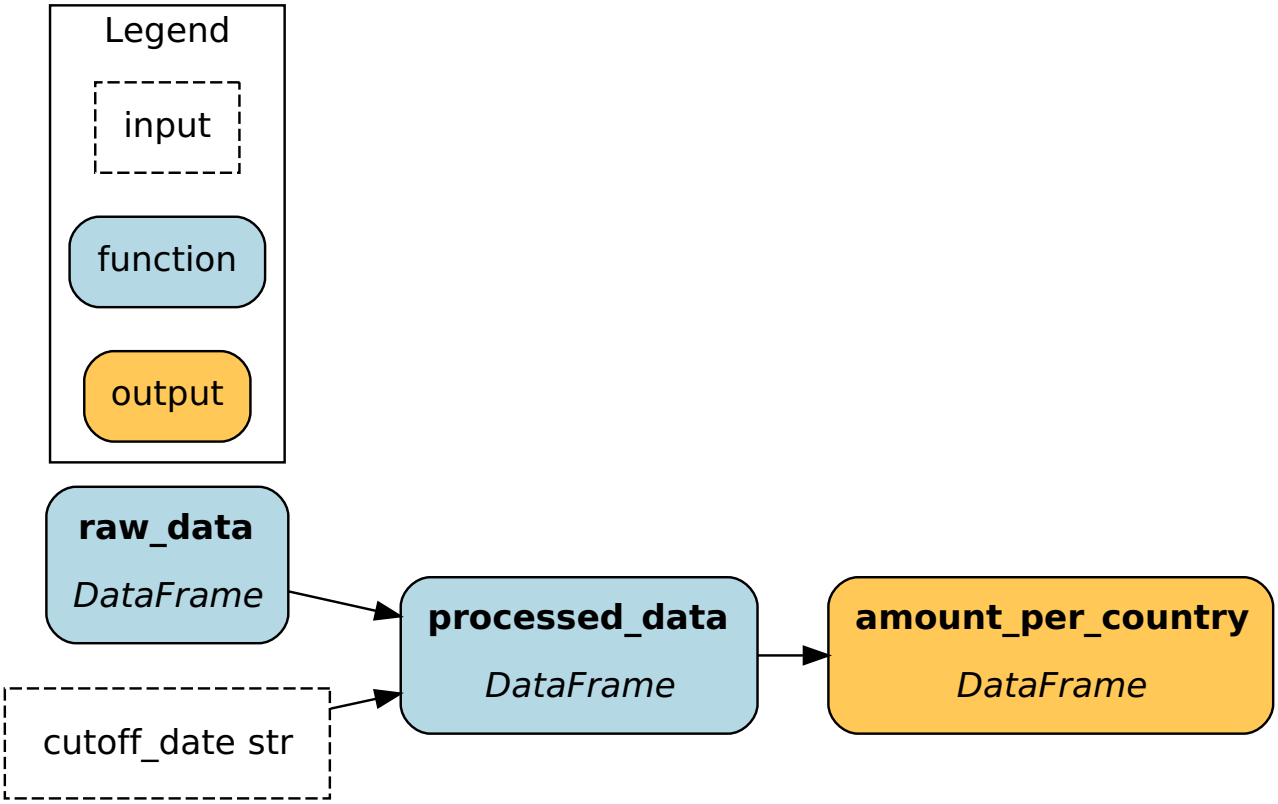
Force recompute all

By specifying `.with_cache(recompute=True)`, you are setting the behavior `RECOMPUTE` for all nodes. This forces recomputation, which is useful for producing a "cache refresh" with up-to-date values.

```
recompute_all_dr = (
    driver.Builder()
    .with_modules(changing_external_decorator_module)
    .with_cache(recompute=True)
    .build()
)

recompute_all_results = recompute_all_dr.execute(
    ["amount_per_country"],
    inputs={"cutoff_date": "2024-09-01"}
)
print()
print(recompute_all_results["amount_per_country"].head())
print()
recompute_all_dr.cache.view_run()
```

```
raw_data::adapter::execute_node
processed_data::adapter::execute_node
amount_per_country::adapter::execute_node
```

```
         amound_in_usd
country
Brazil         77.9004
Canada        941.9570
Mexico         46.2170
USA          2959.7600
```



We see that all nodes were recomputed.

Setting default behavior

Once you enable caching using `.with_cache()`, it is a "opt-out" feature by default. This means all nodes are cached unless you set the `DISABLE` behavior via `@cache` or `.with_cache(disable=[...])`. This can become difficult to manage as the number of nodes increases.

You can make it an "opt-in" feature by setting `default_behavior="disable"` in `.with_cache()`. This way, you're using caching, but only for nodes explicitly specified in `@cache` or `.with_cache()`.

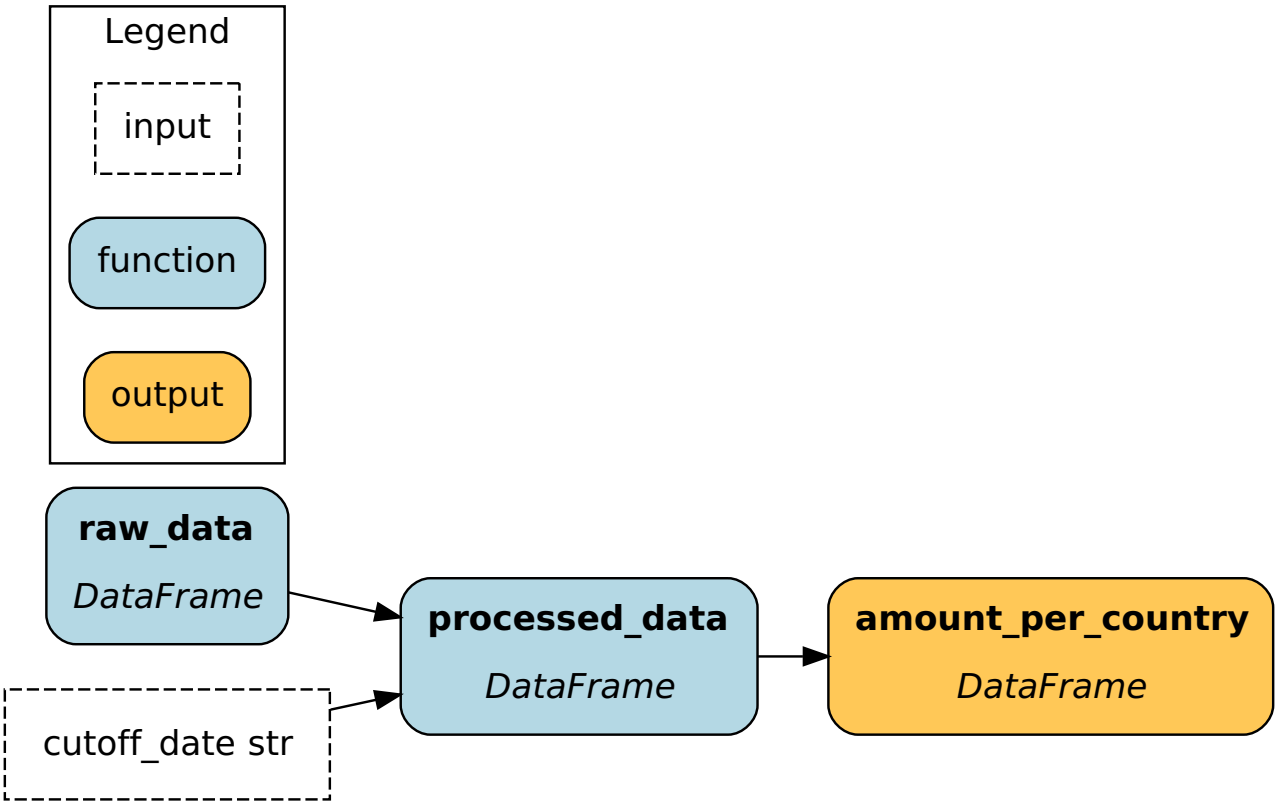
Here, we build a `Driver` with the `changing_external_decorator_module`, where `raw_data` was set to have behavior `RECOMPUTE`, and set the default behavior to `DISABLE`.

```
default_behavior_dr = (
    driver.Builder()
    .with_modules(changing_external_decorator_module)
    .with_cache(default_behavior="disable")
    .build()
)

default_behavior_results = default_behavior_dr.execute(
    ["amount_per_country"],
```

```
    inputs={"cutoff_date": "2024-09-01"}
)
print()
print(default_behavior_results["amount_per_country"].head())
print()
default_behavior_dr.cache.view_run()
```

```
raw_data::adapter::execute_node
processed_data::adapter::execute_node
amount_per_country::adapter::execute_node
```

```
         amound_in_usd
country
Brazil         77.9004
Canada        941.9570
Mexico         46.2170
USA          2959.7600
```




```
default_behavior_dr.cache.behaviors[default_behavior_dr.cache.last_run_id]
```

```
{'amount_per_country': <CachingBehavior.DISABLE: 3>,
 'processed_data': <CachingBehavior.DISABLE: 3>,
```

```
 'raw_data': <CachingBehavior.RECOMPUTE: 2>,
 'cutoff_date': <CachingBehavior.DISABLE: 3>}
```

Materializers

NOTE. You can skip this section if you're not using materializers.

`DataLoader` and `DataSaver` (collectively "materializers") are special Hamilton nodes that connect your dataflow to external data (files, databases, etc.). These constructs are safe to use with caching and are complementary.

Caching

- writing and reading shorter-term data to be used with the dataflow
- strong connection between the code and the data
- automatically handle multiple versions of the same dataset

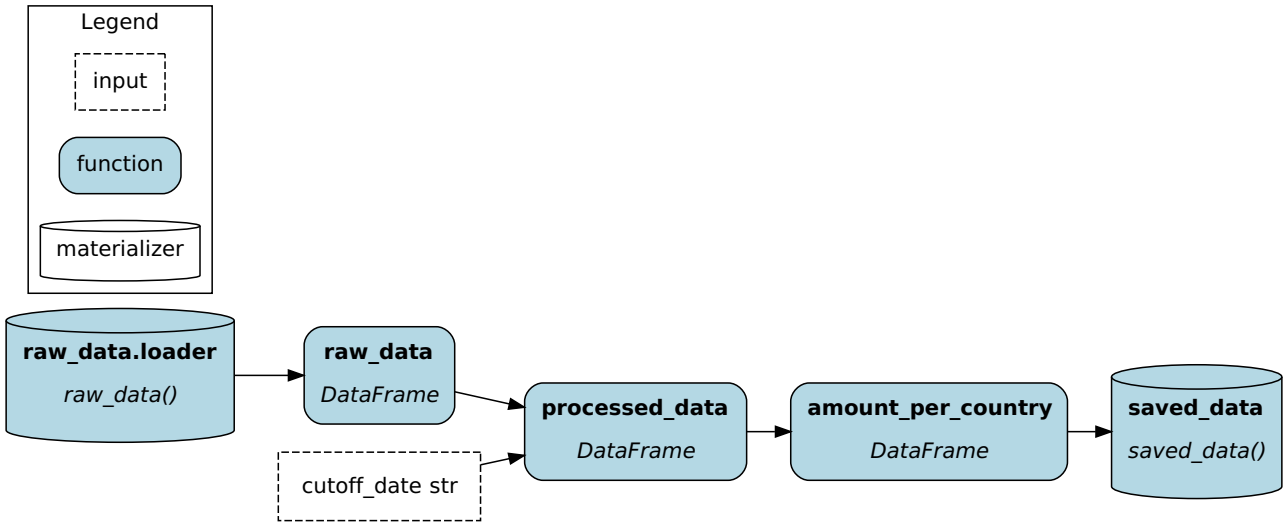
Materializers

- robust mechanism to read/write data from many sources
- data isn't necessarily meant to be used with Hamilton (e.g., loading from a warehouse, outputting a report).
- typically outputs to a static destination; each write overwrites the previous stored dataset.

The next cell uses `@dataloader` and `@datasaver` decorators. In the visualization, we see the added `raw_data.loader` and `saved_data` nodes.

```
%%cell_to_module materializers_module -d
import pandas as pd
from hamilton.function_modifiers import dataloader, datasaver

DATA = {
    "cities": ["New York", "Los Angeles", "Chicago", "Montréal",
"Vancouver", "Houston", "Phoenix", "Mexico City", "Chihuahua City",
"Rio de Janeiro"],
    "date": ["2024-09-13", "2024-09-12", "2024-09-11", "2024-09-11",
"2024-09-09", "2024-09-08", "2024-09-07", "2024-09-06", "2024-09-05",
"2024-09-04"],
    "amount": [478.23, 251.67, 989.34, 742.14, 584.56, 321.85,
918.67, 135.22, 789.12, 432.78],
    "country": ["USA", "USA", "USA", "Canada", "Canada", "USA",
"USA", "Mexico", "Mexico", "Brazil"],
    "currency": ["USD", "USD", "USD", "CAD", "CAD", "USD", "USD",
"MXN", "MXN", "BRL"],
```

```
}

@dataloader()
def raw_data() -> tuple[pd.DataFrame, dict]:
    """Loading raw data. This simulates loading from a file,
database, or external service."""
    data = pd.DataFrame(DATA)
    metadata = {"source": "notebook", "format": "json"}
    return data, metadata

def processed_data(raw_data: pd.DataFrame, cutoff_date: str) ->
pd.DataFrame:
    """Filter out rows before cutoff date and convert currency to
USD."""
    df = raw_data.loc[raw_data.date > cutoff_date].copy()
    df["amound_in_usd"] = df["amount"]
    df.loc[df.country == "Canada", "amound_in_usd"] *= 0.71
    df.loc[df.country == "Brazil", "amound_in_usd"] *= 0.18
    df.loc[df.country == "Mexico", "amound_in_usd"] *= 0.05
    return df

def amount_per_country(processed_data: pd.DataFrame) -> pd.DataFrame:
    """Sum the amount in USD per country"""
    return processed_data.groupby("country")
["amound_in_usd"].sum().to_frame()

@datasaver()
def saved_data(amount_per_country: pd.DataFrame) -> dict:
    amount_per_country.to_parquet("./saved_data.parquet")
    metadata = {"source": "notebook", "format": "parquet"}
    return metadata
```

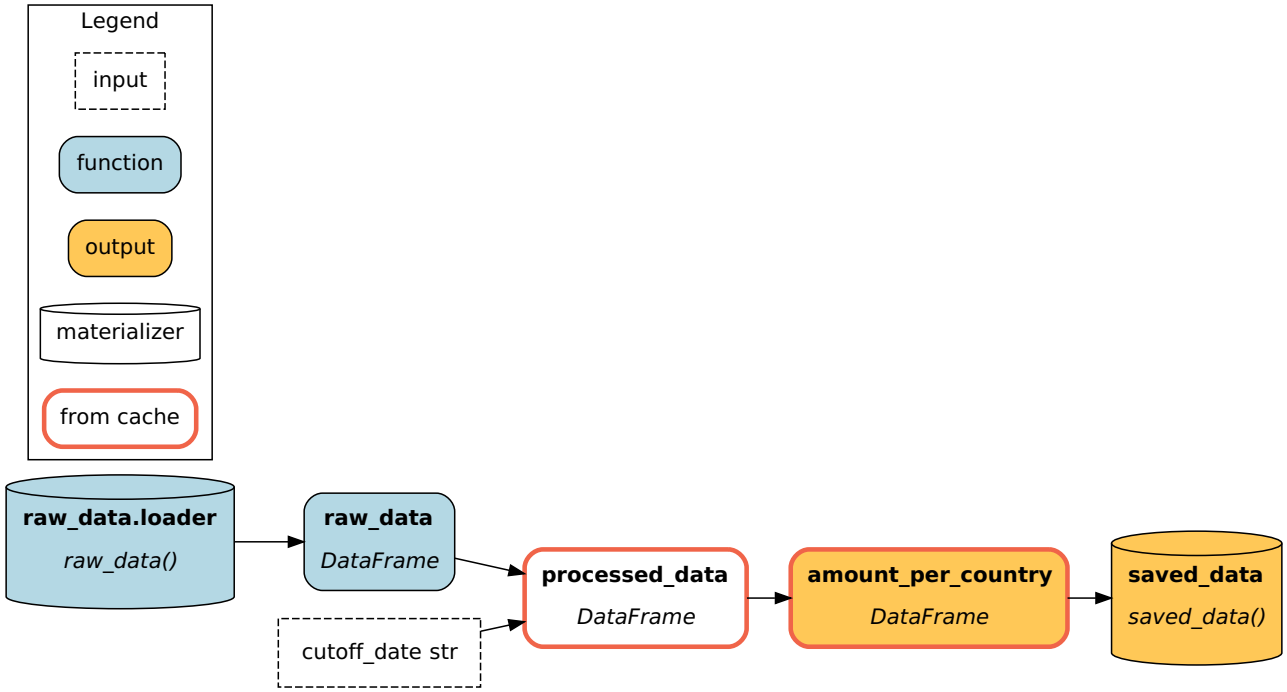


Next, we build a `Driver` as usual.

```
materializers_dr = (
    driver.Builder()
    .with_modules(materializers_module)
    .with_cache()
    .build()
)

materializers_results = materializers_dr.execute(
    ["amount_per_country", "saved_data"],
    inputs={"cutoff_date": "2024-09-01"}
)
print()
print(materializers_results["amount_per_country"].head())
print()
materializers_dr.cache.view_run()
```

```
raw_data.loader::adapter::execute_node
raw_data::adapter::execute_node
processed_data::result_store::get_result::hit
amount_per_country::result_store::get_result::hit
saved_data::adapter::execute_node
```

```
         amound_in_usd
country
Brazil         77.9004
Canada        941.9570
Mexico         46.2170
USA          2959.7600
```

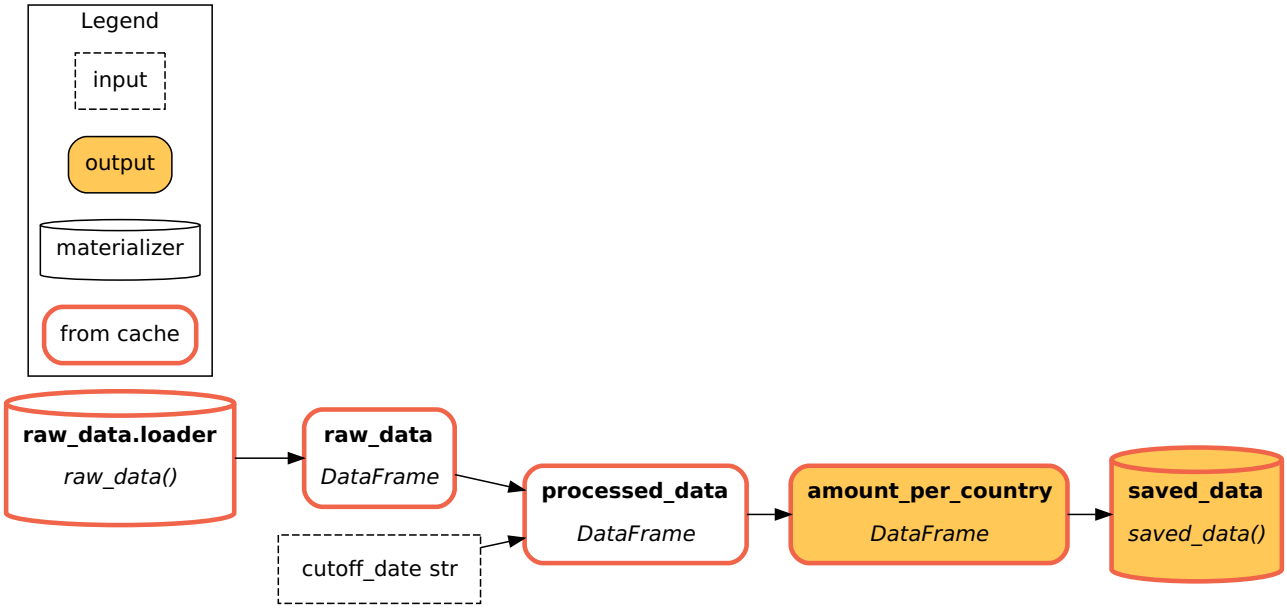


We execute the dataflow a second time to show that loaders and savers are just like any other node; they can be cached and retrieved.

```
materializers_results = materializers_dr.execute(
    ["amount_per_country", "saved_data"],
    inputs={"cutoff_date": "2024-09-01"}
)
print()
print(materializers_results["amount_per_country"].head())
print()
materializers_dr.cache.view_run()
```

```
raw_data.loader::result_store::get_result::hit
raw_data::result_store::get_result::hit
processed_data::result_store::get_result::hit
amount_per_country::result_store::get_result::hit
saved_data::result_store::get_result::hit
```

```
         amound_in_usd
country
Brazil         77.9004
Canada        941.9570
Mexico         46.2170
USA          2959.7600
```



Usage patterns

Here are a few common scenarios:

Loading data is expensive: Your dataflow uses a `DataLoader` to get data from Snowflake. You want to load it once and cache it. When executing your dataflow, you want to use your cached copy to save query time, egress costs, etc.

- Use the `DEFAULT` caching behavior for loaders.

Only save new data: You run the dataflow multiple times (maybe with different parameters or on a schedule) and only want to write to destination when the data changes.

- Use the `DEFAULT` caching behavior for savers.

Always read the latest data: You want to use caching, but also ensure the dataflow always uses the latest data. This involves executing the `DataLoader` every time, get the data in-memory, version it, and then determine what needs to be executed (see **Changing external data**).

- Use the `RECOMPUTE` caching behavior for loaders.

Use the parameters `default_loader_behavior` or `default_saver_behavior` of the `.with_cache()` clause to specify the behavior for all loaders or savers.

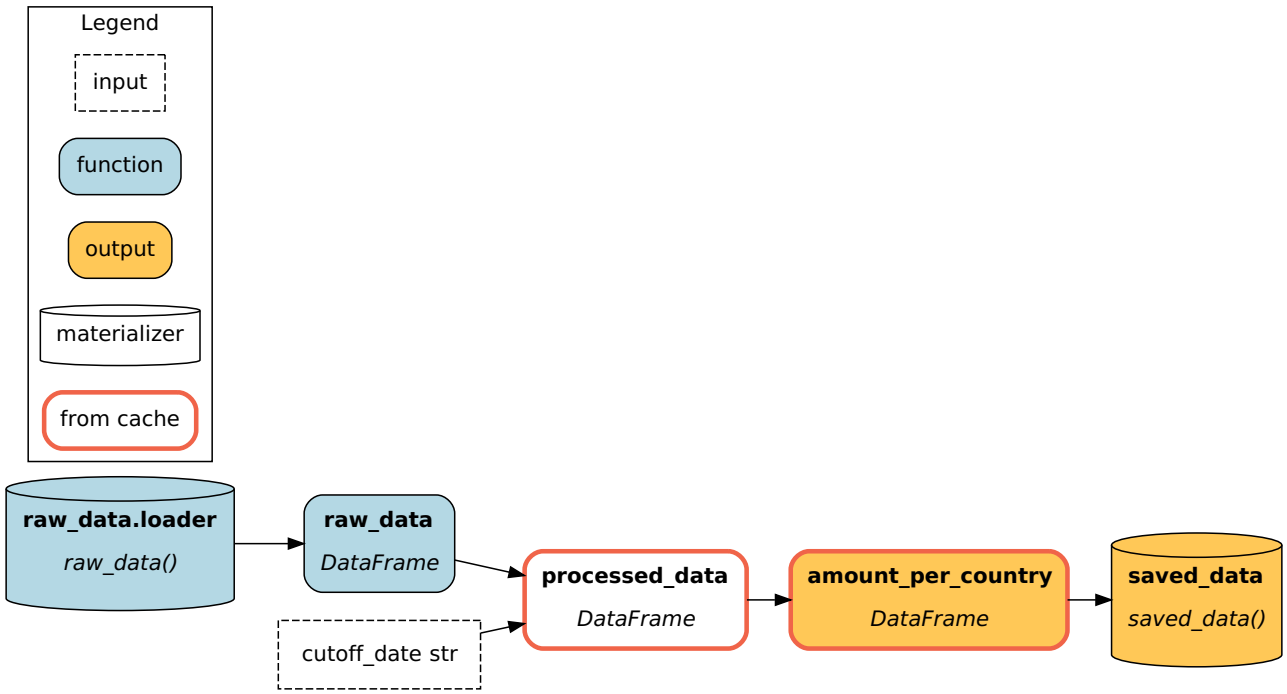
> NOTE. The **Caching + materializers tutorial** notebook details how to achieve granular control over loader and saver behaviors.

```
materializers_dr_2 = (
    driver.Builder()
    .with_modules(materializers_module)
    .with_cache(
```

```
        default_loader_behavior="recompute",
        default_saver_behavior="disable"
    )
    .build()
)

materializers_results_2 = materializers_dr_2.execute(
    ["amount_per_country", "saved_data"],
    inputs={"cutoff_date": "2024-09-01"}
)
print()
print(materializers_results_2["amount_per_country"].head())
print()
materializers_dr_2.cache.view_run()
```

```
raw_data.loader::adapter::execute_node
raw_data::adapter::execute_node
processed_data::result_store::get_result::hit
amount_per_country::result_store::get_result::hit
saved_data::adapter::execute_node
```

```
         amound_in_usd
country
Brazil         77.9004
Canada        941.9570
Mexico         46.2170
USA          2959.7600
```



```
materializers_dr_2.cache.behaviors[materializers_dr_2.cache.last_run_id]
```

```
{'amount_per_country': <CachingBehavior.DEFAULT: 1>,
 'processed_data': <CachingBehavior.DEFAULT: 1>,
 'raw_data.loader': <CachingBehavior.RECOMPUTE: 2>,
 'raw_data': <CachingBehavior.RECOMPUTE: 2>,
 'saved_data': <CachingBehavior.DISABLE: 3>,
 'cutoff_date': <CachingBehavior.DEFAULT: 1>}
```

Changing the cache format

By default, results are stored in `pickle` format. It's a convenient default but comes with caveats. You can use the `@cache` decorator to specify another file format for storing results.

By default this includes:

- `json`
- `parquet`
- `csv`
- `excel`
- `file`
- `feather`
- `orc`

This feature uses `DataLoader` and `DataSaver` under the hood and supports all of the same formats (including your custom ones, as long as they take a `path` attribute).

> This is an area of active development. Feel free to share suggestions and feedback!

The next cell sets `processed_data` to be cached using the `parquet` format.

```
%%cell_to_module cache_format_module
import pandas as pd
from hamilton.function_modifiers import cache

DATA = {
    "cities": ["New York", "Los Angeles", "Chicago", "Montréal",
"Vancouver", "Houston", "Phoenix", "Mexico City", "Chihuahua City",
"Rio de Janeiro"],
    "date": ["2024-09-13", "2024-09-12", "2024-09-11", "2024-09-11",
```

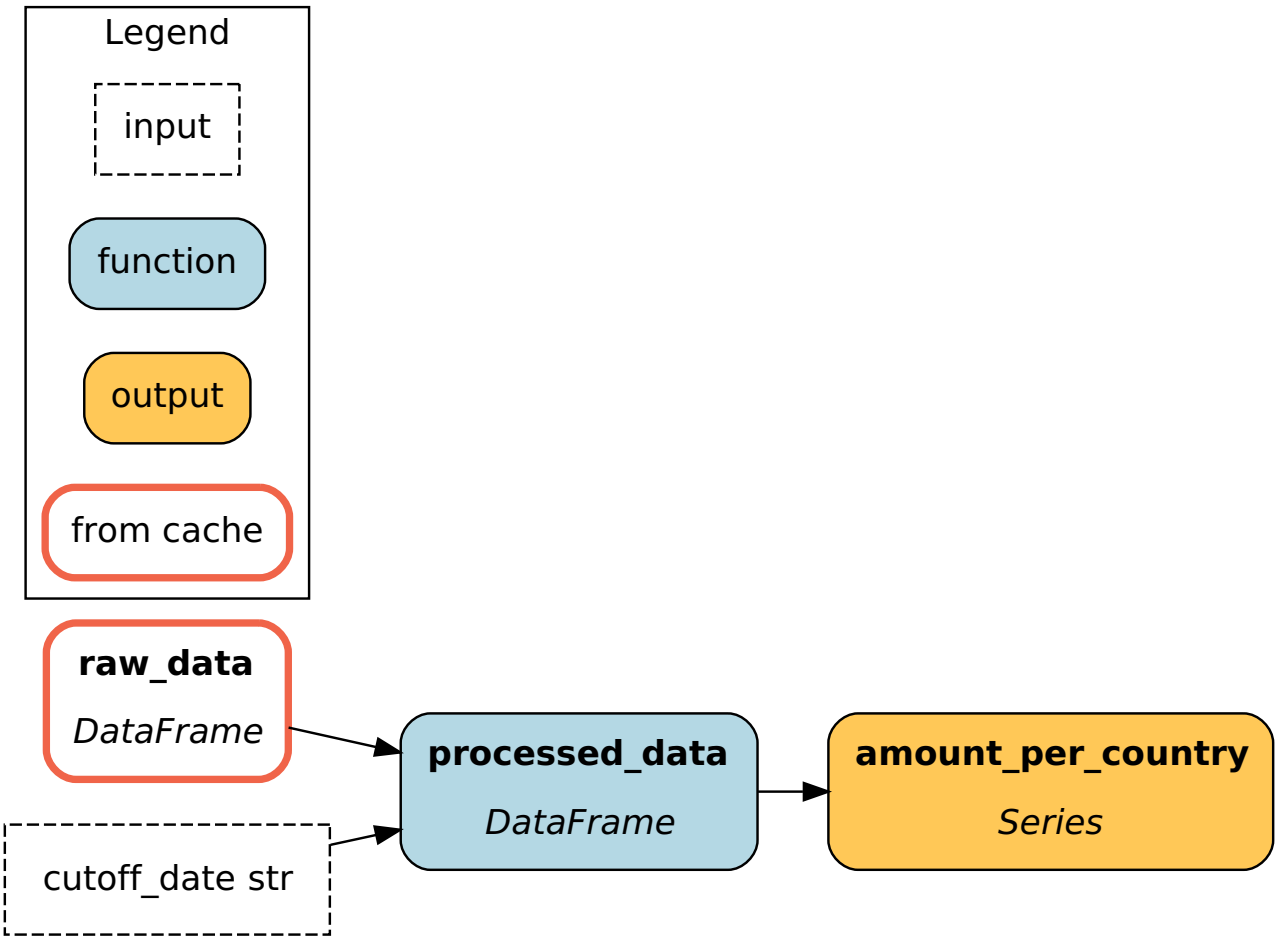
```
"2024-09-09", "2024-09-08", "2024-09-07", "2024-09-06", "2024-09-05",
"2024-09-04"],
    "amount": [478.23, 251.67, 989.34, 742.14, 584.56, 321.85,
918.67, 135.22, 789.12, 432.78],
    "country": ["USA", "USA", "USA", "Canada", "Canada", "USA",
"USA", "Mexico", "Mexico", "Brazil"],
    "currency": ["USD", "USD", "USD", "CAD", "CAD", "USD", "USD",
"MXN", "MXN", "BRL"],
}

def raw_data() -> pd.DataFrame:
    """Loading raw data. This simulates loading from a file,
database, or external service."""
    return pd.DataFrame(DATA)

@cache(format="parquet")
def processed_data(raw_data: pd.DataFrame, cutoff_date: str) ->
pd.DataFrame:
    """Filter out rows before cutoff date and convert currency to
USD."""
    df = raw_data.loc[raw_data.date > cutoff_date].copy()
    df["amound_in_usd"] = df["amount"]
    df.loc[df.country == "Canada", "amound_in_usd"] *= 0.71
    df.loc[df.country == "Brazil", "amound_in_usd"] *= 0.18
    df.loc[df.country == "Mexico", "amound_in_usd"] *= 0.05
    return df

def amount_per_country(processed_data: pd.DataFrame) -> pd.Series:
    """Sum the amount in USD per country"""
    return processed_data.groupby("country")["amound_in_usd"].sum()
```

When executing the dataflow, we see `raw_data` recomputed because it's a dataloader. The result for `processed_data` will be retrieved, but it will be saved again as `.parquet` this time.

```
cache_format_dr =
driver.Builder().with_modules(cache_format_module).with_cache().build()

cache_format_results =
cache_format_dr.execute(["amount_per_country"],
inputs={"cutoff_date": "2024-09-01"})
print()
print(cache_format_results["amount_per_country"].head())
print()
cache_format_dr.cache.view_run()
```

```
raw_data::result_store::get_result::hit
processed_data::adapter::execute_node
amount_per_country::adapter::execute_node
```

```
country
Canada     941.957
USA       1719.240
Name: amound_in_usd, dtype: float64
```



Now, under the `./.hamilton_cache`, there will be two results of the same name, one with the `.parquet` extension and one without. The one without is actually a pickeld `DataLoader` to retrieve the `.parquet` file.

You can access the path programmatically via the `result_store._path_from_data_version(...)` method.

```
data_version =
cache_format_dr.cache.data_versions[cache_format_dr.cache.last_run_id]
["processed_data"]
parquet_path =
cache_format_dr.cache.result_store._path_from_data_version(data_version).with_su
parquet_path.exists()
```

```
True
```

Introspecting the cache

The `Driver.cache` stores information about all executions over its lifetime. Previous `run_id` are available through `Driver.cache.run_ids` and can be used in tandem without other utility functions:

- Resolve the node caching behavior (e.g., “recompute”)
- Access structured logs
- Visualize the cache execution

Also, `Driver.cache.last_run_id` is a shortcut to the most recent execution.

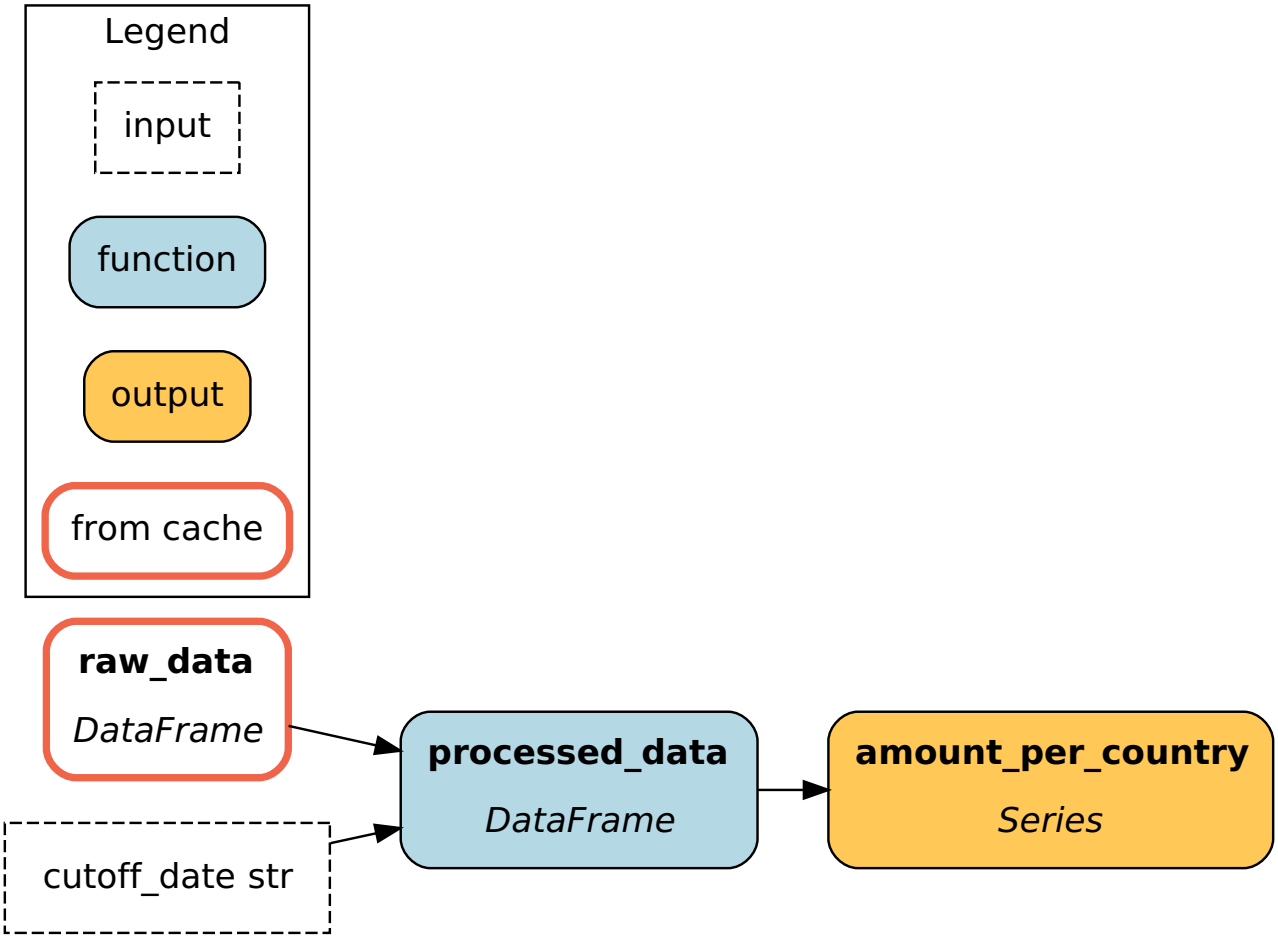
```
cache_format_dr.cache.resolve_behaviors(cache_format_dr.cache.last_run_id)
```

```
{'amount_per_country': <CachingBehavior.DEFAULT: 1>,
 'processed_data': <CachingBehavior.DEFAULT: 1>,
 'raw_data': <CachingBehavior.DEFAULT: 1>,
 'cutoff_date': <CachingBehavior.DEFAULT: 1>}
```

```
run_logs =
cache_format_dr.cache.logs(cache_format_dr.cache.last_run_id,
level="debug")
for event in run_logs["processed_data"]:
    print(event)
```

```
processed_data::adapter::resolve_behavior
processed_data::adapter::set_cache_key
processed_data::adapter::get_cache_key::hit
processed_data::adapter::get_data_version::miss
processed_data::metadata_store::get_data_version::miss
processed_data::adapter::execute_node
processed_data::adapter::set_data_version
processed_data::metadata_store::set_data_version
processed_data::adapter::get_cache_key::hit
processed_data::adapter::get_data_version::hit
processed_data::result_store::set_result
processed_data::adapter::get_data_version::hit
processed_data::adapter::resolve_behavior
```

```
# for `.view_run()` passing no parameter is equivalent to the last
`run_id`
cache_format_dr.cache.view_run(cache_format_dr.cache.last_run_id)
```



Interactively explore runs

By using `ipywidgets` we can easily build a widget to iterate over `run_id` values and display cache information. Below, we create a `Driver` and execute it a few times to generate data then inspect it with a widget.

```
interactive_dr =
driver.Builder().with_modules(cache_format_module).with_cache().build()

interactive_dr.execute(["amount_per_country"],
inputs={"cutoff_date": "2024-09-01"})
interactive_dr.execute(["amount_per_country"],
inputs={"cutoff_date": "2024-09-05"})
interactive_dr.execute(["amount_per_country"],
inputs={"cutoff_date": "2024-09-10"})
interactive_dr.execute(["amount_per_country"],
inputs={"cutoff_date": "2024-09-11"})
```

```
interactive_dr.execute(["amount_per_country"],
inputs={"cutoff_date": "2024-09-13"})
```

```
raw_data::result_store::get_result::hit
processed_data::result_store::get_result::hit
amount_per_country::result_store::get_result::hit
raw_data::result_store::get_result::hit
processed_data::adapter::execute_node
amount_per_country::result_store::get_result::hit
raw_data::result_store::get_result::hit
processed_data::adapter::execute_node
amount_per_country::adapter::execute_node
raw_data::result_store::get_result::hit
processed_data::adapter::execute_node
amount_per_country::adapter::execute_node
raw_data::result_store::get_result::hit
processed_data::adapter::execute_node
amount_per_country::adapter::execute_node
```

```
{'amount_per_country': Series([], Name: amound_in_usd, dtype:
float64)}
```

The following cell allows you to click-and-drag or use arrow-keys to navigate

```
from IPython.display import display
from ipywidgets import SelectionSlider, interact


@interact(run_id=SelectionSlider(options=interactive_dr.cache.run_ids))
def iterate_over_runs(run_id):
    display(interactive_dr.cache.data_versions[run_id])
    display(interactive_dr.cache.view_run(run_id=run_id))
```

Managing storage

Setting the cache `path`

By default, metadata and results are stored under `./.hamilton_cache`, relative to the current directory at execution time. You can also manually set the directory via `.with_cache(path=...)` to isolate or centralize cache storage between dataflows or projects.

Running the next cell will create the directory `./my_other_cache`.

```
manual_path_dr =
driver.Builder().with_modules(cache_format_module).with_cache(path="./
my_other_cache").build()
```

Instantiating the `result_store` and `metadata_store`

If you need to store metadata and results in separate locations, you can do so by instantiating the `result_store` and `metadata_store` manually with their own configuration. In this case, setting `.with_cache(path=...)` would be ignored.

```
from hamilton.caching.stores.file import FileResultStore
from hamilton.caching.stores.sqlite import SQLiteMetadataStore

result_store = FileResultStore(path="./results")
metadata_store = SQLiteMetadataStore(path="./metadata")

manual_stores_dr = (
    driver.Builder()
    .with_modules(cache_format_module)
    .with_cache(
        result_store=result_store,
        metadata_store=metadata_store,
    )
    .build()
)
```

Deleting data and recovering storage

As you use caching, you might be generating a lot of data that you don't need anymore. One straightforward solution is to delete the entire directory where metadata and results are stored.

You can also programmatically call `.delete_all()` on the `result_store` and `metadata_store`, which should reclaim most storage. If you delete results, make sure to also delete metadata. The caching mechanism should figure it out, but it's safer to keep them in sync.

```
manual_stores_dr.cache.metadata_store.delete_all()
manual_stores_dr.cache.result_store.delete_all()
```

Usage patterns

As demonstrated here, caching works great in a notebook environment.

- In addition to iteration speed, caching allows you to restart your kernel or shutdown your computer for the day without worry. When you'll come back, you will still be able to retrieve results from cache.
- A similar benefit is the ability resume execution between environments. For example, you might be running Hamilton in a script, but when a bug happens you can reload these values in a notebook and investigate.
- Caching works great with other adapters like the `HamiltonTracker` that powers the Hamilton UI and the `MLFlowTracker` for experiment tracking.

🚧 INTERNALS

If you're curious the following sections provide details about the caching internals. These APIs are not public and may change without notice.

Manually retrieve results

Using the `Driver.cache` you can directly retrieve results from previous executions. The cache stores "data versions" which are keys for the `result_store`.

Here, we get the `run_id` for the 4th execution (index 3) and the data version for `processed_data` before retrieving its value.

```
run_id = interactive_dr.cache.run_ids[3]
data_version = interactive_dr.cache.data_versions[run_id]
["processed_data"]
result = interactive_dr.cache.result_store.get(data_version)
print(result)
```

```
        cities        date  amount country currency  amound_in_usd
0     New York  2024-09-13  478.23     USA      USD         478.23
1  Los Angeles  2024-09-12  251.67     USA      USD         251.67
```

Decoding the `cache_key`

By now, you should have a better grasp on how Hamilton's caching determines when to execute a node. Internally, it creates a `cache_key` from the `code_version` of the node and the

`data_version` of each dependency. The cache keys are stored on the `Driver.cache` and can be decoded for introspection and debugging.

Here, we get the `run_id` for the 3rd execution (index 2) and the cache key for `amount_per_country`. We then use `decode_key()` to retrieve the `node_name`, `code_version`, and `dependencies_data_versions`.

```
from hamilton.caching.cache_key import decode_key

run_id = interactive_dr.cache.run_ids[2]
cache_key = interactive_dr.cache.cache_keys[run_id]
["amount_per_country"]
decode_key(cache_key)
```

```
{'node_name': 'amount_per_country',
 'code_version':
'c2ccafa54280fbc969870b6baa445211277d7e8cfa98a0821836c175603ffda2',
 'dependencies_data_versions': {'processed_data': 'WgV5-4SfdKTfUY66x-
msj_xXsKNPNTP2guRhfw=='}}
```

Indeed, this match the data version for `processed_data` for the 3rd execution.

```
interactive_dr.cache.data_versions[run_id]["processed_data"]
```

```
'WgV5-4SfdKTfUY66x-msj_xXsKNPNTP2guRhfw=='
```

Manually retrieve metadata

In addition to the `result_store`, there is a `metadata_store` that contains mapping between `cache_key` and `data_version` (cache keys are unique, but many can point to the same data).

Using the knowledge from the previous section, we can use the cache key for `amount_per_country` to retrieve its `data_version` and result. It's also possible to decode its `cache_key`, and get the `data_version` for its dependencies, making the node execution reproducible.

```
run_id = interactive_dr.cache.run_ids[2]
cache_key = interactive_dr.cache.cache_keys[run_id]
["amount_per_country"]
amount_data_version =
interactive_dr.cache.metadata_store.get(cache_key)
amount_result =
```

```
interactive_dr.cache.result_store.get(amount_data_version)
print(amount_result)
```

```
country
Canada     526.9194
USA       1719.2400
Name: amound_in_usd, dtype: float64
```

```
for dep_name, dependency_data_version in decode_key(cache_key)
["dependencies_data_versions"].items():
    dep_result =
interactive_dr.cache.result_store.get(dependency_data_version)
    print(dep_name)
    print(dep_result)
    print()
```

```
processed_data
        cities        date  amount country currency  amound_in_usd
0     New York  2024-09-13  478.23     USA      USD         478.23
1  Los Angeles  2024-09-12  251.67     USA      USD         251.67
2      Chicago  2024-09-11  989.34     USA      USD         989.34
3     Montréal  2024-09-11  742.14  Canada      CAD       526.9194
```

Feature engineering

Apache Hamilton's roots are in time-series offline feature engineering. But it can be used for any type of feature engineering: offline, streaming, online. All our examples are oriented towards Pandas, but rest assured, you can use Apache Hamilton with any python objects, e.g. numpy, polars, and even pyspark.

Here's a 20 minute video (slides), with brief backstory on Apache Hamilton, and an overview (at around the 8:52 mark) of how to use it for feature engineering which was presented at the Feature Store Summit 2022:

Otherwise here we present a high level overview and then direct users to the examples folder for more details. We suggest reading the Offline Feature Engineering section first, since it's the most common use case, and helps explain the python module structure you should be going for with Apache Hamilton. If you need more guidance here, please reach out to us on slack.

Offline Feature Engineering

To use Apache Hamilton for offline feature engineering, a common pattern is:

1. create a data_loader module(s) that loads the data from the source(s) (e.g. a database, a csv file, etc.).
2. create feature transform module(s) that transform the data into features.
3. create a data set module(s) that combines the data_loader and feature transform modules if you want to connect fitting a model with Apache Hamilton. Or, you do this data set definition in your driver code.

Here is a sketch of the above pattern:

```
# data_loader.py
@extract_columns(*...)  # you can choose to expose individual columns
def load_data(...) -> pd.DataFrame:
    return pd.read_csv(...)
...
# feature_transform.py
def feature_a(raw_input_a: pd.Series, ...) -> pd.Series:
    return raw_input_a + ...
...
# dataset.py (optional)
def model_set_x(feature_a: pd.Series, ...) -> pd.DataFrame:
    return pd.DataFrame({'feature_a': feature_a, ...})
# run.py
def main():
    dr = driver.Driver(config, data_loader, feature_transform,
dataset)
    feature_df = dr.execute([feature_transform.feature_a, ...])
    ...
```

Apache Hamilton Example

We do not provide a specific example here, since most of the examples in the examples folder fall under this category. Some examples to browse:

- Hello World shows the basics of how to use Apache Hamilton.
- Data Quality shows how to incorporate runtime data quality checks into your feature engineering pipeline.
- Time-series Kaggle Example shows one way to structure your code to ingest, create features, and fit a model.

- Feature engineering in multiple contexts helps show how you can use Apache Hamilton in multiple contexts reusing code where possible, e.g. offline, & online.
- PySpark UDF Map Examples shows how to use Apache Hamilton to encode map operations for use with PySpark.

Streaming Feature Engineering

Right now, there is no specific streaming support. Instead, we model the problem as we would for offline. Apache Hamilton has an *inputs=* argument to the *execute()* function in the driver. This allows you to then instantiate a Apache Hamilton Driver once, and then call *execute()* multiple times with different inputs. Otherwise you'd have a similar python module structure as for offline feature engineering – perhaps just dropping the data_loader module since you would provide the inputs directly to the *execute()* function.

Here's a sketch of how you might use Apache Hamilton in conjunction with a Kafka Client:

```
# run.py
def main():
    kakfa_client = KafkaClient(...)
    dr = driver.Driver(config, feature_transform)
    for batch in kafka_client.get_batches():  # this is pseudo code,
but you get the idea
        feature_df = dr.execute([feature_transform.feature_a, ...],
inputs=batch.to_dict())
        # do something / emit back to kafka, etc.
```

Caveats to think about. Here are some things to think about when using Apache Hamilton for streaming feature engineering:

- aggregation features, you likely want to understand whether you want to aggregate over the entire stream or just the current batch, or load values that were computed offline.

Apache Hamilton Example

Currently we don't have a streaming example. But we are working on it. We direct users to look at the online example for now, since conceptually from a modularity stand point, things would be set up in a similar way.

Online Feature Engineering

Online feature engineering can be quite simple or quite complex, depending on your situation. However, good news is, that Apache Hamilton should be able to help you in any situation. The modularity of Apache Hamilton allows you to swap out implementations of features easily, as well as override values, and even ask the Driver what features are required from the source data to create the features that you want. We think Apache Hamilton can help you keep things simple, but then extend to helping you handle more complex situations.

The basic structure of your python modules, does not change. Depending on whether you want Apache Hamilton to load data from a feature store, or you have all the data passed in, you just need to appropriately segment your feature transforms into modules, or use the *@config.** decorator, to help you segment your feature computation dataflow to give you the flexibility you need.

Caveats to think about. Here are some things to think about when using Apache Hamilton for online feature engineering:

- aggregation features, most likely you'll want to load aggregated feature values that were computed offline, rather than compute them live.

We skip showing a sketch of structure here, and invite you to look at the examples below.

Apache Hamilton Example

We direct users to look at Feature engineering in multiple contexts that currently describes two scenarios around how you could incorporate Apache Hamilton into an online web-service, and have it aligned with your batch offline processes. Note, these examples should give you the high level first principles view of how to do things. Since having something running in production , we didn't want to get too specific.

Write once, run anywhere blog post:

For a comprehensive post on writing a feature once and using it anywhere see this blog. The companion example code can be found here.

Best Egg Platform Blog Post:

For an overview of how Best Egg built their feature platform on Apache Hamilton see this blog.

FAQ

Q. Can I use Apache Hamilton for feature engineering with Feast?

Yes, you can use Apache Hamilton with Feast. See our [Feast example](https://github.com/apache/hamilton/tree/main/examples/feast) and accompanying [blog post](https://blog.dagworks.io/p/featurization-integrating-hamilton). Typically people use Apache Hamilton on the offline side to compute features that then get pushed to Feast. For the online side it varies as to how to integrate the two.

Model training

As Apache Hamilton is a generic library for representing dataflows in pandas, it can be used for a wide array of tasks. One of the more common applications is using hamilton for training, testing, and executing machine learning models, all the way from feature-engineering through training and inference.

The following two examples show how to use Apache Hamilton to model an entire ML pipeline:

1. A classification pipeline for the iris dataset using scikit-learn
2. An implementation of the m5 kaggle competition to do time-series forecasting on unit sales for using Walmart data.

The goal of these is to get you comfortable with building out ML pipelines using hamilton, potentially giving you inspiration/templates from which you can get started.

LLM workflows

Apache Hamilton is great for describing dataflows, and a lot of “actions” you want an “agent” to perform can be described as one, e.g. create an embedding of some passed in text, query a vector database, find the nearest documents, etc.

The benefit of using Apache Hamilton within an LLM Powered app is that:

1. you can visualize the dataflow.
2. you can easily test, modify, compose, and reuse dataflows. For example, you can easily test the dataflow that creates an embedding of some text without having to worry about the rest of the dataflow.

3. you can easily swap out the implementation details of components surgically. For example, you can swap out the vector database client based on configuration, this helps in ensuring you can quickly and easily modify/update your dataflow and have confidence around the impact of that change.

4. you can use functionality like runtime data quality checks/extend Apache Hamilton's capabilities with your own needs to inject/augment your dataflow with additional functionality, e.g. caching, logging, etc.

5. you can request the intermediate outputs of a dataflow by requesting it as output without any surgery required to any of your code to do so. This is useful for debugging.

The following examples show how to use Apache Hamilton for LLM workflows:

- How to use "OpenAI functions" with a Knowledge Base
- Modular LLM Stack with blog post
- PDF Summarizer which shows a partial RAG workflow (just missing going to a vector store to get the PDF/content) that runs inside FastAPI with a Streamlit frontend.

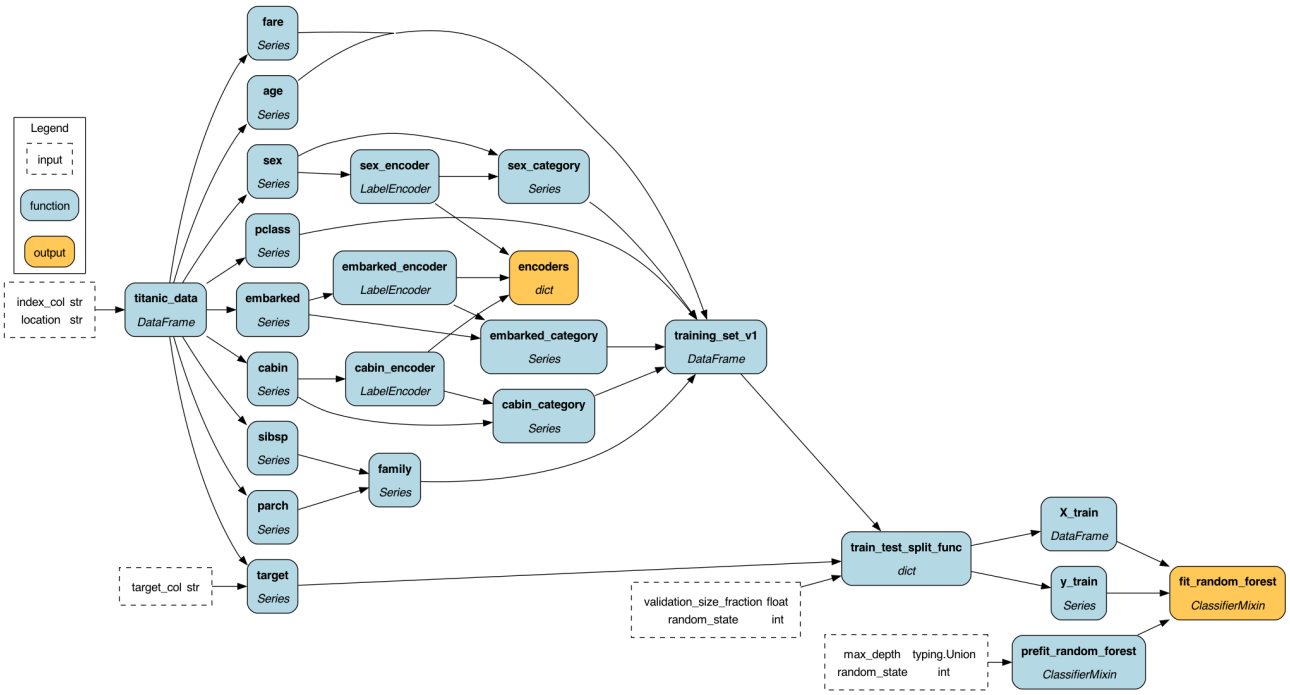
Data quality

Apache Hamilton comes with data quality included out of the box. While you can read more about this in the API reference, we have a few examples to help get you started.

The following two examples showcase a similar workflow, one using the vanilla hamilton data quality decorator, and the other using the pandera integration. The goal of this is to show how to use runtime data quality checks in a larger, more complex ETL.

1. Data quality with hamilton
2. Data quality with pandera

Lineage + Apache Hamilton



Example lineage graph generated by Apache Hamilton when you write Apache Hamilton code.

Here we showcase Apache Hamilton's lineage abilities. We will use the Titanic data set to model a hypothetical company set up where there is data engineering and data science team collaborating together.

If you want to see code and what it does we invite you to jump straight into browsing the lineage_snippets notebook. For those coming from the lineage blog post, you can find the code shown in lineage_script.py.

For those who want to continue, let's first talk about some common problems people encounter, then more formally frame what we mean by lineage, and then explain how Apache Hamilton's lineage capabilities help you solve common problems encountered with Data and Machine Learning.

Note: a quick word on terminology, we use `function` and `node` interchangeably because you write functions, that get turned into nodes that are modeled in a DAG. So if you see `function` or `node` they mean the same thing effectively.

Common Problems (and therefore questions)

As your data and ML work progresses, invariably time passes and someone runs into a problem such as:

- why is my model suddenly behaving badly? What columns does it use and what are its data sources?
- we used to be getting a value for this column/feature, but now we're not. What has changed?
- we ingested some bad data, and we need to know who and what is impacted.
- a person on my team wants to make a change to X, but I'm afraid we're going to break something.
- I have inherited some code running in production, and now something broke, where do I start?
- Governance is asking me for information about data sources to a model, and the work required seems arduous, how can I quickly get this information?
- I'm terrified of inheriting this code base, I don't know what's going on.
- I need to audit that we're in compliance with GDPR, but that's going to take forever.

These are all questions that can be answered with lineage information. Let's talk about what we mean by lineage more concretely.

What is "Lineage"?

In the context of machine learning and data work, "lineage" refers to the historical record or traceability of data, models, and processes. It encompasses the entire life cycle of data, from its origin to its final usage. Lineage helps establish the provenance, quality, and reliability of data and aids in understanding how the data has been transformed.

In the context of machine learning models, lineage provides information about the training data, preprocessing steps, hyperparameters, and algorithms used to train the model. It helps researchers, data scientists, and stakeholders understand how a model was developed and evaluate its reliability and potential biases.

For data pipelines and workflows, lineage tracks the flow of data through different processing steps, transformations, and analyses. It helps identify dependencies, troubleshoot issues, and reproduce results by capturing the sequence of operations performed on the data.

Lineage information is valuable for various reasons, including:

- Reproducibility: Lineage enables the replication of experiments and analyses by recording the exact steps taken, ensuring that results can be reproduced reliably.
- Auditing and Compliance: Lineage provides transparency and accountability, which is crucial for regulatory compliance and ensuring data privacy.
- Troubleshooting and Debugging: Lineage helps identify errors, inconsistencies, or unexpected results by tracing data transformations and model training processes.
- Collaboration: Lineage allows different stakeholders to understand the data's history, facilitating collaboration between teams working on different parts of a project.

Apache Hamilton's Lineage Capabilities

Good news: Apache Hamilton provides a lot of the functionality needed for storing lineage and asking questions of it. Here we'll walk through a few features of Apache Hamilton that will help answer and empower teams targeting the four points above:

- reproducibility
- auditing and compliance
- troubleshooting and debugging
- collaboration

Lineage as Code

To start, Apache Hamilton by design, encodes lineage information as code. This means, as you write each Apache Hamilton function, you are encoding lineage information required to compute it, i.e. by specifying a dataflow you have in effect, specified lineage! This means, as you write your code and commit to, for example a version control system, you have a record of how computation should happen. A huge benefit of this, is when the code changes, so does this information – all without you having to manage a separate system!

TL;DR:

> No need for a separate system to store lineage information, it's already in your code!

The one thing you need to manage is that Apache Hamilton does not store information on artifacts it produces. If you're producing a dataframe, a model, or some other object that your Apache Hamilton code computes, you need to store that. The good news is, you're likely already doing this! **But**, you're probably not storing the lineage information that produced that artifact, and that's where Apache Hamilton comes in.

For each artifact you produce, you just need to associate the Apache Hamilton DAG that produced it. This is as simple as storing the git SHA of the code and snapshot of configuration that created your Apache Hamilton DAG, so you can retrieve the code and ask questions of it. Adding this extra information is easy since most destinations to store artifacts allow for extra metadata to be stored, e.g. from MLFlow for models, Snowflake as table metadata for tables/dataframes, to flat files on S3.

Let's explain how using Apache Hamilton helps get at the four points above.

Reproducibility

By writing Apache Hamilton code and connecting it with a version control system (e.g. git) you have by definition written code that can reproduce results. This is because Apache Hamilton DAGs are deterministic. The version control system is a straightforward way to store evolutions of your code and configuration, and therefore your DAGs.

By versioning code, you are therefore versioning lineage information. This means you can go back in time and ask questions about the past. For example, you can ask what the lineage information was for a model that was trained at a specific point in time. This is as simple as checking out the git SHA of the code that produced the model, and asking Apache Hamilton to visualize (e.g. see `visualize_execution()`), the DAG and ask questions of it.

Auditing and Compliance

The `@tag` and (`@tag_outputs`) feature allows you to annotate your functions with metadata. No extra YAML file to manage, just directly together with your Apache Hamilton code. This means you can tag functions with information such as "PII", "GDPR", "HIPAA", etc. and then ask Apache Hamilton to return nodes with certain tags, e.g. get me all my "sources", or "what is PII, and what consumes it?", etc.

The Apache Hamilton Driver has a lot of functions that allow you to ask questions of your DAGs to make (1) easier. The driver code can be run in a CI/CD system, or as part of a reporting/auditing pipeline. For example, you can ask:

- What are all the functions and their tags via `list_available_variables()`
- What are the possible places that consume the output of this function via `what_is_downstream_of()`
- What are the possible sources that feed into this function via `what_is_upstream_of()`

With these three functions you can the find functions with specific tags and then ask questions in relation to them.

Troubleshooting and Debugging

The good news is that what is great for reproducibility, auditing and reproducibility, is also great for troubleshooting and debugging.

Debugging is methodical and procedural with Apache Hamilton. The way functions are written and executed mean that one can easily walk through just the part of the DAG of interest to debug an issue. To help with this, Apache Hamilton has various methods to visualize lineage so you can more easily see what you're walking through connects to:

- `display_all_functions()`
- `display_downstream_of()`
- `display_upstream_of()`
- `visualize_execution()`
- `visualize_path_between()`

Collaboration

When any organization scales, or has personnel changes, it's important to have a system that helps people get up to in a self-service manner. Apache Hamilton's lineage as code approach means that new team members can easily get up to speed because functions are written in a standard way, and the lineage information is encoded in the code itself. The Apache Hamilton Driver code enables one to ask questions of the DAGs, and therefore the code, to get up to speed quickly.

Recipe for using Apache Hamilton's Lineage Capabilities

At a high level, the recipe for utilizing Apache Hamilton's lineage capabilities is as follows:

1. Write Apache Hamilton code.
2. Use `@tag` and `@tag_outputs` to annotate functions.
3. Instantiate a Apache Hamilton Driver, it'll then have a representation of how data and compute flow as defined by your Apache Hamilton code. The Driver object can then emit/provide information on lineage!
4. If you store Apache Hamilton code with your version control system, you can then go back in time to understand how lineage changes over time, since it's encoded in code!

In code this should look something like the following:

(1) and (2) write Apache Hamilton code and annotate with `@tag` (and/or `@tag_outputs`).

```
@tag(owner="data-science", importance="production", artifact="model")
def fit_random_forest(
    prefit_random_forest: base.ClassifierMixin,
    X_train: pd.DataFrame,
    y_train: pd.Series,
) -> base.ClassifierMixin:
  """Returns a fit RF model."""
  # ... contents of function not important ... code skipped for
brevity
```

(3) Instantiate a Apache Hamilton Driver and ask questions of it.

```
from hamilton import base
from hamilton import driver
import data_loading, features, model_pipeline, sets # import modules
config = {}  # This example has no configuration.
# instantiate the driver
adapter = base.DefaultAdapter()
dr = driver.Driver(config, data_loading, features, sets,
model_pipeline, adapter=adapter)
# ask questions of the driver
# E.g. How do the feature encoders get computed and what flows into
them?
inputs = {
    "location": "data/train.csv",
    "index_col": "passengerid",
    "target_col": "survived",
    "random_state": 42,
    "max_depth": None,
    "validation_size_fraction": 0.33,
}
dr.visualize_execution(
    [features.encoders], "encoder_lineage", {"format": "png"},
inputs=inputs
)
# what is upstream of the fit_random_forest node?
upstream_nodes = dr.what_is_upstream_of("fit_random_forest")
# can now filter the nodes by tags, and pull that information out...

# what is downstream of the titanic_data node?
downstream_nodes = dr.what_is_downstream_of("titanic_data")
# can now filter the nodes by tags, and pull that information out...

# what nodes are PII?
pii_nodes = [n for n in dr.list_available_variables()
             if n.tags.get("PII") == "true"]

# what nodes are called between "age" and "fit_random_forest"?
nodes_in_path = dr.what_is_the_path_between("age",
"fit_random_forest")
```

```
# etc
```

To see more code, we invite you to:

1. Browse the modules to see what the functions are and what they're annotated with.
2. Browse either the lineage_snippets notebook or the lineage_script to see how to use the Apache Hamilton Driver to ask questions of your DAGs.
3. We invite you to then go back in time, by checking out this repository and checking out an older commit and re-running the script or notebook and seeing how things change. The command to "go back in time" would be:

```
# see current lineage
python lineage_script.py
# go back in time
git checkout 7e2e92a79644b904856c0a81b8faa7f1ae00c64e
# see past lineage
python lineage_script.py
# to reset to current lineage
git checkout main
```

A script you could write to ask questions of your DAGs

To help you get programmatic access to your DAGs, we have an example script you could write to quickly get lineage answers. The script is lineage_commands.py. The main point of the script, is to show you that it could encode a runbook for your team, or be used within your CI/CD system to query, visualize, and otherwise emit lineage information.

Scaling computation

Apache Hamilton enables a variety of tools for allowing you to scale your data processing by integrating with third-party libraries.

Specifically, we have four examples that show how to scale Apache Hamilton both by parallelizing transformations (ray and dask) and running on larger, distributed datasets (pandas on spark, pyspark map UDFs).

1. Integrating hamilton with pandas on spark.
2. Integrating hamilton with ray.

3. Integrating hamilton with dask.

4. Integrating hamilton with pyspark.

Microservice

While we've mainly been discussing running Apache Hamilton in a batch environment, it can easily be used in a microservice/online setting. This is valuable if you want insight into exactly how your endpoints transform/load data, or if you want to execute the same transforms you did in batch in an online setting.

The following example shows how to execute an asynchronous set of transforms in a microservice:

We will be releasing feature-specific examples shortly, as well.

https://github.com/apache/hamilton/tree/main/examples/async

Extension autoloading

Under `hamilton.plugins`, there are many modules named `*_extensions` (e.g., `hamilton.plugins.pandas_extensions`, `hamilton.plugins.mlflow_extensions`). They implement Apache Hamilton features for 3rd party libraries, including `@extract_columns`, materializers (`to.parquet`, `from_.mlflow`), and more.

Autoloading behavior

By default, Apache Hamilton attempts to load all extensions one-by-one. This means that as you have more Python packages in your environment (e.g., `pandas`, `pyspark`, `mlflow`, `xgboost`), importing Apache Hamilton appears to become slower because it actually imports many packages.

This behavior can be less desirable when your Apache Hamilton dataflow doesn't use any of these packages, but you need them in your Python environment nonetheless. For example, if only `pandas` is needed for your dataflow, but you have `mlflow` and `xgboost` in your environment their respective extensions will be loaded each time.

Disable autoloading

Disabling extension autoloading allows to import Apache Hamilton without any extensions, which can reduce import time from 2-3 sec to less than 0.5 sec. This speedup is welcomed when you

need to restart a notebook's kernel often or you're operating in a low RAM environment (some Python packages are larger than 50Mbs).

There are three ways to opt-out: programmatically, environment variables, configuration file. You must opt-out before having any other `hamilton` import.

1. Programmatically

```
from hamilton import registry
registry.disable_autoload()
```

2. Environment variables

From the console

```
export HAMILTON_AUTOLOAD_EXTENSIONS=0
```

Programmatically via Python `os.environ`.

```
import os
os.environ["HAMILTON_AUTOLOAD_EXTENSIONS"] = "0"
```

Programmatically in Jupyter notebooks

```
%env HAMILTON_AUTOLOAD_EXTENSIONS=0
```

3. Configuration file

Using the following command disables autoloading via the configuration file `./hamilton.conf`. Apache Hamilton won't autoload extensions anymore (i.e., you won't need to use approach 1 or 2 each time).

```
hamilton-disable-autoload-extensions
```

To revert this configuration use the following command

```
hamilton-enable-autoload-extensions
```

To reenable autoloading in specific files, you can delete the environment variable or use `registry.enable_autoload()` before calling `registry.initialize()`

```
from hamilton import registry
registry.enable_autoload()
registry.initialize()
```

Manually loading extensions

If you disabled autoloading, extensions need to be loaded manually. You should load them before having any other `hamilton` import to avoid hard-to-track bugs. There are two ways.

1. Importing the extension

```
from hamilton.plugins import pandas_extensions, mlflow_extensions
```

2. Registering the extension

This approach has good IDE support via `typing.Literal`

```
from hamilton import registry
registry.load_extensions("mlflow")
```

Wrapping the Driver

The APIs that the Hamilton Driver is built on, are considered internal. So it is possible for you to define your own driver in place of the stock Hamilton Driver, we suggest the following path if you don't like how the current Apache Hamilton Driver interface is designed:

Write a "Wrapper" class that delegates to the Hamilton Driver.

i.e.

```
from hamilton import driver

class MyCustomDriver(object):
    def __init__(self, constructor_arg, ...):
        self.constructor_arg = constructor_arg
        ...
```

```
    # some internal functions specific to your context
    # ...

    def my_execute_function(self, arg1, arg2, ...):
        """What actually calls the Hamilton"""
        dr = driver.Driver(self.constructor_arg, ...)
        df = dr.execute(self.outputs)
        return self.augmetn(df)
```

That way, you can create the right API constructs to invoke Hamilton in your context, and then delegate to the stock Hamilton Driver. By doing so, it will ensure that your code continues to work, since we intend to honor the Hamilton Driver APIs with backwards compatibility as much as possible.

Command line interface

This page covers the Apache Hamilton CLI. It is built directly from the CLI, but note that the command `hamilton --help` always provide the most accurate documentation.

Installation

The dependencies for the Apache Hamilton CLI can be installed via

```
pip install sf-hamilton[cli]
```

You can verify the installation with

```
hamilton --help
```

`hamilton` (global)

Options:

- `--verbose / --no-verbose`: [default: no-verbose]
- `--json-out / --no-json-out`: [default: no-json-out]
- `--install-completion`: Install completion for the current shell.

- `--show-completion`: Show completion for the current shell, to copy it or customize the installation.
- `--help`: Show this message and exit.

Commands:

- `build`: Build a single Driver with MODULES
- `diff`: Diff between the current MODULES and their...
- `version`: Version NODES and DATAFLOW from dataflow...
- `view`: Build and visualize dataflow with MODULES

`hamilton build`

Build a single Driver with MODULES

Usage:

```
$ hamilton build [OPTIONS] MODULES...
```

Arguments:

- `MODULES...`: [required]

Options:

- `--help`: Show this message and exit.

`hamilton diff`

Diff between the current MODULES and their specified GIT_REFERENCE

Usage:

```
$ hamilton diff [OPTIONS] MODULES...
```

Arguments:

- `MODULES...`: [required]

Options:

- `--git-reference TEXT` : [default: HEAD]
- `--view / --no-view` : [default: no-view]
- `--output-file-path PATH` : [default: diff.png]
- `--help` : Show this message and exit.

`hamilton version`

Version NODES and DATAFLOW from dataflow with MODULES

Usage:

```
$ hamilton version [OPTIONS] MODULES...
```

Arguments:

- `MODULES...` : [required]

Options:

- `--help` : Show this message and exit.

`hamilton view`

Build and visualize dataflow with MODULES

Usage:

```
$ hamilton view [OPTIONS] MODULES...
```

Arguments:

- `MODULES...` : [required]

Options:

- `--output-file-path PATH` : [default: ./dag.png]
- `--help` : Show this message and exit.

pre-commit hooks

Use pre-commit hooks for safer Apache Hamilton code changes

This page gives an introduction to pre-commit hooks and how to use custom hooks to validate your Apache Hamilton code.

What are pre-commit hooks?

A pre-commit hook is a script or command that's executed automatically before making a commit. The goal of these hooks is to standardize code formatting and catch erroneous code before being committed. For example, popular hooks include ensuring files have no syntax errors, sorting imports, and normalizing line breaks.

Note that it's different from testing, which focuses on the behavior of the code. You can think of pre-commit hooks as checks and formatting you would do everytime you save a file.

Add pre-commit hooks to your project

Hooks are a mechanism of the `git` version control system. You can find your project's hooks under the `.git/hooks` directory (it might be hidden by default). There should be many files with the `.sample` extension that serve as example scripts.

The preferred way of working with pre-commit hooks is through the prek library. This library allows you to import and configure hooks for your repository with a `.pre-commit-config.yaml` file.

Steps to get started

1. install the prek library

```
pip install prek
```

2. add a `.pre-commit-config.yaml` to your repository

```
# .pre-commit-config.yaml
repos:
    # repository with hook definitions
-   repo: https://github.com/pre-commit/pre-commit-hooks
```

```
    rev: v6.0.0  # release version of the repo
    hooks:  # list of hooks from the repo to include in this project
    -   id: end-of-file-fixer
    -   id: trailing-whitespace
    -   id: check-yaml
        args: ['--unsafe']  # some accept arguments

    # download another repository with hooks
-   repo: https://github.com/psf/black
    rev: 22.10.0
    hooks:
    -   id: black
```

3. install the hooks defined in `.pre-commit-config.yaml`

```
prek install
```

Now, hooks will automatically run on `git commit`

4. to manually run hooks

```
prek run --all-files
```

Custom Apache Hamilton pre-commit hooks

pre-commit hooks are great developer tools, but off-the-shelf solutions aren't aware of the Apache Hamilton framework. Hence, we developed a pre-commit hook to help you author Apache Hamilton dataflows! Under the hood, they leverage the `hamilton` CLI, so if you are unfamiliar with it, feel free to install it and view the `--help` messages.

```
pip install sf-hamilton[cli]
hamilton --help
```

Checking dataflow definition

Apache Hamilton doesn't have many syntactic constraints, but there's a few things we want to catch:

- functions parameters and return are type annotated
- a node consistently has the same type (e.g., a parameter in multiple functions)
- functions with a name starting with underscore (`_`) are ignored from the dataflow

- functions with a `@config` decorator received a trailing double underscore with a suffix (e.g., `hello__weekday()`, `hello__weekend()`)

Instead of reimplementing this logic, we can try to build the Hamilton Driver with the command `hamilton build MODULES` and catch errors. This also ensures the verification is always in sync with the actual build mechanism. This hook will help prevent us from committing invalid dataflow definitions.

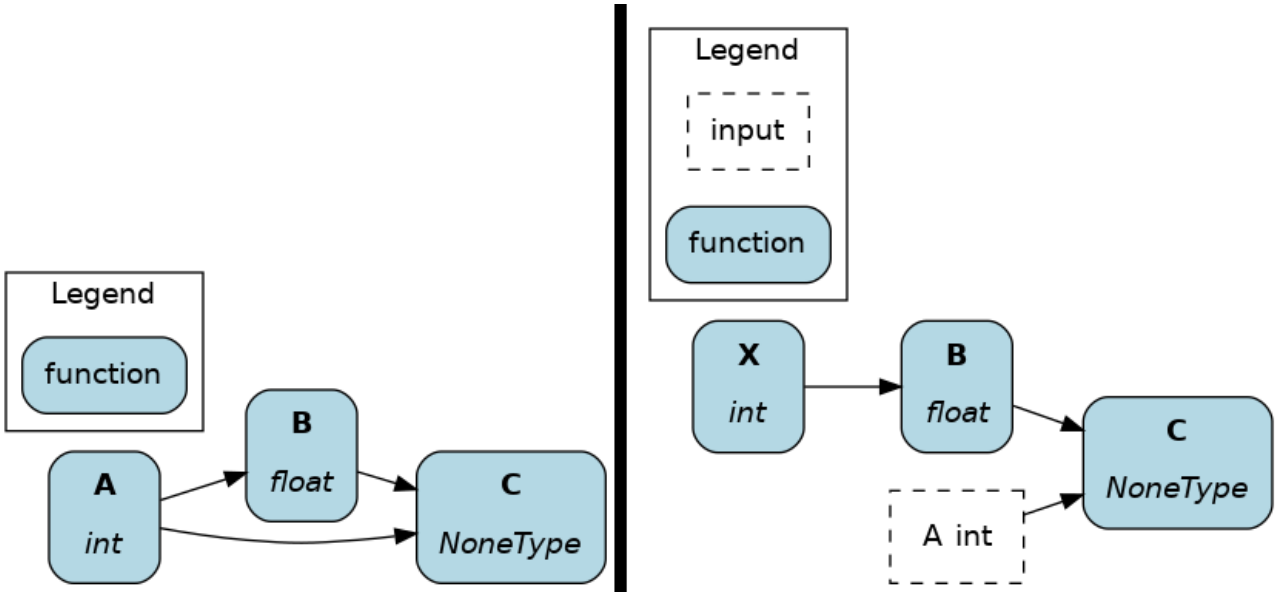
Checking dataflow paths

A dataflow definition might be valid, but it might break paths in unexpected ways. The command `hamilton validate` (which internally uses `Driver.validate_execution()`) can check if a node is reachable.

For example, take a look at `my_module.py`, which contains the nodes `A, B, C`, and the changes between `v1` and `v2`.

```python
# my_module.py - v1
def A() -> int: ...

def B(A: int) -> float: ...

def C(A: int, B: float) -> None: ...

# driver code
dr = driver.Builder().with_modules(my_module).build()
dr.validate_execution(final_vars=["C"]) # <- success
```

```python
# my_module.py - v2
def B(X: int) -> float: ...

def C(A: int, B: float) -> None: ...

# driver code
dr = driver.Builder().with_modules(my_module).build()
dr.validate_execution(final_vars=["C"]) # <- failure. missing `A`
```



In `v1`, the dataflow could be validated for `C` without any inputs. Now, a developer made `B` depend on `X` instead of `A` and removed `A`. This change accidentally impacted `C` which now depends on the external input `A`. Note that both `v1` and `v2` have a valid dataflow definition. To catch breaking changes to the path to `C`, we could use `hamilton validate --context context.json my_module.py` with the context:

```
// context.json
{ "HAMILTON_FINAL_VARS": ["C"] }
// will call .validate_execution(final_vars["C"])
```

Note

pre-commit hooks can prevent commits from breaking a core path, but you should use unit and integration tests for more robust checks.

Add Apache Hamilton pre-commit to your project

We alluded to the relationship between pre-commit hooks and the `hamilton` command line tool. In fact, the basic hook is designed to take a list of `hamilton` commands and will execute them in order when hooks are triggered.

To use them, add this snippet to your `.pre-commit-config.yaml` and adapt it to your project:

```
- repo: https://github.com/dagworks/hamilton-pre-commit
  rev: v0.1.2  # use a ref >= 0.1.2
  hooks:
```

```
  - id: cli-command
    name: Apache Hamilton CLI command
    args: [  # list of CLI commands to execute
      hamilton build my_module.py,
      hamilton build my_module2.py,
      hamilton validate --context context.json my_module.py
my_module2.py,
    ]
```

The above snippet would:

- check the dataflow definition of `my_module.py`
- check the dataflow definition of `my_module2.py`
- validate the execution path specified in `context.json` for dataflow composed of `my_module.py` and `my_module2.py`

You can pass any `hamilton` CLI command to the pre-commit hook, but it will only care about it succeeding or failing.

Apache Hamilton UI

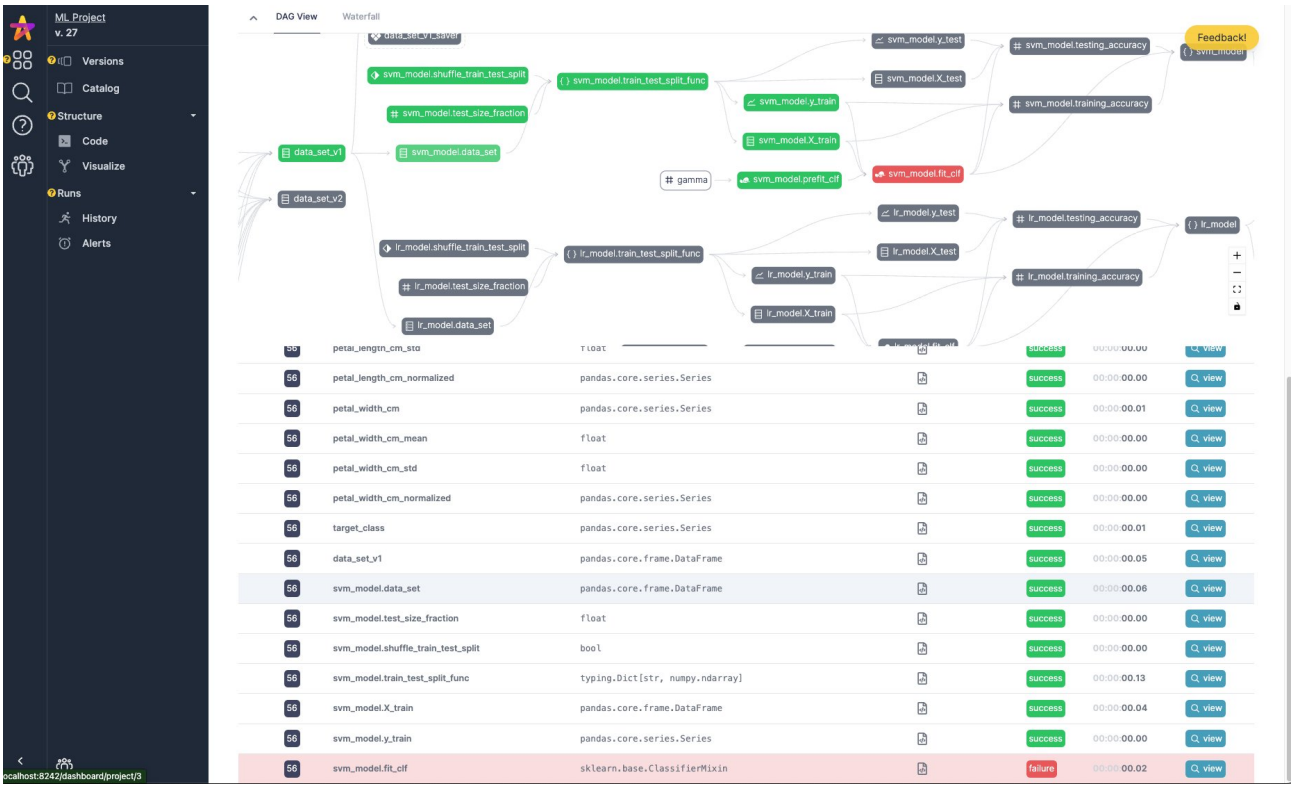
Reference

UI Overview

Apache Hamilton comes with a fully open-source UI that can be run both for local deployment and on a remote server. The UI consists of the following features:

1. Telemetry for hamilton executions – both on the history of executions and the data itself.
2. A feature/artifact catalog for browsing/connecting executions of nodes -> results.
3. A dataflow (i.e. DAG) visualizer for exploring and looking at your code, and determining lineage.
4. A project explorer for viewing curating projects and viewing versions of your Apache Hamilton dataflows.

In short, the Apache Hamilton UI aims to combine a large swath of MLOps/data observability systems in one simple application.



—

The Apache Hamilton UI has two modes: 1. Run locally using sqlite3 2. Run on docker images with postgres (meant for deployment)

Local Mode

To run the hamilton UI in local mode, you can do the following:

```
pip install "sf-hamilton[ui,sdk]"
hamilton ui
# python -m hamilton.cli.__main__ ui # on windows
```

This will launch a browser window in localhost:8241. You can then navigate to the UI and start using it! While this can potentially handle a small production workflow, you may want to run on postgres with a separate frontend/backend/db for full scalability and a multi-read/write db.

Docker/Deployed Mode

The Apache Hamilton UI can be contained within a set of Docker images. You launch with docker-compose, and it will start up the UI, the backend server, and a Postgres database. If you'd like a quick overview of some of the features, you can watch the following:

Note: if you run into the "Invalid HTTP_HOST" error, then please set the environment variable *HAMILTON_ALLOWED_HOSTS="*"* (or comma separated list of domains of choice) for the backend docker container. You can inject this via *-e* or in the *docker-compose[-prod].yml* file itself.

Install

If you'd like a video walkthrough on getting set up, you can watch the following:

As prerequisites, you will need to have Docker installed – you can follow instructions here.

1. Clone the Apache Hamilton repository locally

```
git clone https://github.com/apache/hamilton
```

1. Navigate to the `hamilton/ui` directory

```
cd hamilton/ui
```

1. Execute the installation script with the following command

```
./run.sh
```

This will:

- Pull all Docker images from the Docker Hub
- Start a local Postgres database
- Start the backend server
- Start the frontend server

This takes a bit of time! So be patient. The server will be running on port 8242.

1. Then navigate to `http://localhost:8242` in your browser, and enter your email (this will be the username used within the app).

Building the Docker Images locally

If building the Docker containers from scratch, increase your Docker memory to 10gb or more – you can do this in the Docker Desktop settings.

To build the images locally, you can run the following command:

```
# from the hamilton/ui directory
./dev.sh --build
```

This will build the containers from scratch. If you just want to mount the local code, you can run just

```
./dev.sh
```

Self-Hosting

If you know docker, you should be good to go. The one environment variable to know is *HAMILTON_ALLOWED_HOSTS*, which you can set to * to allow all hosts, or a comma separated list of hosts you want to allow.

To host the UI on a subpath, set *REACT_APP_HAMILTON_SUB_PATH* to the subpath required. For example, to run on *https://domain.com/hamilton*:

```
- REACT_APP_HAMILTON_SUB_PATH=/hamilton
```

Make sure that the sub path environment variable begins with / if set.

Please reach out to us if you want to deploy on your own infrastructure and need help - join slack. More extensive self-hosting documentation is in the works, e.g. Snowflake, Databricks, AWS, GCP, Azure, etc.; we'd love a helm chart contribution!

Running on Snowflake

You can run the Apache Hamilton UI on Snowflake Container Services. For a detailed guide, see the blog post Observability of Python code and application logic with Apache Hamilton UI on Snowflake Container Services by Greg Kantyka and the Apache Hamilton Snowflake Example.

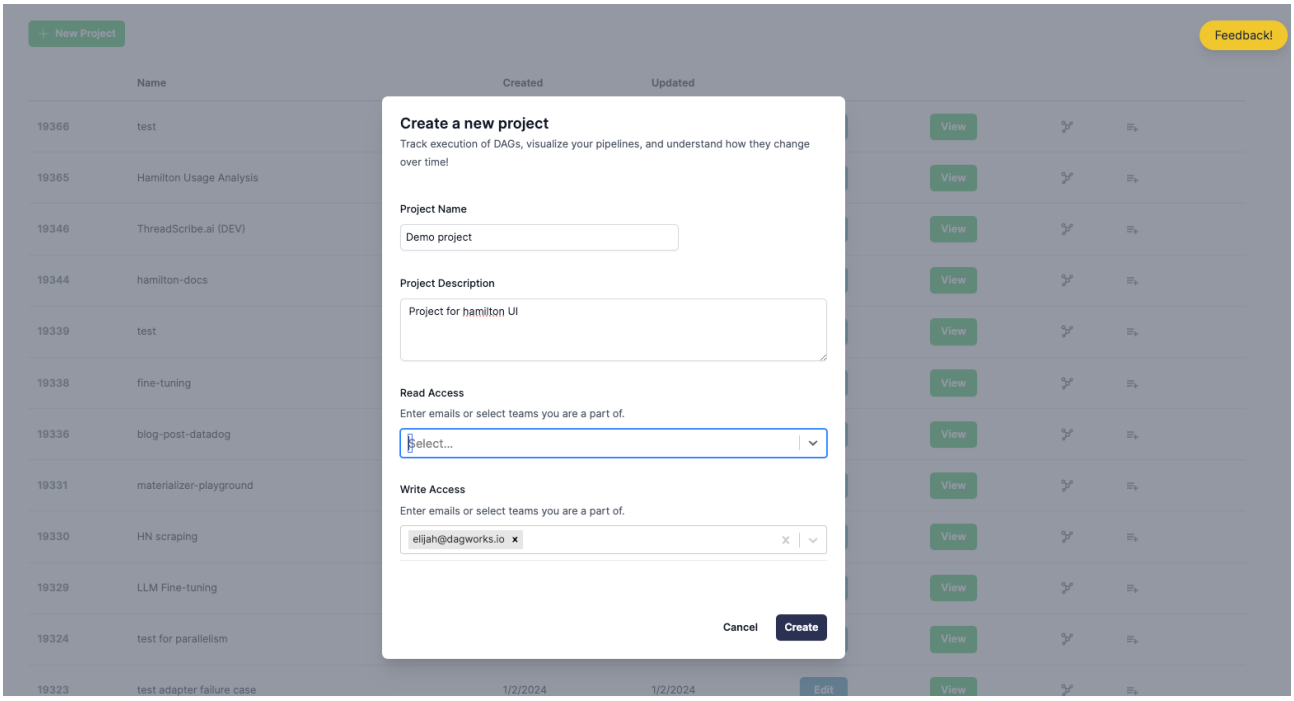
Get started

Now that you have your server running, you can run a simple dataflow and watch it in the UI! You can follow instructions in the UI when you create a new project, or follow the instructions here.

First, install the SDK:

```
pip install "sf-hamilton[sdk]"
```

Then, navigate to the project page (dashboard/projects), in the running UI, and click the green `+ New DAG` button.



Remember the project ID – you'll use it for the next steps.

Existing Apache Hamilton Code

Add the following adapter to your code if you have existing Apache Hamilton code:

```
from hamilton_sdk import adapters

tracker = adapters.HamiltonTracker(
   project_id=PROJECT_ID_FROM_ABOVE,
   username="USERNAME/EMAIL_YOU_PUT_IN_THE_UI",
   dag_name="my_version_of_the_dag",
   tags={"environment": "DEV", "team": "MY_TEAM", "version": "X"}
)

dr = (
  driver.Builder()
    .with_config(your_config)
    .with_modules(*your_modules)
    .with_adapters(tracker)
    .build()
)
```

Then run your DAG, and follow the links in the logs! Note that the link is correct if you're using the local mode – if you're on postgres it links to 8241 (but you'll want to follow it to 8241).

I need some Apache Hamilton code to run

If you don't have Apache Hamilton code to run this with, you can run Apache Hamilton UI example under examples/hamilton_ui:

```
# we assume you're in the Apache Hamilton repository root
cd examples/hamilton_ui
# make sure you have the right python packages installed
pip install -r requirements.txt
# run the pipeline providing the email and project_id you created in
the UI
python run.py --email <email> --project_id <project_id>
```

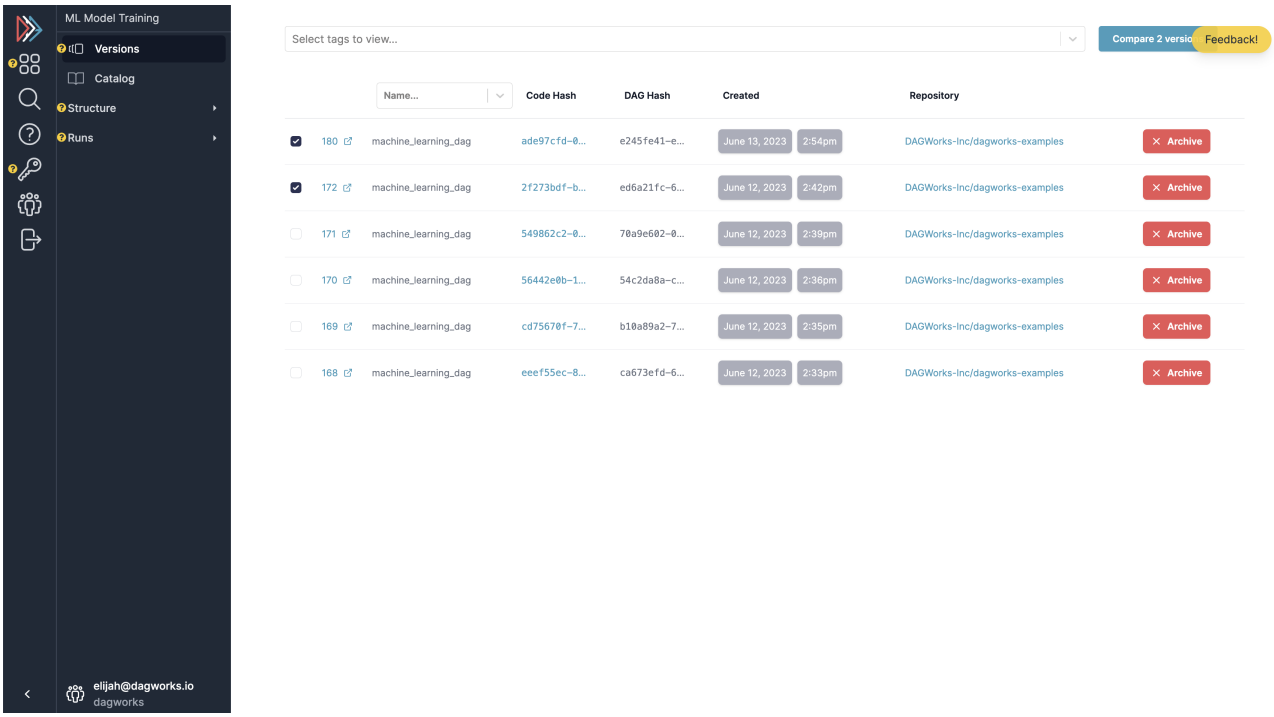
You should see links in the logs to the UI, where you can see the DAG run + the data summaries captured.

Features

Once you get to the UI, you can navigate to the projects page (left hand nav-bar). Assuming you have created a project and logged to it, you can then navigate to view it and then more details about it. E.g. versions, code, lineage, catalog, execution runs. See below for a few screenshots of the UI.

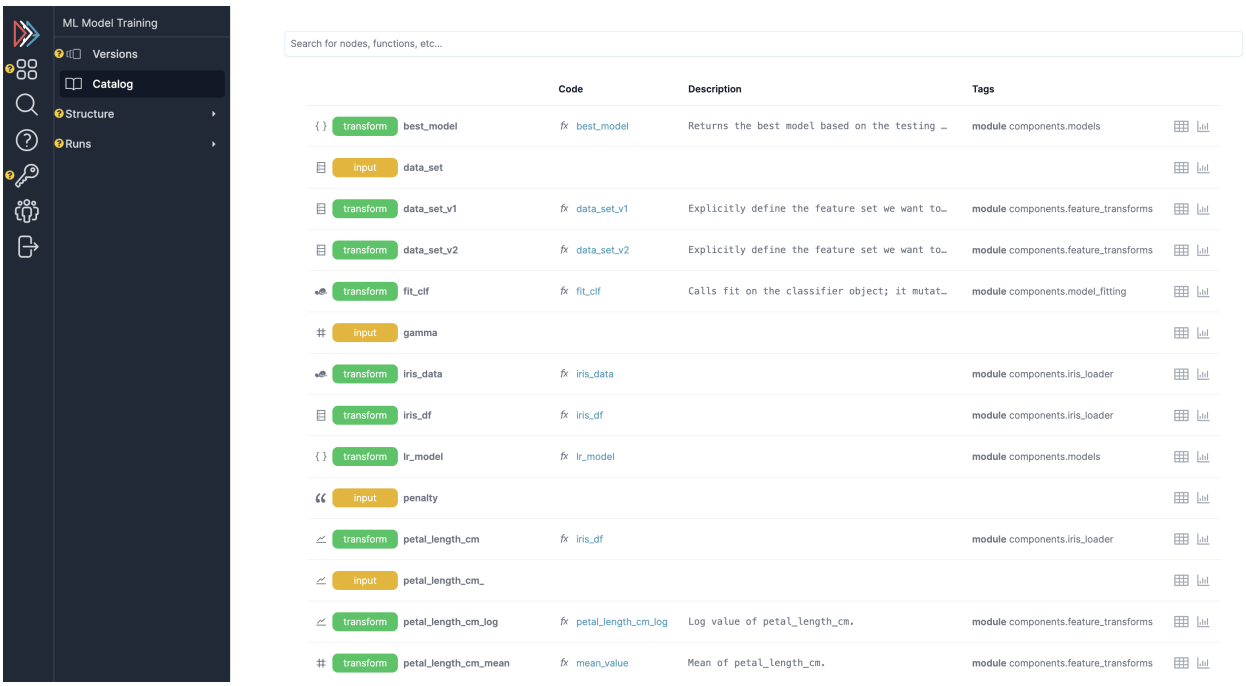
Dataflow versioning

Select a dataflow versions to compare and visualize.



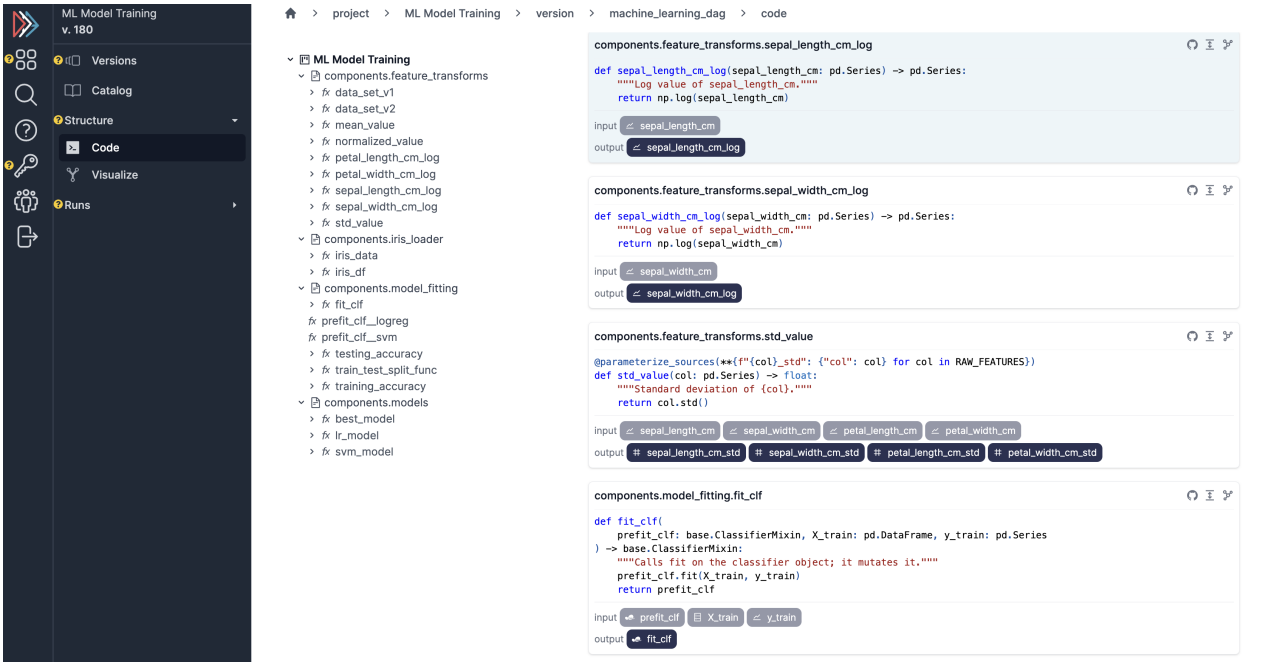
Assets/features catalog

View functions, nodes, and assets across a history of runs.



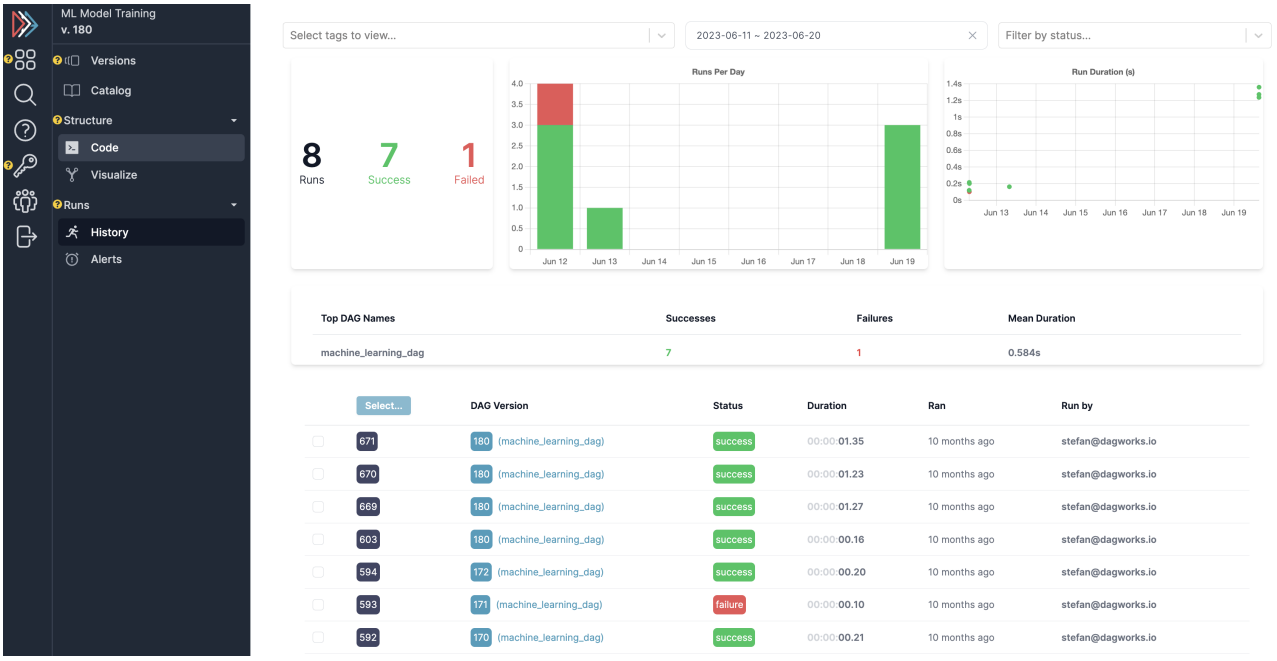
Browser

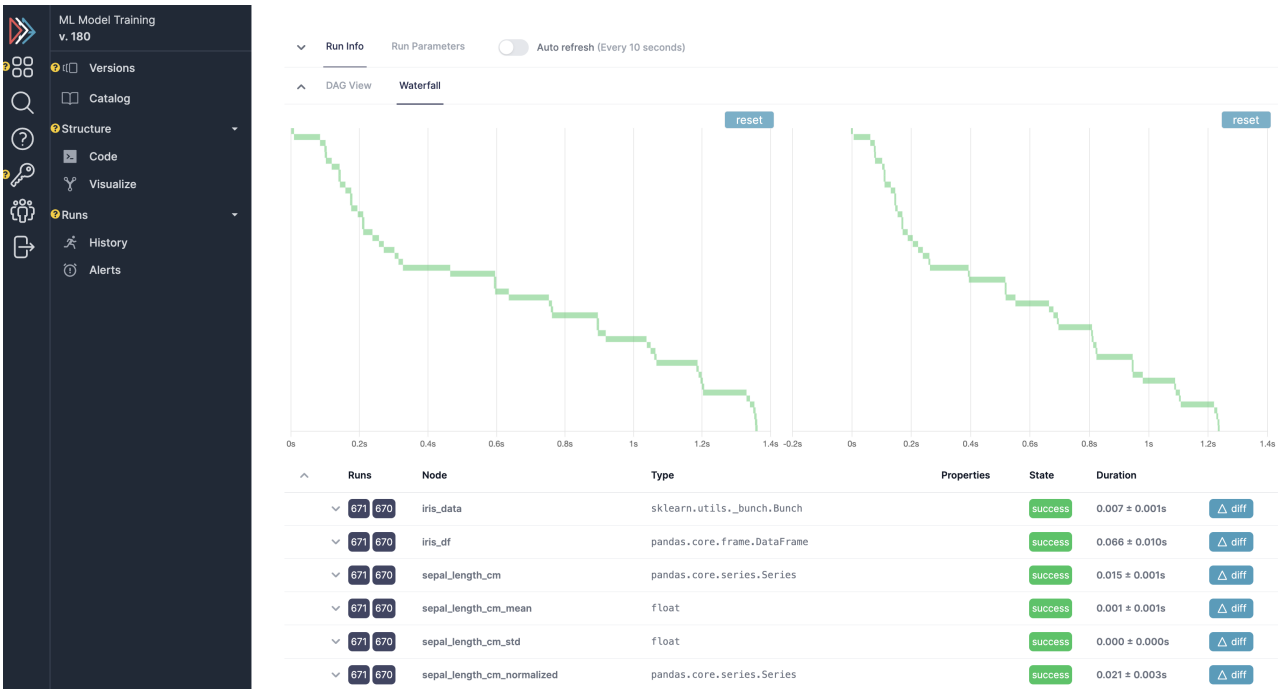
View dataflow structure and code.

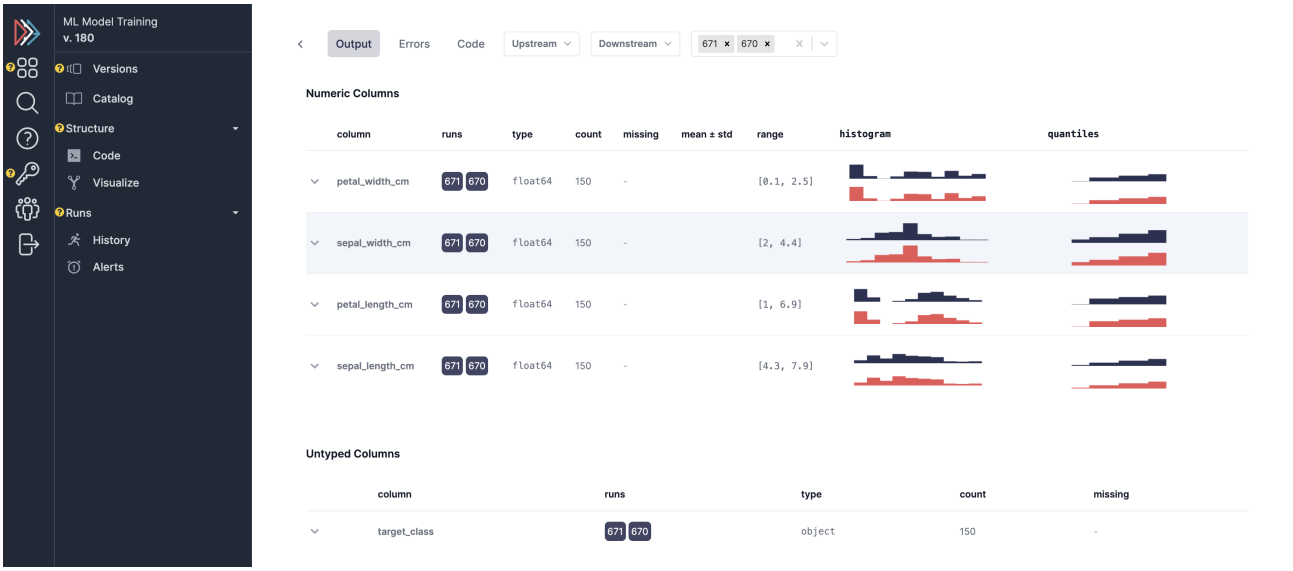


Run tracking + telemetry

View a history of runs, telemetry on runs/comparison, and data for specific runs:


ML Model Training
v. 180
Versions
Catalog
Structure
Code
Visualize
Runs
History
Alerts
Select tags to view...
2023-06-11 ~ 2023-06-20
Filter by status...
8
Runs
7
Success
1
Failed
Runs Per Day
Run Duration (s)
Top DAG Names
Successes
Failures
Mean Duration
machine_learning_dag
7
1
0.584s
Select...
DAG Version
Status
Duration
Ran
Run by
671 180 (machine_learning_dag) success 00:00:01.35 10 months ago stefan@dagworks.io
670 180 (machine_learning_dag) success 00:00:01.23 10 months ago stefan@dagworks.io
669 180 (machine_learning_dag) success 00:00:01.27 10 months ago stefan@dagworks.io
603 180 (machine_learning_dag) success 00:00:00.16 10 months ago stefan@dagworks.io
594 172 (machine_learning_dag) success 00:00:00.20 10 months ago stefan@dagworks.io
593 171 (machine_learning_dag) failure 00:00:00.10 10 months ago stefan@dagworks.io
592 170 (machine_learning_dag) success 00:00:00.21 10 months ago stefan@dagworks.io


ML Model Training
v. 180
Versions
Catalog
Structure
Code
Visualize
Runs
History
Alerts
Run Info
Run Parameters
Auto refresh (Every 10 seconds)
DAG View
Waterfall
reset
reset
Runs
Node
Type
Properties
State
Duration
671 670 iris_data sklearn.utils._bunch.Bunch success 0.007 ± 0.001s △ diff
671 670 iris_df pandas.core.frame.DataFrame success 0.066 ± 0.010s △ diff
671 670 sepal_length_cm pandas.core.series.Series success 0.015 ± 0.001s △ diff
671 670 sepal_length_cm_mean float success 0.001 ± 0.001s △ diff
671 670 sepal_length_cm_std float success 0.000 ± 0.000s △ diff
671 670 sepal_length_cm_normalized pandas.core.series.Series success 0.021 ± 0.003s △ diff



SDK Configuration

This section documents HamiltonTracker configuration options.

Changing where data is sent

You can change where telemetry is logged by passing in *hamilton_api_url* and/or *hamilton_ui_url* to the HamiltonTracker constructor. By default, these are set to *localhost:8241/8242*.

```
from hamilton_sdk import adapters

tracker = adapters.HamiltonTracker(
   project_id=PROJECT_ID_FROM_ABOVE,
   username="USERNAME/EMAIL_YOU_PUT_IN_THE_UI",
   dag_name="my_version_of_the_dag",
   tags={"environment": "DEV", "team": "MY_TEAM", "version": "X"},
   hamilton_api_url="http://YOUR_DOMAIN_HERE:8241",
   hamilton_ui_url="http://YOUR_DOMAIN_HERE:8242" # if using docker
the UI is on 8242.
)

dr = (
  driver.Builder()
    .with_config(your_config)
    .with_modules(*your_modules)
    .with_adapters(tracker)
    .build()
)
```

Changing behavior of what is captured

By default, a lot is captured and sent to the Apache Hamilton UI.

Here are a few options that can change that - these can be found in *hamilton_sdk.tracking.constants*. You can either change the defaults by directly changing the constants, by specifying them in a config file, or via environment variables.

Here we first explain the options:

Simple Invocation

Option	Default	Explanation
CAPTURE_DATA_STATISTICS	True	Whether to capture any data insights/ statistics
MAX_LIST_LENGTH_CAPTURE	50	Max length for list capture
MAX_DICT_LENGTH_CAPTURE	100	Max length for dict capture
DEFAULT_CONFIG_URI	~/.hamilton.conf	Default config file URI.

To change the defaults via a config file, you can do the following:

```
[SDK_CONSTANTS]
MAX_LIST_LENGTH_CAPTURE=100
MAX_DICT_LENGTH_CAPTURE=200

# save this to ~/.hamilton.conf
```

To change the defaults via environment variables, you can do the following, prefixing them with *HAMILTON_*:

```
export HAMILTON_MAX_LIST_LENGTH_CAPTURE=100
export HAMILTON_MAX_DICT_LENGTH_CAPTURE=200
python run_my_hamilton_code.py
```

To change the defaults directly, you can do the following:

```
from hamilton_sdk.tracking import constants

constants.MAX_LIST_LENGTH_CAPTURE = 100
constants.MAX_DICT_LENGTH_CAPTURE = 200

tracker = adapters.HamiltonTracker(
   project_id=PROJECT_ID_FROM_ABOVE,
   username="USERNAME/EMAIL_YOU_PUT_IN_THE_UI",
   dag_name="my_version_of_the_dag",
   tags={"environment": "DEV", "team": "MY_TEAM", "version": "X"}
)

dr = (
  driver.Builder()
    .with_config(your_config)
    .with_modules(*your_modules)
    .with_adapters(tracker)
    .build()
)
dr.execute(...)
```

In terms of precedence, the order is:

1. Module default.

2. Config file values.

3. Environment variables.

4. Directly set values.

IDE extension

Reference

Apache Hamilton VSCode

⚠ Warning

The Apache Hamilton VSCode extension is an experimental feature under active development. Edge cases, evolving features, and partial documentation are to be expected. Please open a GitHub issue or reach out on Slack for troubleshooting!

The Apache Hamilton VSCode extension enables interactive dataflow development in VSCode. This developer productivity tool helps your editor understand how Apache Hamilton works (code completion, symbol navigation, etc.). It is powered by the Apache Hamilton Language Server and can be installed directly from the VSCode marketplace.

Features

Dataflow visualization

Visualize the dataflow defined in the current Python file. As you type and add functions, the visualization automatically updates. There is a UI button to rotate the visualization 90-degree.

Note

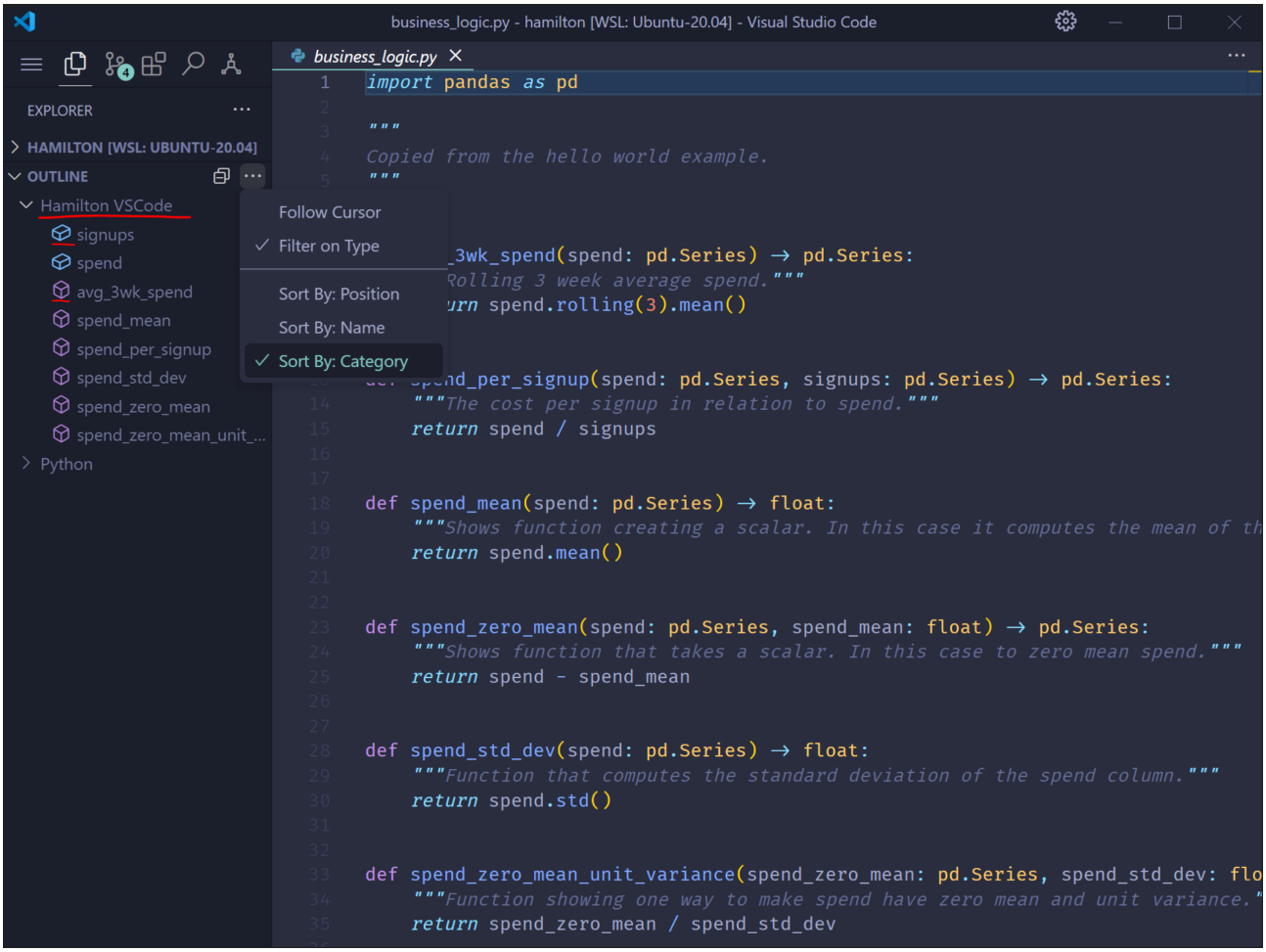
We suggest moving the visualization tab to the VSCode panel (`CTRL+J`) or the secondary sidebar (`CTRL+ALT+B`) by drag-and-dropping the tab.

Completion suggestions

Get completion suggestions when defining new nodes. It will even insert the appropriate type when selected! Completion suggestions have the `Node` type and can even display their docstring when hovered over.

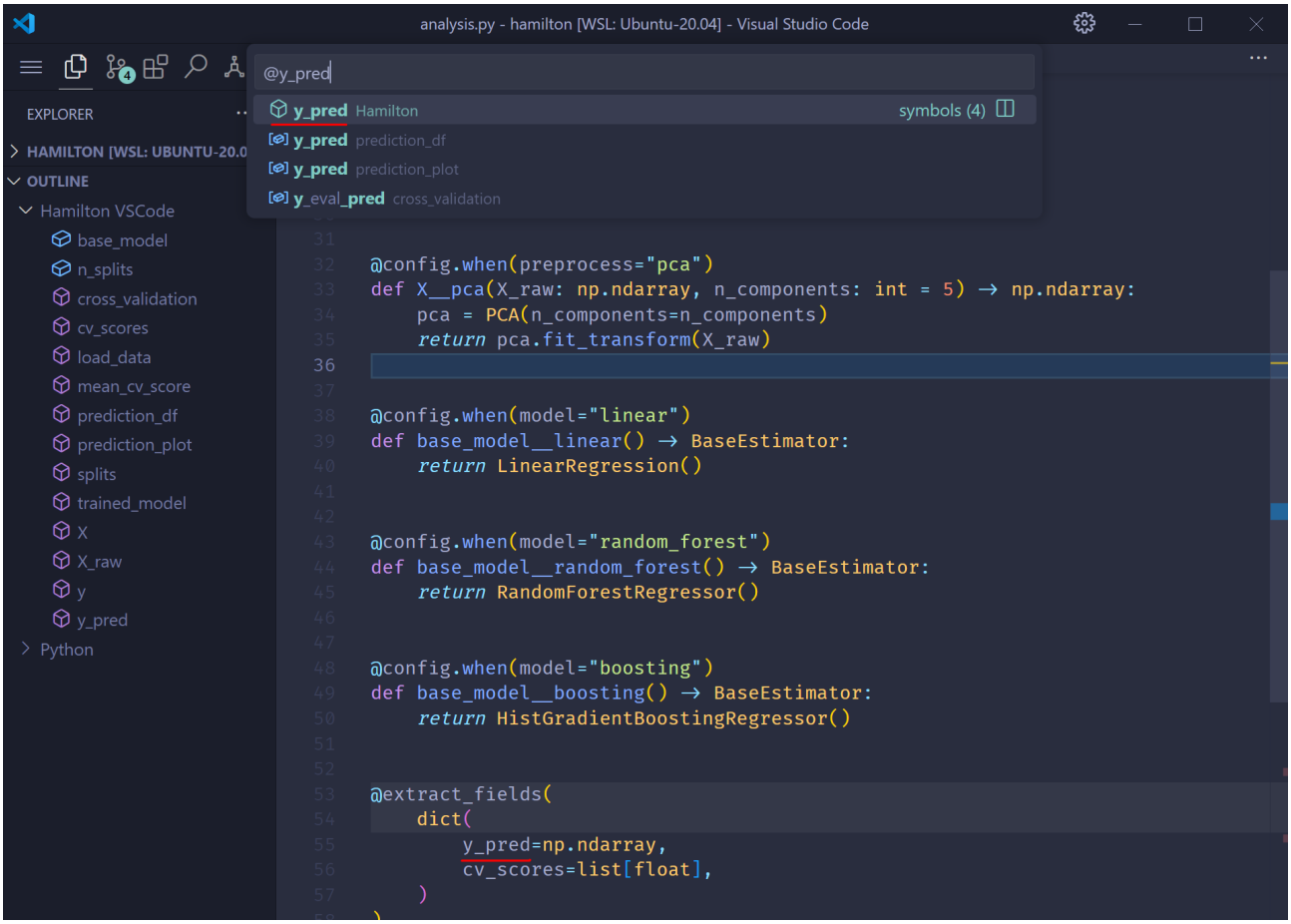
Outline

The **OUTLINE** menu now displays a **Apache Hamilton VSCode** entry. Nodes and inputs from the current Python file are listed and denoted by distinct icons.



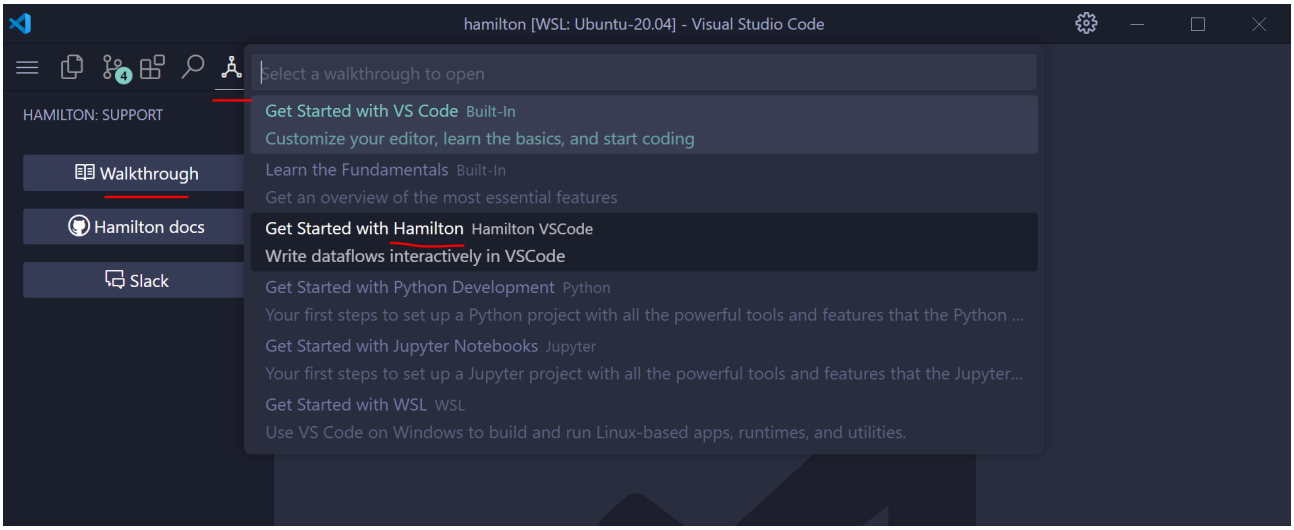
Symbol navigation

When entering symbol navigation (`CTRL+SHIFT+O`), you can jump directly to any node definition. Notice from the screenshot that it even pickups node defined in a decorator.



Extension walkthrough

Under the Apache Hamilton menu (the icon at the top), you can find a list of buttons. Selecting `Walkthrough` and then `Get started with Apache Hamilton` will launch an interactive menu to get you set up along with some tips.



Note

Some of this content may become outdated since the extension is evolving quickly.

Roadmap

There are many features that we'd be interested in implementing. Let us know on Slack your favorite ones!

- Go To Definition: jump to where the node defined
- Go To References: jump to where the node is a dependency
- Rename: rename a node across locations (can be tricky when mentioned in a decorator)
- Support dataflows spanning multiple modules
- Configure the visualization (i.e., match the Python features)
- Integrate with the Apache Hamilton UI (e.g., click a node to open it's Apache Hamilton UI page and see execution details)
- Visualize notebook cells using the Apache Hamilton Jupyter extension (seems possible)

Language Server

⚠ Warning

The Apache Hamilton Language Server is an experimental feature under active development. Edge cases, evolving features, and partial documentation are to be expected. Please open a GitHub issue or reach out on Slack for troubleshooting!

The Apache Hamilton Language Server is an implementation of the Language Server Protocol (LSP). It is designed to power the Apache Hamilton VSCode extension which can be installed directly from the VSCode marketplace.

Language servers power IDE features like completion suggestion, go to definition, collect document symbols, etc. The LSP standard was established to make servers portable across IDE frontends (e.g., VSCode, PyCharm, Emacs). Learn more.

Installation

If you're using the Apache Hamilton VSCode extension, you will prompted to install the language server if it's not found. Simply click the button and it will install it in your current Python interpreter.

You can also manually install the language server in your Python environment via

```
pip install "sf-hamilton[lsp]"
```

Developers

If you want to dig in the internals of the language server and integrate it with another IDE, you can find the source code in the `dev tools/` section of the Apache Hamilton GitHub repository. It is also directly available on PyPi at sf-hamilton-lsp.

Note that the package name is `hamilton_lsp` when used directly via Python code.

Integrations

This section showcases how Apache Hamilton integrates with popular frameworks.

dlt

dlt stands for "data load tool". It's an open-source Python library providing a ton of data `Sources` (Slack, Stripe, Google Analytics, Zendesk, etc.) and `Destinations` (S3, Snowflake, BigQuery, Postgres, etc.). `Pipelines` make it easy to connect `Sources` and `Destinations` and provide advanced engineering features such as table normalization, incremental loading, and automatic schema evolution.

dlt is an "extract and load" tool and Apache Hamilton is a "transform" tool, allowing various usage patterns.

On this page, you'll learn:

- Extract, Transform, Load (ETL)
- Extract, Load, Transform (ELT)
- dlt materializer plugin for Apache Hamilton

Note

See this blog post for a more detailed discussion about ETL with dlt + Apache Hamilton

Extract, Transform, Load (ETL)

The key consideration for ETL is that the data has to move twice:

> ingest raw data (dlt) -> transform (Apache Hamilton) -> store transformed data (dlt)

1. **Extract**: dlt moves the raw data to a processing server
2. **Transform**: on the server, Apache Hamilton executes transformations
3. **Load**: dlt moves the final data to its destination (database, dashboard, etc.)

Pros

- Reduce storage cost: raw data isn't stored
- Data centralization: transformed data is better separated from raw and low quality data

Cons

- Increased latency: data has to move twice
- Reduced flexibility: to try new transformations, data needs to

Extract

1. Create a dlt pipeline for raw data ingestion (see dlt guide).
2. Write the dlt pipeline execution code in `run.py`

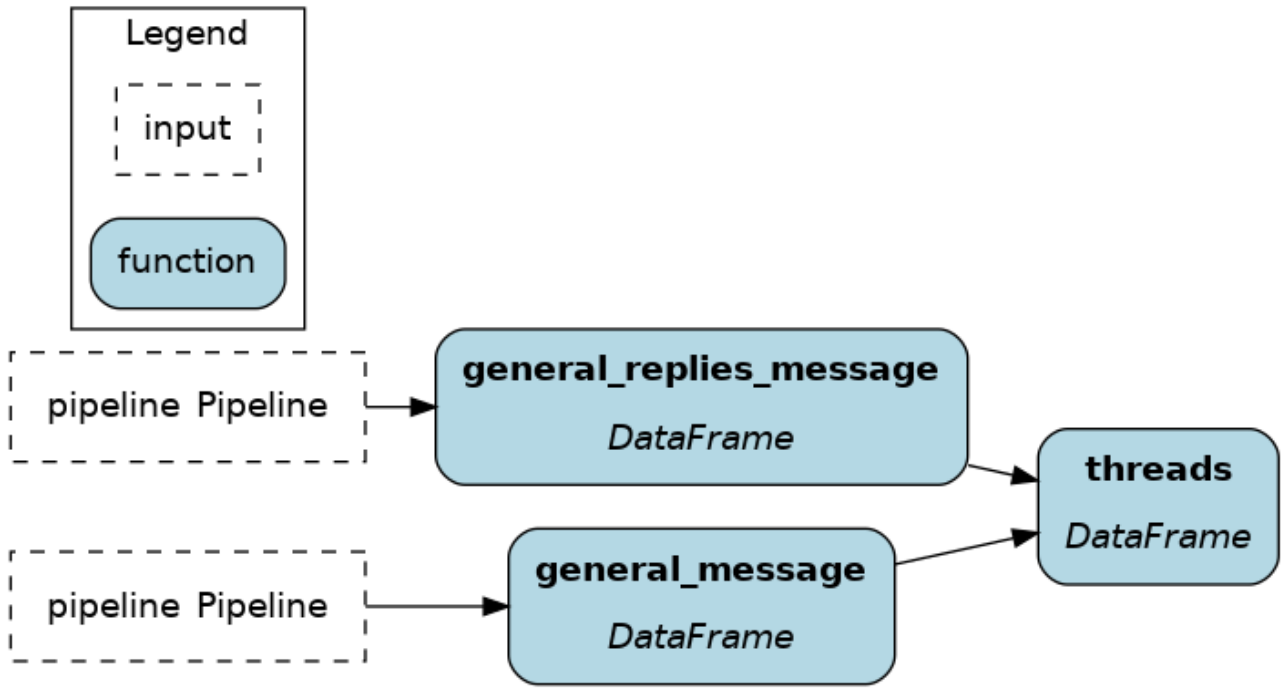
```
# run.py
import dlt
import slack  # NOTE this is dlt code, not an official Slack library

# define dlt pipeline to a local duckdb instance
extract_pipeline = dlt.pipeline(
    pipeline_name="slack_raw",
    destination='duckdb',
    dataset_name="slack_community_backup"
)
# configure dlt slack source
source = slack.slack_source(
    selected_channels=["general"], replies=True
)
# moves data from source to destination
raw_load_info = extract_pipeline.run(source)
```

Transform

1. Define the Apache Hamilton dataflow of transformations

```
# transform.py
import dlt
import pandas as pd

def _table_to_df(client, table_name: str) -> pd.DataFrame:
    """Load data as DataFrame using the dlt SQL client"""
    with client.execute_query("SELECT * FROM %s" % table_name) as t:
        return t.df()
```

```
def general_message(pipeline: dlt.Pipeline) -> pd.DataFrame:
    """Load table `general_message` from dlt data"""
    with pipeline.sql_client() as client:
        return _table_to_df(client, "general_message")

def general_replies_message(pipeline: dlt.Pipeline) -> pd.DataFrame:
    """Load table `general_replies_message` from dlt data"""
    with pipeline.sql_client() as client:
        return _table_to_df(client, "general_replies_message")

def threads(
    general_message: pd.DataFrame,
    general_replies_message: pd.DataFrame,
) -> pd.DataFrame:
    """Reassemble from the union of parent messages and replies"""
    columns = ["thread_ts", "ts", "user", "text"]
    return pd.concat(
        [general_message[columns],
general_replies_message[columns]],
        axis=0
    )
```



1. Add the Apache Hamilton dataflow execution code to `run.py`

```
# run.py
from hamilton import driver
import transform  # module containing dataflow definition

# pass the `transform` module
```

```
dr = driver.Builder().with_modules(transform).build()
# request the node `threads`; pass the dlt `pipeline` as inputs
results = dr.execute(["threads"],
inputs=dict(pipeline=extract_pipeline))
# `results` is a dictionary with key `threads`
```

Load

1. Create a 2nd dlt pipeline to load the transformed data. The `pipeline_name` should be different from the **Extract** step.

```
# run.py
# define dlt pipeline to bigquery (our prod env)
load_pipeline = dlt.pipeline(
    pipeline_name="slack_final",
    destination='bigquery',
    dataset_name="slack_community_backup"
)
# pass the results from Apache Hamilton to dlt
data = results["threads"].to_dict(orient="records")
final_load_info = load_pipeline.run(data, table_name="threads")
```

ETL Summary

You need to set up your dlt pipeline for raw and transformed data, and define your Apache Hamilton transformation dataflow. Then, your execution code consist of executing the ETL step in sequence. It should look like this:

```
# run.py
import dlt
from hamilton import driver
import slack  # NOTE this is dlt code, not an official Slack library
import transform  # module containing dataflow definition

# EXTRACT
extract_pipeline = dlt.pipeline(
    pipeline_name="slack_raw",
    destination='duckdb',
    dataset_name="slack_community_backup"
)
source = slack.slack_source(
    selected_channels=["general"], replies=True
)
raw_load_info = extract_pipeline.run(source)

# TRANSFORM
```

```
dr = driver.Builder().with_modules(transform).build()
results = dr.execute(["threads"],
inputs=dict(pipeline=extract_pipeline))

# LOAD
load_pipeline = dlt.pipeline(
    pipeline_name="slack_final",
    destination='bigquery',
    dataset_name="slack_community_backup"
)
data = results["threads"].to_dict(orient="records")
final_load_info = load_pipeline.run(data, table_name="threads")
```

Extract, Load, Transform (ELT)

Compared to ETL, ELT moves data once.

> ingest and store raw data (dlt) -> transform (Apache Hamilton)

Transformations happen within the data destination, typically a data warehouse. To achieve this, we will leverage the Ibis library, which allows to execute data transformations directly on the destination backend.

1. **Extract & Load**: dlt moves the raw data to the destination

2. **Transform**: Apache Hamilton + Ibis execute transformations within the destination

Pros

- Deduplicate computation: redundant operations can be optimized using raw and intermediary data
- Simpler architecture: no transformation server is needed, unlike ETL

Cons

- Increased storage cost: more space is required to store raw and intermediary data
- Decreased data quality: the sprawl of data of various quality levels needs to be governed

Extract & Load

1. Create a dlt pipeline for raw data ingestion (see dlt guide).

2. Write the dlt pipeline execution code in `run.py`

```
# run.py
import dlt
import slack  # NOTE this is dlt code, not an official Slack library

# define dlt pipeline to duckdb
pipeline = dlt.pipeline(
    pipeline_name="slack",
    destination='duckdb',
    dataset_name="slack_community_backup"
)
# load dlt slack source
source = slack.slack_source(
    selected_channels=["general"], replies=True
)
# execute dlt pipeline
load_info = pipeline.run(source)
```

Transform

1. Define a dataflow of transformations using Apache Hamilton + Ibis

```
# transform.py
import ibis
import ibis.expr.types as ir

def db_con(pipeline: dlt.Pipeline) -> ibis.BaseBackend:
    backend = ibis.connect(f"{pipeline.pipeline_name}.duckdb")
    ibis.set_backend(backend)
    return backend

def general_message(db_con: ibis.BaseBackend, pipeline:
dlt.Pipeline) -> ir.Table:
    """Load table `general_message` from dlt data"""
    return db_con.table(
        "general_message",
        schema=pipeline.dataset_name,
        database=pipeline.pipeline_name
    ).mutate(
        thread_ts=ibis._.thread_ts.cast(str),
        ts=ibis._.ts.cast(str),
    )

def general_replies_message(db_con: ibis.BaseBackend, pipeline:
dlt.Pipeline) -> ir.Table:
    """Load table `general_replies_message` from dlt data"""
    return db_con.table(
```

```
        "general_replies_message",
        schema=pipeline.dataset_name,
        database=pipeline.pipeline_name
    )

def threads(
    general_message: ir.Table,
    general_replies_message: ir.Table,
) -> ir.Table:
    """Create the union of `general_message` and
`general_replies_message`"""
    columns = ["thread_ts", "ts", "user", "text"]
    return ibis.union(
        general_message.select(columns),
        general_replies_message.select(columns),
    )

def insert_threads(threads: ir.Table) -> bool:
    db_con = ibis.get_backend()  # retrieves the backend set in
`db_con()`
    db_con.create_table("threads", threads)
    return True
```

2. Execute the Apache Hamilton dataflow to trigger transformations on the backend

```
# run.py
# hamilton transform
from hamilton import driver
import transform  # module containing dataflow definition

dr = driver.Builder().with_modules(transform).build()
dr.execute(
    ["insert_threads"],  # execute node `insert_threads`
    inputs=dict(pipeline=pipeline)  # pass the dlt pipeline
)
```

ELT Summary

You need to set up your dlt pipeline for raw, and define your Apache Hamilton transformation dataflow. Then, your execution code consist of using dlt to move data to the backend and Apache Hamilton + Ibis to execute transformations.

```
# run.py
import dlt
from hamilton import driver
import slack  # NOTE this is dlt code, not an official Slack library
import transform  # module containing dataflow definition
```

```
# EXTRACT & LOAD
pipeline = dlt.pipeline(
    pipeline_name="slack",
    destination='duckdb',
    dataset_name="slack_community_backup"
)
source = slack.slack_source(
    selected_channels=["general"], replies=True
)
load_info = pipeline.run(source)

# TRANSFORM
dr = driver.Builder().with_modules(transform).build()
results = dr.execute(
    ["insert_threads"],  # query the `threads` node
    inputs=dict(pipeline=pipeline)  # pass the dlt load info
)
```

dlt materializer plugin

We added custom Data Loader/Saver to plug dlt with Apache Hamilton. Compared to the previous approach, it allows to include the dlt operations as part of the Apache Hamilton dataflow and improve lineage / visibility.

Note

See this notebook for a demo.

DataLoader

The `DataLoader` allows to read in-memory data from a `dlt.Resource`. When working with `dlt.Source`, you can access individual `dlt.Resource` with `source.resource["source_name"]`. This removes the need to write utility functions to read data from dlt (with pandas or Ibis). Contrary to the previous ETL and ELT examples, this approach is useful when you don't want to store the dlt Source data. It effectively connects dlt to Apache Hamilton to enable "Extract, Transform" (ET).

```
# run.py
from hamilton import driver
from hamilton.io.materialization import from_
import slack  # NOTE this is dlt code, not an official Slack library
import transform

source = slack.source(selected_channels=["general"], replies=True)
```

```
dr = driver.Builder().with_modules(transform).build()

materializers = [
    from_.dlt(
        target="general_message",  # node name assigned to the data
        resource=source.resources["general_message"]
    ),
    from_.dlt(
        target="general_replies_message",
        resource=source.resources["general_replies_message"]
    ),
]
# when using only loaders (i.e., `from_`), you need to specify
# `additional_vars` to compute, like you would in
`.execute(final_vars=["threads"])`
dr.materialize(*materializers, additional_vars=["threads"])
```

DataSaver

The `DataSaver` allows to write node results to any `dlt.Destination`. You'll need to define a `dlt.Pipeline` with the desired `dlt.Destination` and you can specify arguments for the `pipeline.run()` behavior (e.g., incremental loading, primary key, load_file_format). This provides a "Transform, Load" (TL) connector from Apache Hamilton to dlt.

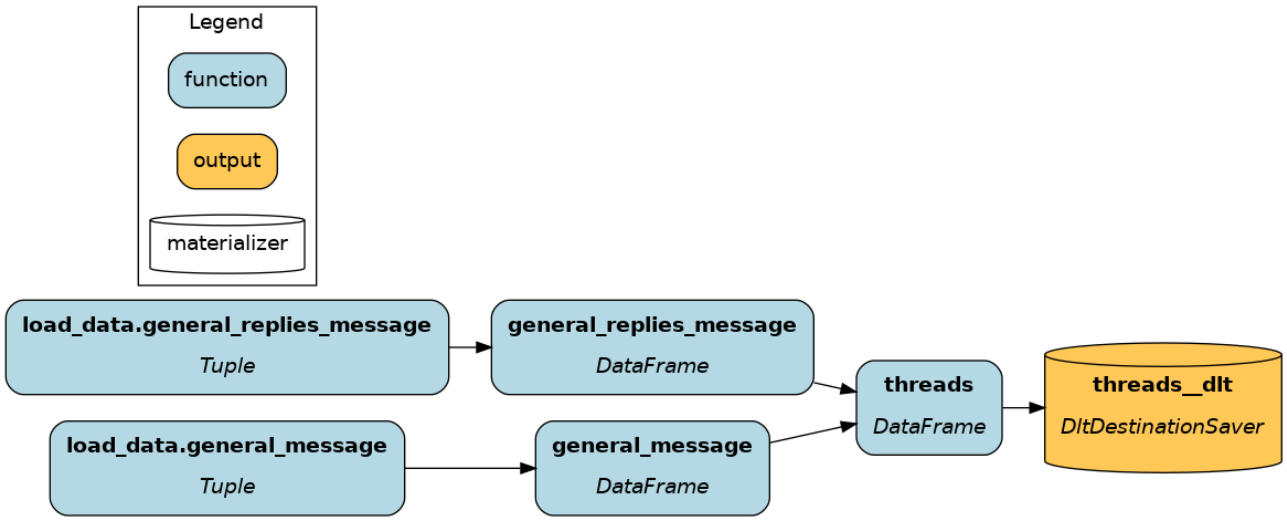
```
# run.py
import dlt
from hamilton import driver
from hamilton.io.materialization import to
import slack  # NOTE this is dlt code, not an official Slack library
import transform

pipeline = dlt.pipeline(
    pipeline_name="slack",
    destination='duckdb',
    dataset_name="slack_community_backup"
)

dr = driver.Builder().with_modules(transform).build()

materializers = [
    to.dlt(
        id="threads__dlt",  # node name
        dependencies=["threads"],
        table_name="slack_threads",
        pipeline=pipeline,
    )
]
```

```
dr.materialize(*materializers)
```

Combining both

You can also combine both the `DataLoader` and `DataSaver`. You will see below that it's almost identical to the ELT example, but now all operations are part of the Apache Hamilton dataflow!

```
# run.py
import dlt
from hamilton import driver
from hamilton.io.materialization import from_, to
import slack  # NOTE this is dlt code, not an official Slack library
import transform

pipeline = dlt.pipeline(
    pipeline_name="slack",
    destination='duckdb',
    dataset_name="slack_community_backup"
)
source = slack.source(selected_channels=["general"], replies=True)

dr = driver.Builder().with_modules(transform).build()

materializers = [
    from_.dlt(
        target="general_message",
        resource=source.resources["general_message"]
    ),
    from_.dlt(
        target="general_replies_message",
        resource=source.resources["general_replies_message"]
    ),
    to.dlt(
        id="threads__dlt",
        dependencies=["threads"],
        table_name="slack_threads",
        pipeline=pipeline,
    )
]

dr.materialize(*materializers)
```



Next steps

- Our full code example to ingest Slack data and generate thread summaries is available on GitHub.
- Another important pattern in data engineering is reverse ETL, which consists of moving data analytics back to your sources (CRM, Hubspot, Zendesk, etc.). See this dlt blog to get started.

FastAPI

FastAPI is an open-source Python web framework to create APIs. It is a modern alternative to Flask and a more lightweight option than Django. In FastAPI, endpoints are defined using Python functions. The parameters indicate the request specifications and the return value specifies the response. Decorators are used to specify the HTTP methods (GET, POST, etc.) and to route the request.

```python
from typing import Union
from fastapi import FastAPI

app = FastAPI()  # Instantiate the FastAPI server

@app.get("/")  # GET method with base route "/"
def read_root():
    return {"Hello": "World"}

@app.get("/items/{item_id}")  # dynamic route with variable `item_id`
def read_item(item_id: int, q: Union[str, None] = None):
    return {"item_id": item_id, "q": q}
```

```
if __name__ == "__main__":
    # launch the server with `uvicorn`
    import uvicorn
    uvicorn.run(app, host="0.0.0.0", port=8000)  # specify host and
port
```

On this page, you'll learn how Apache Hamilton can help you:

- Test you application
- Reduce the friction from proof-of-concept to production
- Document your API

Challenges

1. Test your FastAPI application

FastAPI endpoints are simply decorated Python function, allowing a great deal of flexibility as to what is executed (functions, classes, web requests, etc.). On one hand, we want to test that endpoints are defined and behave properly by starting a server and testing the GET, POST, etc. requests. FastAPI provides great documentation and tooling to do so. On the other hand, these tests conflate the role of the FastAPI server and the endpoint behavior. To run them, a server-client pair need to be created, which will slow down your test suite, and endpoints need to be mocked to avoid connecting to a production environment. By coupling the role of the FastAPI server and the endpoint behavior, more efforts and resources are needed to write and run tests. The content of the endpoints and the structure of your codebase might make it difficult to test endpoint logic outside the context of a FastAPI server.

2. Document your API

FastAPI already does a great job at automating API documentation by integrating with Swagger UI and OpenAPI. It leverages the endpoints' name, path, docstring, and type annotations, and also allows to add descriptions and example inputs. However, since docstrings, descriptions, and example inputs are not directly tied to the code, they risk becoming out of sync as changes are made.

Apache Hamilton + FastAPI

Adding Apache Hamilton to your FastAPI server can provide a better separation between the dataflow and the API endpoints. Each endpoint can use `Driver.execute()` to request variables and wrap results into an HTTP response. Then, data transformations and interactions with

resources (e.g., database, web service) are curated into standalone Python modules and decoupled from the server code.

Since Apache Hamilton dataflows will run the same way inside or outside FastAPI, you can write simpler unit tests for Hamilton functions without defining a mock server and client. Additionnally, visualizations for the defined Apache Hamilton dataflows can be added to the FastAPI Swagger UI documentation. They will remain in sync with the API behavior because they are generated from the code.

Example

In this example, we'll build a backend for a PDF summarizer application.

> The full code can be found on GitHub

Client

The client defines an HTTP POST request to send a PDF file along a selected OpenAI GPT model, the content type of the PDF file, and a query for the summarization. The `files` parameter allows for multipart encoding uploads and `data` sets the content of the body of the request.

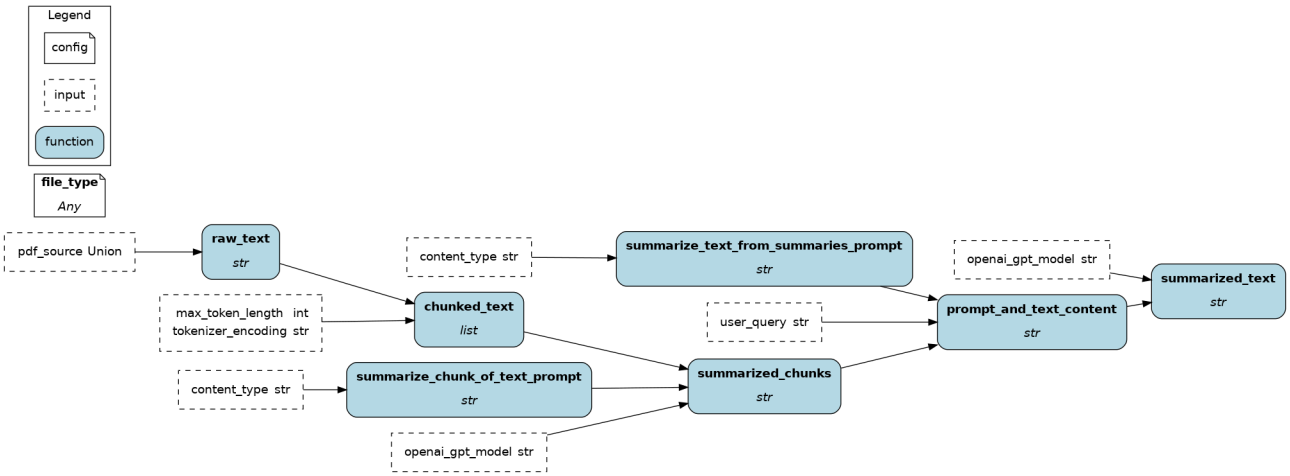
```
# client.py
from typing import IO
import requests

def post_summarize(
    uploaded_pdf: IO[bytes],
    openai_gpt_model: str,
    content_type: str,
    user_query: str,
) -> requests.Response:
    """POST request to summarize a PDF via the `/summarize`
endpoint"""
    return requests.post(
        url="http://0.0.0.0:8000/summarize",  # http://HOST:PORT/
ENDPOINT as specified in server.py
        files=dict(pdf_file=uploaded_pdf),
        data=dict(
            openai_gpt_model=openai_gpt_model,
            content_type=content_type,
            user_query=user_query,
        ),
    )
```

> 💡 For more complex FastAPI applications, you can automatically generate the client code in Python and other languages (TypeScript, Rust, etc.)

Backend dataflow with Apache Hamilton

Apache Hamilton transformations are defined in the module `summarization.py`. This includes loading and chunking the raw text, summarizing chunks with the OpenAI API, and reducing chunks into a final summary.

Visualization of the Apache Hamilton dataflow



Server definition with FastAPI

Then, the FastAPI server is defined in `server.py`. Notice a few things:

- the `Driver` is built only once in the global context.
- the endpoint types are set using `Annotated[...]` to accept multipart encoded forms
- the HTTP POST request is passed as `inputs` to `Driver.execute()`
- the Apache Hamilton results are wrapped into a Pydantic `SummarizeResponse` model

```python
# server.py
from typing import Annotated

from fastapi import FastAPI, Form, UploadFile
from pydantic import BaseModel
from hamilton import driver

import summarization

app = FastAPI()

# build the Hamilton Driver with the summarization module
dr = (
    driver.Builder()
    .with_modules(summarization)
    .build()
)
```

```
class SummarizeResponse(BaseModel):
    """Response to the /summarize endpoint"""
    summary: str

@app.post("/summarize")  # POST request, `/summarize` endpoint
def summarize_pdf(
    pdf_file: Annotated[UploadFile, Form()],
    openai_gpt_model: Annotated[str, Form()],
    content_type: Annotated[str, Form()],
    user_query: Annotated[str, Form()],
) -> SummarizeResponse:
    """Summarize the text from the PDF file"""
    results = dr.execute(
        ["summarized_text"],
        inputs=dict(
            pdf_source=pdf_file.file,
            openai_gpt_model=openai_gpt_model,
            content_type=content_type,
            user_query=user_query,
        ),
    )
    return SummarizeResponse(summary=results["summarized_text"])

if __name__ == "__main__":
    import uvicorn
    uvicorn.run(app, host="0.0.0.0", port=8000)  # specify host and
port
```

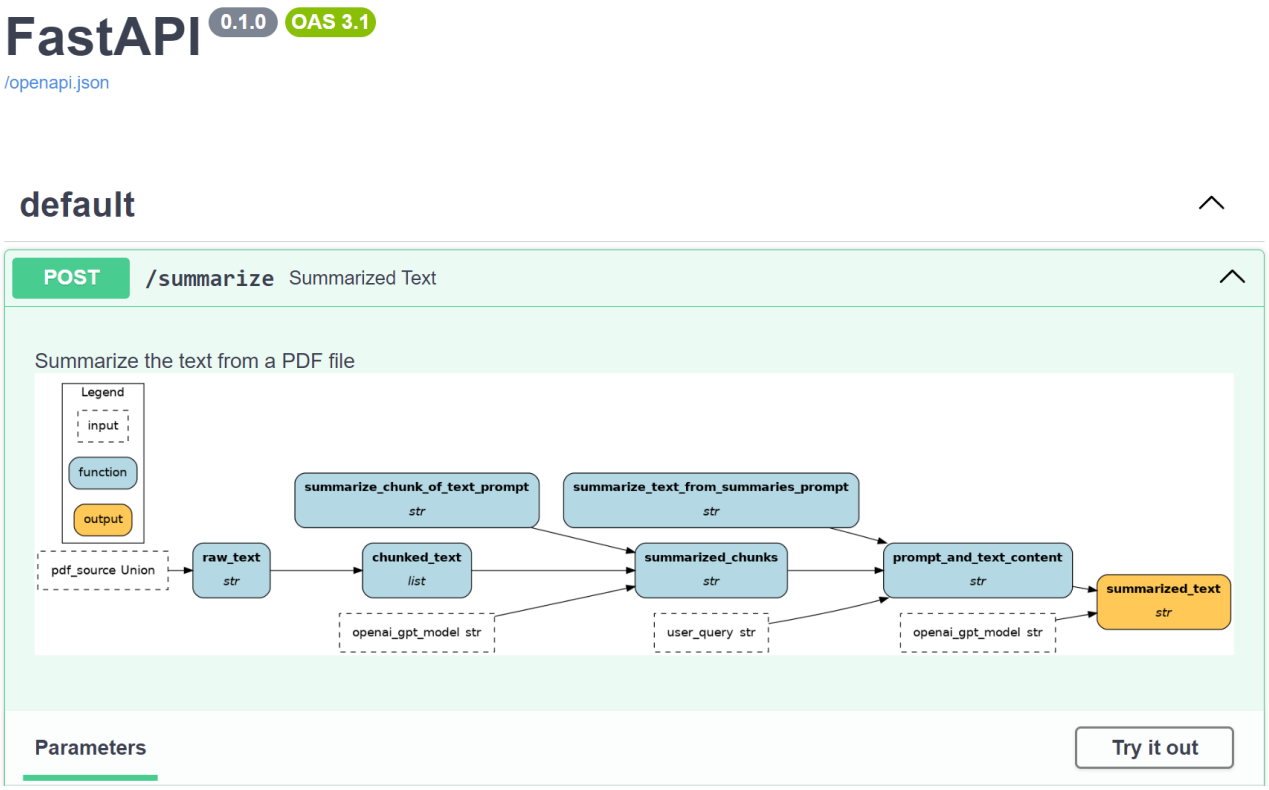
Visualize endpoints' dataflow

The Apache Hamilton dataflow visualizations can be added to the automatically generated FastAPI Swagger UI documentation, which can be viewed at http://0.0.0.0:8000/docs

```
# server.py
# ... after defining all endpoints

# get the visualization
visualization = dr.visualize_execution(["summarized_text"],
inputs=dict(pdf_source=bytes(), openai_gpt_model="", user_query=""))
# encode the PNG object into a base64 string
base64_viz =
base64.b64encode(visualization.pipe(format="png")).decode("utf-8")
# route[-1] is the last defined, i.e. `/summarize`
# append the base64 string of a PNG to the API endpoint text
description
app.routes[-1].description += f"""<img src="
{base64_viz}"/"""
```

```
# ... before `if __name__ == "__main__":`
```

📞 If you are interested in a generic approach to add visualizations to all of your endpoints, please reach out to us on Slack!



Benefits

- **Separation of concerns**: the decoupling between `server.py` and `summarization.py` makes it easier to extend and test the server separately from the data transformations.
- **Reusable code**: the module `summarization.py` can be reused elsewhere with Apache Hamilton. For instance, if you first started by building a proof-of-concept with Streamlit + Apache Hamilton, the logic you produced could be reused to power your FastAPI server.
- **Richer documentation**: Apache Hamilton allows to view and better understand the dataflow of an operation. This helps onboard new API users and greatly facilitates transferring the ownership of the API to other engineers.

Ibis

Ibis is the portable Python dataframe library. It allows you to define data transformations once and execute them in multiple backends (BigQuery, DuckDB, PySpark, SQLite, Snowflake, Postgres, Polars; see the full list). If you never used Ibis before, it should feel similar to SQL with a touch of dataframes (e.g., pandas). You'll be primarily writing expressions (similar to an SQL query), which compute values only after calling for execution via `.to_pandas()` for example.

On this page, you'll learn how Ibis + Apache Hamilton can help:

- Create a modular codebase for better collaboration and maintainability
- Reduce the development-production gap

Standalone Ibis

Here's an Ibis code snippet to load data from a CSV, compute features, and select columns / filter rows. It illustrates typical feature engineering operations.

Reading the code, you'll notice that:

- We use "expression chaining", meaning there's a series of `.method()` attached one after another.
- The variable `ibis._` is a special character referring to the current expression e.g., `ibis._.pet` accesseses the column "pet" of the current table.
- The table method `.mutate(col1=, col2=, ...)` assigns new columns or overwrites existing ones.

```
import ibis

raw_data_path = ...
feature_selection = [
    "id", "has_children", "has_pet", "is_summer_brazil",
    "service_time", "seasons", "disciplinary_failure",
    "absenteeism_time_in_hours",
],

# write the expression
feature_set = (
    ibis.read_csv(sources=raw_data_path, table_name="absenteism")
    .rename("snake_case")
    .mutate(
        has_children=ibis.ifelse(ibis._.son > 0, 1, 0),
        has_pet=ibis.ifelse(ibis._.pet > 0, 1, 0),
```

```
        is_summer_brazil=ibis._.month_of_absence.isin([1, 2,
12]).cast(int),
    )
    .select(*feature_selection)
    .filter(ibis.ifelse(ibis._.has_pet == 1, True, False))
)
# execute the expression
feature_df = feature_set.to_pandas()
```

Challenge 1 - Maintain and test large data transformations codebases

Ibis has an SQL-like syntax and supports chaining operations, allowing for powerful queries in a few lines of code. Conversely, there's a risk of sprawling complexity as expressions as statements are appended, making them harder to test and debug. Preventing this issue requires a lot of upfront discipline and refactoring.

Challenge 2 - Orchestrate Ibis code in production

Ibis alleviates a major pain point by enabling data transformations to work across backends. However, moving from dev to prod still requires some code changes such as changing backend connectors, swapping unsupported operators, adding some orchestration and logging execution. This is outside the scope of the Ibis project and is expected to be enabled by other means.

How Apache Hamilton complements Ibis

Write modular Ibis code

Apache Hamilton was initially developed to structure pandas code for a large catalog of features, and has been adopted by multiple organizations since. Its syntax encourages users to chunk code into meaningful and reusable components, which facilitates documentation, unit testing, code reviews, and improves iteration speed. These benefits directly translate to organizing Ibis code.

Now, we'll refactor the above code to use Apache Hamilton. Users have the flexibility to chunk code at the table or the column-level depending on the needed granularity. This modularity is particularly beneficial to Ibis because:

- Well-scoped functions with type annotations and docstring are easier to understand for new Ibis users and facilitate onboarding.
- Unit testing and data validation becomes easier with smaller expressions. These checks become more important when working across backends since the operation coverage varies and bugs may arise.

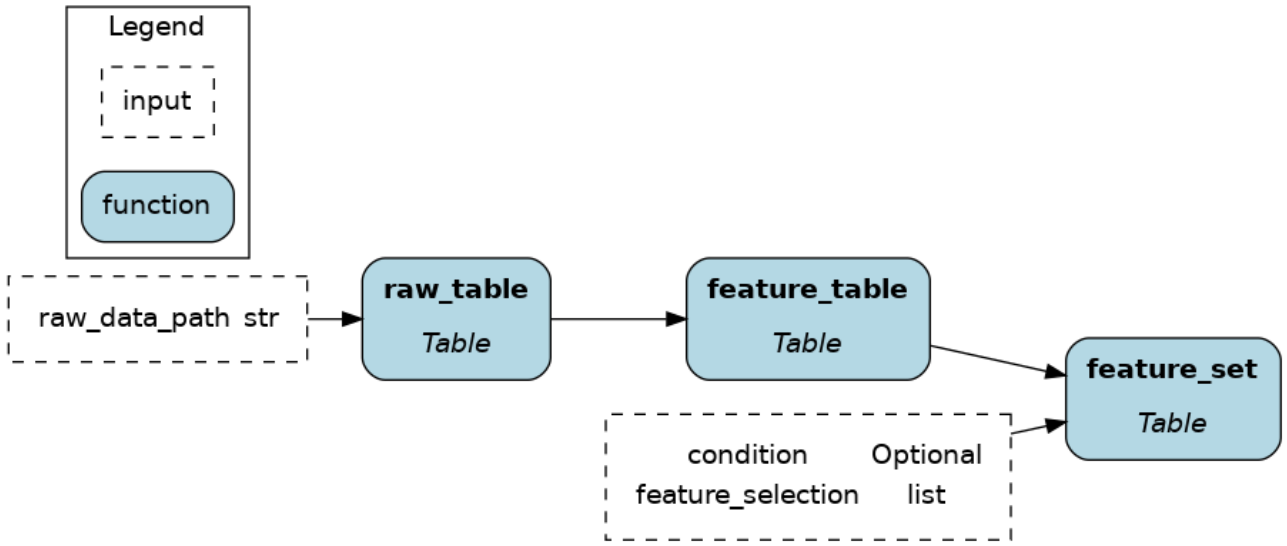
Table-level

Table-level operations might feel most familiar to SQL and Spark users. Also, Ibis + Apache Hamilton is reminiscent of dbt for the Python ecosystem.

Working with tables is very efficient when your number of columns/features is limited, and you don't need full lineage. As you want to reuse components, you can progressively breakdown "table-level code" in to "column-level code".

The initial Ibis code is now 3 functions with type annotations and docstrings. We have a clear sense of the expected external outputs and we could implement schema checks between functions.

```
import ibis
import ibis.expr.types as ir

def raw_table(raw_data_path: str) -> ir.Table:
    """Load CSV from `raw_data_path` into a Table expression
    and format column names to snakecase
    """
    return (
        ibis.read_csv(sources=raw_data_path, table_name="absenteism")
        .rename("snake_case")
    )

def feature_table(raw_table: ir.Table) -> ir.Table:
    """Add to `raw_table` the feature columns `has_children`
    `has_pet`, and `is_summer_brazil`
    """
    return raw_table.mutate(
        has_children=(ibis.ifelse(ibis._.son > 0, True, False)),
        has_pet=ibis.ifelse(ibis._.pet > 0, True, False),
        is_summer_brazil=ibis._.month_of_absence.isin([1, 2, 12]),
    )

def feature_set(
    feature_table: ir.Table,
    feature_selection: list[str],
    condition: Optional[ibis.common.deferred.Deferred] = None,
) -> ir.Table:
    """Select feature columns and filter rows"""
    return feature_table[feature_selection].filter(condition)
```



Column-level

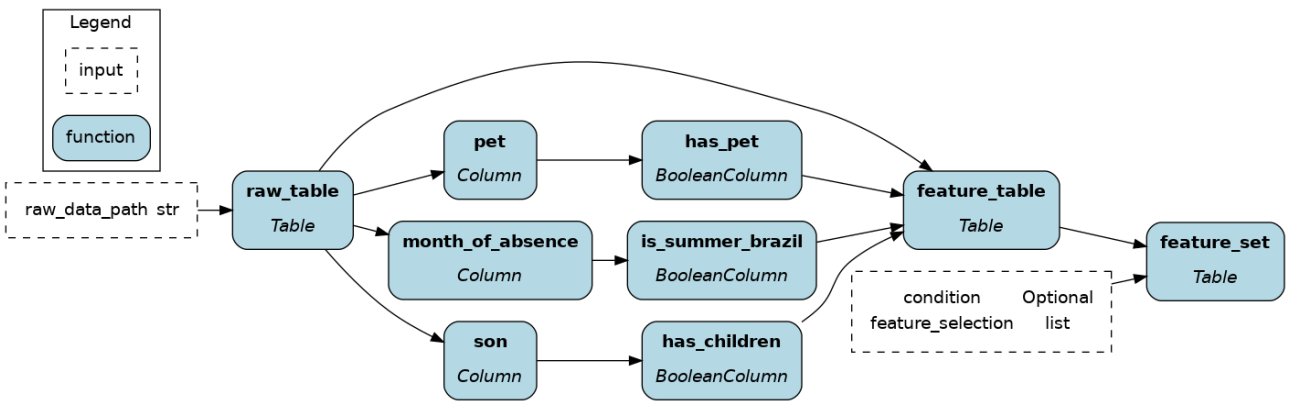
Apache Hamilton was built around column-level operations, which is most common in dataframe libraries (pandas, Dask, polars).

Column-level code leads to fully-reusable feature definitions and a great level of lineage. Notably, this allows to trace sensitive data and evaluate downstream impacts of code changes. However, it is more verbose to get started with, but remember that code is more often read than written.

Now, the `raw_table` is loaded and the columns `son`, `pet`, and `month_of_absence` are extracted to engineer new features. After transformations, features are joined with `raw_table` to create `feature_table`.

```
import ibis
import ibis.expr.types as ir
from hamilton.function_modifiers import extract_columns
from hamilton.plugins import ibis_extensions

# extract specific columns from the table
@extract_columns("son", "pet", "month_of_absence")
def raw_table(raw_data_path: str) -> ir.Table:
    """Load the CSV found at `raw_data_path` into a Table expression
    and format columns to snakecase
    """
    return (
        ibis.read_csv(sources=raw_data_path, table_name="absenteism")
        .rename("snake_case")
    )

# accesses a single column from `raw_table`
def has_children(son: ir.Column) -> ir.BooleanColumn:
    """True if someone has any children"""
    return ibis.ifelse(son > 0, True, False)
```

```python
# narrows the return type from `ir.Column` to `ir.BooleanColumn`
def has_pet(pet: ir.Column) -> ir.BooleanColumn:
    """True if someone has any pets"""
    return ibis.ifelse(pet > 0, True, False).cast(bool)

# typing and docstring provides business context to features
def is_summer_brazil(month_of_absence: ir.Column) ->
ir.BooleanColumn:
    """True if it is summer in Brazil during this month

    People in the northern hemisphere are likely to take vacations
    to warm places when it's cold locally
    """
    return month_of_absence.isin([1, 2, 12])

def feature_table(
    raw_table: ir.Table,
    has_children: ir.BooleanColumn,
    has_pet: ir.BooleanColumn,
    is_summer_brazil: ir.BooleanColumn,
) -> ir.Table:
    """Join computed features to the `raw_data` table"""
    return raw_table.mutate(
        has_children=has_children,
        has_pet=has_pet,
        is_summer_brazil=is_summer_brazil,
    )

def feature_set(
    feature_table: ir.Table,
    feature_selection: list[str],
    condition: Optional[ibis.common.deferred.Deferred] = None,
) -> ir.Table:
    """Select feature columns and filter rows"""
    return feature_table[feature_selection].filter(condition)
```



Note

If your code is already structured with Apache Hamilton, migrating from pandas to Ibis should be easy!

Orchestrate Ibis anywhere

Apache Hamilton is ideal orchestrate for Ibis because it has the lightest footprint and will run anywhere Python does (script, notebook, FastAPI, pyodide, etc.) In fact, the Apache Hamilton library only has 4 dependencies. You don't need "framework code" to get started, just plain Python functions. When moving to production, Apache Hamilton has all the necessary features to complement Ibis such as swapping components, configurations, and lifecycle hooks for logging, alerting, and telemetry.

A simple usage pattern of Apache Hamilton + Ibis is to use the `@config.when` function modifier. In the following example, we have alternative implementations for the backend connection, which will be used for computing and storing results. When running your code, specify in your config `backend="duckdb"` or `backend="bigquery"` to swap between the two.

```
# ibis_dataflow.py
import ibis
import ibis.expr.types as ir
from hamilton.function_modifiers import config

# ... entire dataflow definition

@config.when(backend="duckdb")
def backend_connection__duckdb(
    connection_string: str
) -> ibis.backends.BaseBackend:
    """Connect to DuckDB backend"""
    return ibis.duckdb.connect(connection_string)

@config.when(backend="bigquery")
def backend_connection__bigquery(
    project_id: str,
    dataset_id: str,
) -> ibis.backends.BaseBackend:
    """Connect to BigQuery backend
    Install dependencies via `pip install ibis-framework[bigquery]`
    """
    return ibis.bigquery.connect(
        project_id=project_id,
        dataset_id=dataset_id,
    )
```

```
def insert_results(
    backend_connection: ibis.backends.BaseBackend,
    result_table: ir.Table,
    table_name: str
):
    """Execute expression and insert results"""
    backend_connection.insert(
        table_name=table_name,P
        obj=result_table
    )
```

Note

A potential architecture for Ibis + Apache Hamilton would be running CRON jobs on GitHub actions to periodically launch AWS Lambda with the Apache Hamilton code to orchestrate Ibis data transformations directly on the backend. This has the potential to save meaningful cloud egress cost and greatly diminishes orchestration complexity.

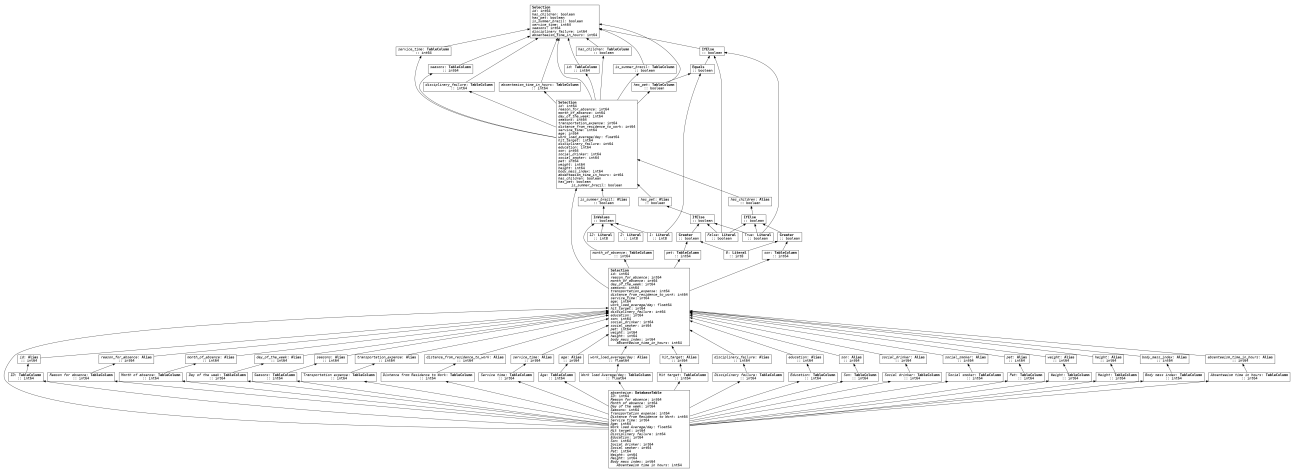
How Ibis complements Apache Hamilton

Performance boost

Leveraging DuckDB as the default backend, Apache Hamilton users migrating to Ibis should immediately find performance improvements both for local dev and production. In addition, the portability of Ibis has the potential to greatly reduce development time.

Atomic data transformation documentation

Apache Hamilton can directly produce a dataflow visualization from code, helping with project documentation. Ibis pushes this one step further by providing a detailed view of the query plan and schemas. See this Ibis visualization for the column-level Apache Hamilton dataflow defined above. It includes all renaming, type casting, and transformations steps (Please open the image in a new tab and zoom in 🔎).



Working across rows with user-defined functions (UDFs)

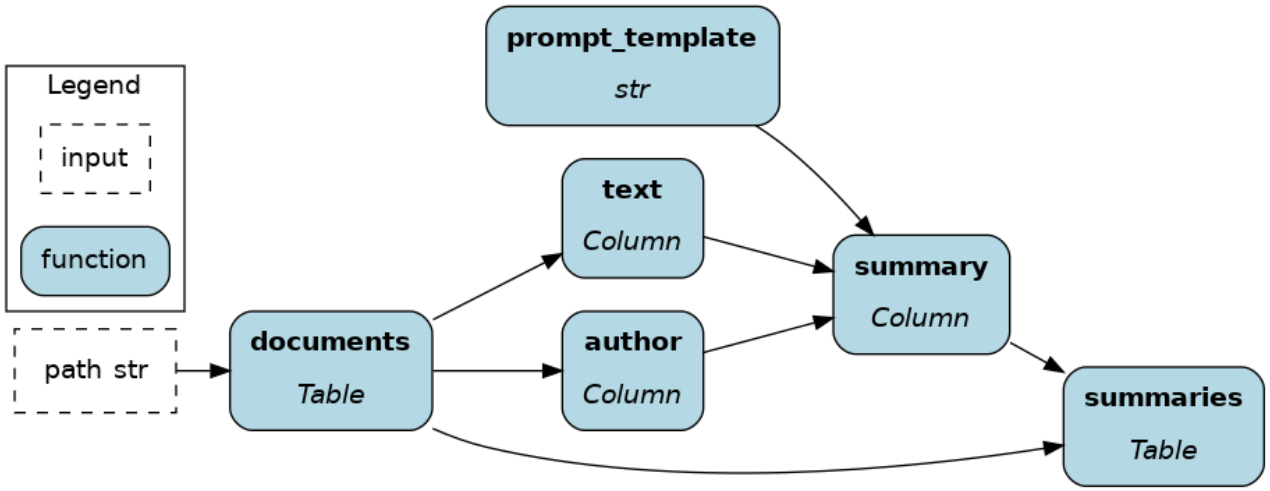
Apache Hamilton and most backends are designed to work primarily on tables and columns, but sometimes you'd like to operate over a row (think of `pd.DataFrame.apply()`). However, pivoting tables is costly and manually iterating over rows to collect values and create a new column is quickly inconvenient. By using scalar user-defined functions (UDFs), Ibis makes it possible to execute arbitrary Python code on rows directly on the backend.

> **Note**
>
> Using `@ibis.udf.scalar.python` creates a non-vectorized function that iterates row-by-row. See the docs to use backend-specific UDFs with `@ibis.udf.scalar.builtin` and create vectorized scalar UDFs.

For instance, you could embed rows of a text column using an LLM API without leaving the datawarehouse.

```
import ibis
import ibis.expr.types as ir

def documents(path: str) -> ir.Table:
    """load text documents from file"""
    return ibis.read_parquet(sources=path, table_name="documents")

# the function name would need to start
# with `_` to avoid being added as a node
@ibis.udf.scalar.python
def _generate_summary(author: str, text: str, prompt_template: str) -
> str:
    """UDF Function to call the OpenAI API line by line"""
    prompt = prompt_template.format(author=author, text=text)
    client = openai.OpenAI(...)
```

```
    try:
        response = client.chat.completions.create(...)
        return_value = response.choices[0].message.content
    except Exception:
        return_value = ""
    return return_value


def prompt_template() -> str:
    return """summarize the following text from {author} and add
    contextual notes based on it biography and other written work

    TEXT
    {text}
    """

def summaries(documents: ir.Table, prompt_template: str) -> ir.Table
    """Compute the UDF against the family"""
    return documents.mutate(
        summary=_generated_summary(
            _.author,
            _.text,
            prompt_template=prompt_template
        )
    )
```



Ibis + Apache Hamilton - a natural pairing

- **What works in dev works in prod**: Ibis and Apache Hamilton allows you to write and structure code data transformations portable across backends for small and big data alike. The two being lightweight libraries, installing dependencies on remote workers is fast and you're unlikely to ever encounter dependency conflicts.

- **Maintainable and testable code**: Modular functions facilitates writing high quality code and promotes reusability, compounding your engineering efforts. It becomes easier for new users to contribute to a dataflow and pull requests are merged faster.
- **Greater visibility**: With Apache Hamilton and Ibis, you have incredible visualizations directly derived from your code. This is a superpower for documentation, allowing users to make sense of a dataflow, and also reason about changes.

Streamlit

Streamlit is an open-source Python library to create web applications with minimal effort. It's an effective solution to create simple dashboards, interactive data visualizations, and proof-of-concepts for data science, machine learning, and LLM applications. On this page, you'll learn how Apache Hamilton can help you:

- Write cleaner Streamlit applications
- Reduce friction transition between proof-of-concept and production
- Improve Streamlit performance

Challenges

1. Hard to read UI and data flows.

Complex Streamlit applications become difficult to debug because of the complex flow of operations. In the simplest case, all the code is under the main function call `app()` and components are added to the UI from top to bottom. Components can be nested under `columns`, `containers`, `expanders`, `tabs`, and more to organize them on the page by using the `with` Python syntax. A good coding practice is to separate data transformations from UI, but Streamlit can blur these lines. Things are further complicated when components are added or updated outside the main scope `app()`. As user interactions, data transformations, state, and UI layout become difficult to trace, risks of breaking changes increase and debugging is more challenging.

```
import streamlit as st

# external function writing component
def greeting(name: str) -> None:
    st.write(f"Hello {name}")

def app():
    st.title("Apache Hamilton + Streamlit 🐱🚀")
```

```
    main, settings = st.tabs(["Main", "Settings"])
    left, right = st.columns(2)

    # nesting tabs and columns
    with main:
        with left:
            name = st.text_input(
                "What's your name", value="Lambda"
            )

        with right:
            greeting(name)

if __name__ == "__main__":
    app()
```

⚠ This example is illustratory and real applications quickly get more complex.

2. Cache and state management

When the user interacts with the app, Streamlit reruns your entire Python code to update what's displayed on screen (reference). By default, no data is preserved between updates and all computations need to be executed again. Your application suffer slow downs if you handle large dataframes or load machine learning models in memory for instance. To overcome this limitation, Streamlit allows to cache expensive operations via the decorators `@streamlit.cache_data` and `@streamlit.cache_resource` and store state variables between reruns in the global dictionary `streamlit.session_state` or via `key` attributes of input widget. State management becomes particularly important when building a multipage app where each page is defined in a separate Python file and can't communicate by default.

```
import pandas as pd
import streamlit as st

@st.cache_data
def load_dataframe(path: str) -> pd.DataFrame:
    return pd.read_parquet(path)

def app():
    st.title("Apache Hamilton + Streamlit 🐱🚀")

    # load_dataframe() will only run the first time
    df = load_dataframe(path="...")
    st.dataframe(df)

    # If favorite flavor is known, display it.
    if st.session_state("favorite"):
        st.write(f"Your favorite ice cream is:
{st.session_state['favorite']}")
```

```
    # Ask for the favorite ice cream until an answer is given.
    else:
        st.text_input(
            "What's your favorite ice cream flavor?",
            key="favorite",  # key to st.session_state
        )

if __name__ == "__main__":
    app()
```

⚠ This example is illustratory and real applications quickly get more complex.

Apache Hamilton + Streamlit

Adding Apache Hamilton to your Streamlit application can provide a better separation between the dataflow and the UI logic. They pair nicely together because Apache Hamilton is also stateless. Once defined, each call to `Driver.execute()` is independent. Therefore, on each Streamlit rerun, you use `Driver.execute()` to complete computations. Using Apache Hamilton this way allows you to write your dataflow into Python modules and outside of the Streamlit.

Example

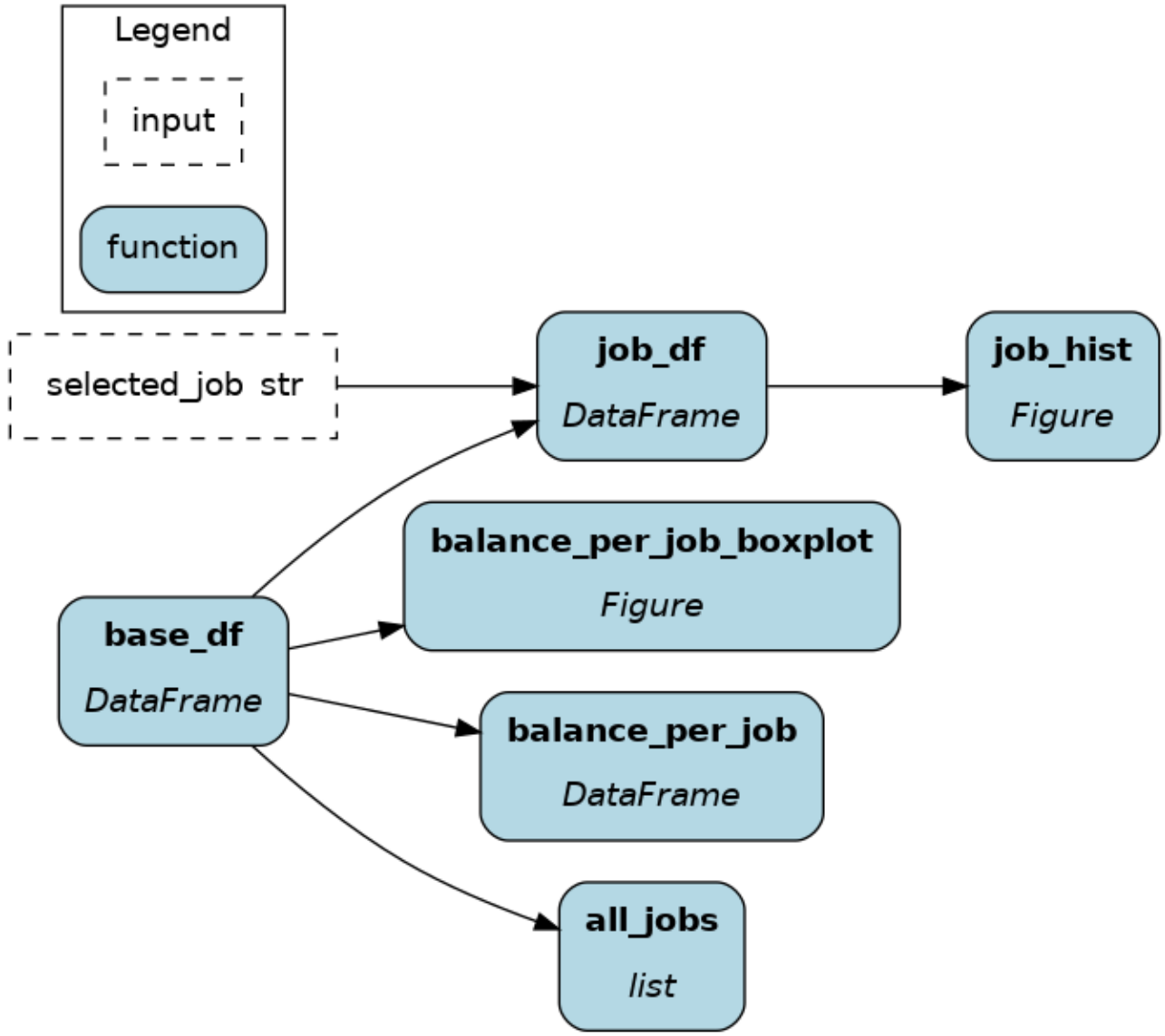
In this example, we will build a simple financial dashboard based on the Kaggle Bank Marketing Dataset.

The full code can be found on GitHub

First, Apache Hamilton transformations are defined in the module `logic.py`. This includes downloading the data from the web, getting unique values for `job`, conducting groupby aggregates, and creating `plotly` figures.

```
# logic.py
import pandas as pd
import plotly.express as px
from plotly.graph_objs import Figure


def base_df() -> pd.DataFrame:
    path = "https://raw.githubusercontent.com/Lexie88rus/bank-
marketing-analysis/master/bank.csv"
    return pd.read_csv(path)


def all_jobs(base_df: pd.DataFrame) -> list[str]:
    return base_df["job"].unique().tolist()
```

```python
def balance_per_job(base_df: pd.DataFrame) -> pd.DataFrame:
    return base_df.groupby("job")["balance"].describe().astype(int)


def balance_per_job_boxplot(base_df: pd.DataFrame) -> Figure:
    return px.box(base_df, x="job", y="balance")


def job_df(base_df: pd.DataFrame, selected_job: str) -> pd.DataFrame:
    return base_df.loc[base_df['job']==selected_job]


def job_hist(job_df: pd.DataFrame) -> Figure:
    return px.histogram(job_df["balance"])
```



Then, the Streamlit UI is defined in `app.py`. Notice a few things:

- `app.py` doesn't have to depend on `pandas` and `plotly`.
- `@cache_resource` allows to create the `Driver` only once.
- `@cache_data` on `_execute()` will automatically cache any Apache Hamilton result based on the combination of arguments (`final_vars`, `inputs`, and `overrides`)
- `get_state_inputs()` and `get_state_overrides()` will collect values from user inputs.
- `execute()` parses the inputs and overrides from the state and call `_execute()`.

```
# app.py
from typing import Optional

from hamilton import driver
import streamlit as st

import logic

# cache to avoid rebuilding the Driver
@st.cache_resource
def get_hamilton_driver() -> driver.Driver:
    return (
        driver.Builder()
        .with_modules(logic)
        .build()
    )

# cache results for the set of inputs
@st.cache_data
def _execute(
    final_vars: list[str],
    inputs: Optional[dict] = None,
    overrides: Optional[dict] = None,
) -> dict:
    """Generic utility to cache Apache Hamilton results"""
    dr = get_hamilton_driver()
    return dr.execute(final_vars, inputs=inputs, overrides=overrides)


def get_state_inputs() -> dict:
    keys = ["selected_job"]
    return {k: v for k, v in st.session_state.items() if k in keys}


def get_state_overrides() -> dict:
    keys = []
    return {k: v for k, v in st.session_state.items() if k in keys}
```

```
def execute(final_vars: list[str]):
    return _execute(final_vars, get_state_inputs(),
get_state_overrides())


def app():
    st.title("Apache Hamilton + Streamlit 🐱🚀")

    # run the base data that always needs to be displayed
    data = execute(["all_jobs", "balance_per_job",
"balance_per_job_boxplot"])

    # display the base dataframe and plotly chart
    st.dataframe(data["balance_per_job"])
    st.plotly_chart(data["balance_per_job_boxplot"])

    # get the selection options from `data`
    # store the selection in the state `selected_job`
    st.selectbox("Select a job", options=data["all_jobs"],
key="selected_job")
    # get the value from the dict
    st.plotly_chart(execute(["job_hist"])["job_hist"])


if __name__ == "__main__":
    app()
```

Benefits

- **Clearer scope**: the decoupling between `app.py` and `logic.py` makes it easier to add data transformations or extend UI, and debug errors associated with either.
- **Reusable code**: the module `logic.py` can be reused elsewhere with Apache Hamilton.
  - If you are building a proof-of-concept with Streamlit, your Apache Hamilton module will be able to grow with your project and be useful for your production pipelines.
  - If you are already building dataflows with Apache Hamilton, using it with Streamlit ensures your dashboard metrics have the same implementation with your production pipeline (i.e., prevent implementation skew)
- **Performance boost**: by caching the Hamilton Driver and its execution call, we are able to effectively cache all data operations in a few lines of code. Furthermore, Apache Hamilton can scale further by using a remote task executor on a separate machine from the Streamlit application.

dbt

If you're familiar with DBT, you likely noticed that it can fill a similar role to Apache Hamilton. What DBT does for SQL files (organizing functions, providing lineage capabilities, making testing easier), Apache Hamilton does for python functions.

Many projects span the gap between SQL and python, and Apache Hamilton is a natural next step for an ML workflow after extracting data from DBT.

This example shows how you can use DBT's new python capabilities to integrate a Apache Hamilton dataflow with a DBT pipeline.

Find the full, working dbt project here.

Code Comparisons

This section showcases what Apache Hamilton code looks like in comparison to other popular libraries and frameworks.

Kedro

Both `Kedro` and `Apache Hamilton` are Python tools to help define directed acyclic graph (DAG) of data transformations. While there's overlap between the two in terms of features, we note two main differences:

- `Kedro` is imperative and focuses on tasks; `Apache Hamilton` is declarative and focuses on assets.
- `Kedro` is heavier and comes with a project structure, YAML configs, and dataset definition to manage; `Apache Hamilton` is lighter to adopt and you can progressively opt-in features that you find valuable.

On this page, we'll dive into these differences, compare features, and present some code snippets from both tools.

Note

See this GitHub repository to compare a full project using Kedro or Apache Hamilton.

Imperative vs. Declarative

There are 3 steps to build and run a dataflow (a DAG, a data pipeline, etc.)

1. Define transformation steps

2. Assemble steps into a dataflow

3. Execute the dataflow to produce data artifacts (tables, ML models, etc.)

1. Define steps

Imperative (`Kedro`) vs. declarative (`Apache Hamilton`) leads to significant differences in **Step 2** and **Step 3** that will shape how you work with the tool. However, **Step 1** remains similar. In fact, both tools use the term **nodes** to refer to steps.

Kedro (imperative)

```
# nodes.py
import pandas as pd

def _is_true(x: pd.Series) ->
pd.Series:
    return x == "t"

def preprocess_companies(companies:
pd.DataFrame) -> pd.DataFrame:
    """Preprocesses the data for
companies."""
    companies["iata_approved"] =
_is_true(companies["iata_approved"])
    return companies

def preprocess_shuttles(shuttles:
pd.DataFrame) -> pd.DataFrame:
    """Preprocesses the data for
shuttles."""
    shuttles["d_check_complete"] =
_is_true(
        shuttles["d_check_complete"]
    )

shuttles["moon_clearance_complete"]
= _is_true(

shuttles["moon_clearance_complete"]
    )
    return shuttles

def create_model_input_table(
    shuttles: pd.DataFrame,
companies: pd.DataFrame,
) -> pd.DataFrame:

"""Combines all data to create a
model input table."""
    shuttles = shuttles.drop("id",
axis=1)
```

Apache Hamilton (declarative)

```
# dataflow.py
import pandas as pd

def _is_true(x: pd.Series) ->
pd.Series:
    return x == "t"

def
companies_preprocessed(companies:
pd.DataFrame) -> pd.DataFrame:
    """Companies with added column
`iata_approved`"""
    companies["iata_approved"] =
_is_true(companies["iata_approved"])
    return companies

def shuttles_preprocessed(shuttles:
pd.DataFrame) -> pd.DataFrame:
    """Shuttles with added columns
`d_check_complete`
    and
`moon_clearance_complete`."""
    shuttles["d_check_complete"] =
_is_true(
        shuttles["d_check_complete"]
    )

shuttles["moon_clearance_complete"]
= _is_true(

shuttles["moon_clearance_complete"]
    )
    return shuttles

def model_input_table(
    shuttles_preprocessed:
pd.DataFrame,
    companies_preprocessed:
pd.DataFrame,
) -> pd.DataFrame:
```

Kedro (imperative)

```
    model_input_table =
shuttles.merge(
        companies,
left_on="company_id", right_on="id"
    )
    model_input_table =
model_input_table.dropna()
    return model_input_table
```

Apache Hamilton (declarative)

```
"""Table containing shuttles and
companies data."""
    shuttles_preprocessed =
shuttles_preprocessed.drop("id",
axis=1)
    model_input_table =
shuttles_preprocessed.merge(
        companies_preprocessed,
left_on="company_id", right_on="id"
    )
    model_input_table =
model_input_table.dropna()
    return model_input_table
```

The function implementations are exactly the same. Yet, notice that the function names and docstrings were edited slightly. Imperative approaches like `Kedro` typically refer to steps as *tasks* and prefer verbs to describe "the action of the function". Meanwhile, declarative approaches such as `Apache Hamilton` describe steps as *assets* and use nouns to refer to "the value returned by the function". This might appear superficial, but it relates to the difference in **Step 2** and **Step 3**.

2. Assemble dataflow

With `Kedro`, you need to take your functions from **Step 1** and create `node` objects, specifying the node's name, inputs, and outputs. Then, you create a `pipeline` from a set of `nodes` and `Kedro` assembles the nodes into a DAG. Imperative approaches need to specify how tasks (Kedro nodes) relate to each other.

With `Apache Hamilton`, you pass the module containing all functions from **Step 1** and let Apache Hamilton create the `nodes` and the `dataflow`. This is possible because in declarative approaches like Apache Hamilton, each function defines a transform **and** its dependencies on other functions. Notice how in **Step 1**, `model_input_table()` has parameters `shuttles_preprocessed` and `companies_preprocessed`, which refers to other functions in the module. This contains all the required information to build the DAG.

Kedro (imperative)

```
# pipeline.py
from kedro.pipeline import Pipeline,
node, pipeline
```

Apache Hamilton (declarative)

```
# run.py
from hamilton import driver
import dataflow  # module containing
```

Kedro (imperative)

```
from nodes import (
    create_model_input_table,
    preprocess_companies,
    preprocess_shuttles
)

def create_pipeline(**kwargs) ->
Pipeline:
    return pipeline(
        [
            node(

func=preprocess_companies,
                inputs="companies",

outputs="preprocessed_companies",

name="preprocess_companies_node",
            ),
            node(

func=preprocess_shuttles,
                inputs="shuttles",

outputs="preprocessed_shuttles",

name="preprocess_shuttles_node",
            ),
            node(

func=create_model_input_table,
                inputs=[

"preprocessed_shuttles",

"preprocessed_companies"
                ],

outputs="model_input_table",

name="create_model_input_table_node",
            ),
        ]
    )
```

Apache Hamilton (declarative)

```
definitions

# pass the module to the `Builder` t
`Driver`
dr =
driver.Builder().with_modules(datafl
```

Benefits of adopting a declarative approach

- Less errors since you skip manual node creation (i.e., strings **will** lead to typos).
- Handle complexity since assembling a dataflow remains the same for 10 or 1000 nodes.
- Maintainability improves since editing your functions (**Step 1**) modifies the structure of your DAG, removing the pipeline definition as a failure point.
- Readability improves because you can understand how functions relate to each other without jumping between files.

These benefits of `Apache Hamilton` encourage developers to write smaller functions that are easier to debug and maintain, leading to major code quality gains. On the opposite, the burden of `node` and `pipeline` creation as projects grow in size lead to users stuffing more and more logic in a single node, making it increasingly harder to maintain.

3. Execute dataflow

The primary way to execute `Kedro` pipelines is to use the command line tool with `kedro run --pipeline=my_pipeline`. Pipelines are typically designed for all nodes to be executed while reading data and writing results while going through nodes. It is closer to macro-orchestration frameworks like Airflow in spirit.

On the opposite, `Apache Hamilton` dataflows are primarily meant to be executed programmatically (i.e., via Python code) and return results in-memory. This makes it easy to use `Apache Hamilton` within a FastAPI service service or to power an LLM application.

For comparable side-by-side code, we can dig into `Kedro` and use the `SequentialRunner` programmatically. To return pipeline results in-memory we would need to hack further with `kedro.io.MemoryDataset`.

> **Note**
>
> Apache Hamilton also has rich support for I/O operations (see **Feature comparison** below)

Kedro (imperative)	Apache Hamilton (declarati

```python
# run.py
from kedro.runner import SequentialRunner
from kedro.framework.session import KedroSession
from kedro.framework.startup import bootstrap_project
```

```python
# run.py
import pandas as pd
from hamilton impor
import dataflow
```

Kedro (imperative)

```
from pipeline import create_pipeline
# ^ from Step 2

bootstrap_project(".")
with KedroSession.create() as session:
    context = session.load_context()
    catalog = context.catalog

pipeline =
create_pipeline().to_nodes("create_model_input_table")
SequentialRunner().run(pipeline, catalog)
# doesn't return values in-memory
```

Apache Hamilton (declarati

```
dr =
driver.Builder().wi
# ^ from Step 2
inputs = dict(
    companies=pd.re
companies.parquet")
    shuttles=pd.rea
shuttles.parquet"),
)
results = dr.execut
inputs=inputs)
# results is a dict
VALUE}
```

An imperative pipeline like `Kedro` is a series of step, just like a recipe. The user can specify "from nodes" or "to nodes" to *slice* the pipeline and not have to execute it in full.

For declarative dataflows like `Apache Hamilton` you request assets / nodes by name and the tool will determine the required nodes to execute (here `"model_input_table"`) avoiding wasteful compute.

The simple Python interface provided by `Apache Hamilton` allows you to potentially define and execute your dataflow from a single file, which is great to kickstart an analysis or project. Just use `python dataflow.py` to execute it!

```
# dataflow.py
import pandas as pd

def _is_true(x: pd.Series) -> pd.Series:
    return x == "t"

def preprocess_companies(companies: pd.DataFrame) -> pd.DataFrame:
    """Preprocesses the data for companies."""
    companies["iata_approved"] = _is_true(companies["iata_approved"])
    return companies

def preprocess_shuttles(shuttles: pd.DataFrame) -> pd.DataFrame:
    """Preprocesses the data for shuttles."""
    shuttles["d_check_complete"] = _is_true(
        shuttles["d_check_complete"]
    )
    shuttles["moon_clearance_complete"] = _is_true(
        shuttles["moon_clearance_complete"]
    )
```

```
    return shuttles

def create_model_input_table(
    shuttles: pd.DataFrame, companies: pd.DataFrame,
) -> pd.DataFrame:
    """Combines all data to create a model input table."""
    shuttles = shuttles.drop("id", axis=1)
    model_input_table = shuttles.merge(
        companies, left_on="company_id", right_on="id"
    )
    model_input_table = model_input_table.dropna()
    return model_input_table

if __name__ == "__main__":
    from hamilton import driver
    import dataflow  # import itself as a module

    dr = driver.Builder().with_modules(dataflow).build()
    inputs=dict(
        companies=pd.read_parquet("path/to/companies.parquet"),
        shuttles=pd.read_parquet("path/to/shuttles.parquet"),
    )
    results = dr.execute(["model_input_table"], inputs=inputs)
```

Framework weight

After imperative vs. declarative, the next largest difference is the type of user experience they provide. `Kedro` is a more opiniated and heavier framework; `Apache Hamilton` is on the opposite end of the spectrum and tries to be the lightest library possible. This changes the learning curve, adoption, and how each tool will integrate with your stack.

Kedro

`Kedro` is opiniated and provides clear guardrails on how to do things. To begin using it, you'll need to learn to:

- Define nodes and register pipelines
- Register datasets using the data catalog construct
- Pass parameters to data runs
- Configure environment variables and credentials
- Navigate the project structure

This provides guidance when building your first data pipeline, but it's also a lot to take in at once. As you'll see in the project comparison on GitHub, `Kedro` involves more files making it harder to navigate. Also, it's reliant on YAML which is generally seen as an unreliable format. If you have an existing data stack or favorite library, it might clash with `Kedro`'s way of thing (e.g., you have credentials management tool; you prefer Hydra for configs).

Apache Hamilton w~~~~~~~~~~~~~~~

`Apache Hamilton` attempts to get you started quickly. In fact, this page pretty much covered what you need to know:

- Define nodes and a dataflow using regular Python functions (no need to even import `hamilton`!)
- Build a `Driver` with your dataflow module and call `.execute()` to get results

`Apache Hamilton` allows you to start light and opt-in features as your project's requirements evolve (data validation, scaling compute, testing, etc.). Python is a powerful language with rich editor support and tooling hence why it advocates for "everything in Python" instead of external configs in YAML or JSON. For example, parameters, data assets, and configurations can very much live as dataclasses within a `.py` file. `Apache Hamilton` was built with an extensive plugin system. There are many extensions, some contributed by users, to adapt Apache Hamilton to your project, and it's easy for you to extend yourself for further customization.

In fact, `Apache Hamilton` is so lightweight, you could even run it inside `Kedro`!

Feature comparison

Trait	Kedro	Apache Hamilton
Focuses on	Tasks (imperative)	Assets (declarative)
Code structure	Opiniated. Makes assumptions about pipeline creation & registration and configuration.	Unopiniated.
In-memory execution	Execute using a KedroSession, but returning values in-memory is hacky.	Default

Trait	Kedro	Apache Hamilton
I/O execution	Datasets and Data Catalog	Data Savers & Loaders
Expressive DAG definition	⛔	Function modifiers
Column-level transformations	⛔	✅
LLM applications	⛔ Limited by in-memory execution and return values.	✅ declarative API in-memory makes it easy (RAG app).
Static DAG visualizations	Need `Kedro Viz` installed to export static visualizations.	Visualize entire dataflow, execution path, query what's upstream, etc. directly in a notebook or output to a file (`.png`, `.svg`, etc.). Single dependency is `graphviz`.
Interactive DAG viewer	Kedro Viz	Apache Hamilton UI
Data validation	Community Pandera plugin	Native and Pandera plugin
Executors	Sequential, multiprocessing, multi-threading	Sequential, async, multiprocessing, multi-threading
Executor extension	Spark integration	PySpark, Dask, Ray, Modal
Dynamic branching	⛔	Parallelizable/Collect for easy parallelization.
	✅	✅

Trait	Kedro	Apache Hamilton
Command line tool (CLI)		
Node and pipeline testing	✅	✅
Jupyter notebook extensions	✅	✅

Both `Kedro` and `Apache Hamilton` provide applications to view dataflows/pipelines and interact with their results. Here, `Kedro` provides a lighter webserver and UI, while `Apache Hamilton` offers a production-ready containerized application.

Trait	Kedro Viz	Apache Hamilton UI
Interactive dataflow viewer	✅	✅
View code definition of nodes	✅	✅
Code versioning	Git SHA (may be out of sync with actual code)	Node-level versioning at runtime
Collapsible view	✅	✅
Tag nodes	✅	✅
Execution observability	⛔	✅
	⛔	✅

Trait	Kedro Viz	Apache Hamilton UI
Artifact lineage and versioning		
Column-level lineage	⛔	✅
Compare run results	✅	✅
Rich artifact view	Preview 5 dataframe rows. Metadata about artifact (column count, row count, size).	Automatic statistical profiling of various dataframe libraries.

More information

For a full side-by-side example of Kedro and Apache Hamilton, visit this GitHub repository

For more questions, join our Slack Channel

Dagster

Here are some code snippets to compare the macro orchestrator Dagster to the micro orchestrator Apache Hamilton. Apache Hamilton can run inside Dagster, but you wouldn't run Dagster inside Apache Hamilton.

While the two have different scope, there's a lot of overlap between the two both in terms of functionality and API. Indeed, Dagster's software-defined assets introduced in 2022 matches Apache Hamilton's declarative approach and should feel familiar to users of either.

TL;DR

Trait	Apache Hamilton	Dagster
Declarative API	✅	✅
Dependencies	Lightweight library with minimal dependencies (`numpy`, `pandas`, `typing_inspect`). Minimizes dependency conflicts.	Heavier framework/system with several dependencies (`pydantic`, `sqlalchemy`, `requests`, `Jinja2`, `protobuf`). `urllib3` on which depends `requests` introduced breaking changes several times and `pydantic` v1 and v2 are incompatible.
Macro orchestration	DIY or in tandem with Dagster, Airflow, Prefect, Metaflow, etc.	Includes: manual, schedules, sensors, conditional execution
Micro orchestration (i.e., `dbt`, `LangChain`)	Can run anywhere (locally, notebook, macro orchestrator, FastAPI, Streamlit, pyodide, etc.)	❌
Code structure	Since it's micro, there are no restrictions.	Since it's macro, a certain code structure is required to properly package code. The prevalent use of relative imports in the tutorial reduces code reusability.
LLM applications	Well-suited for LLM applications since it's a micro orchestration framework.	❌
Lineage	Fine-grained / column-level lineage. Includes utilities to explore lineage.	Coarser operations to reduce orchestration and I/O overhead.

Trait	Apache Hamilton	Dagster
Visualization	View the dataflow and produce visual artifacts. Configurable and supports extensive custom styling.	Export Daster UI in `.svg`. No styling.
Run tracking	DAGWorks (premium)	Dagster UI
Experiment Managers	Has an experiment manager plugin	❌
Materializers	Data Savers & Loaders	IO Managers
Data validation	Native validators and pandera plugin	Asset checks (experimental), pandera integration
Versioning operations	Nodes and dataflow versions are derived from code.	Asset code version is specified manually.
Versioning data	Automated code version + data value are used to read from cache or compute new results with DiskCacheAdapter	Manual asset code version + upstream changes are used to trigger re-materialization
In-memory Execution	Default	Materialize in-memory
Task-based Execution	TaskBasedExecutor	Default
Dynamic branching	Parallelizable/Collect	Mapping/Collect
Hooks	Lifecycle hooks (easier to extend)	Op Hooks

Trait	Apache Hamilton	Dagster
Plugins	Spark, Dask, Ray, Datadog, polars, pandera, and more (Apache Hamilton is less restrictive and easier to extend)	Spark, Dask, polars, pandera, Databricks, Snowflake, Great Expections, and more (Dagster integrations are more involved to develop)
Interactive Development	Jupyter Magic, VSCode extension	❌

Dataflow definition

HackerNews top stories

Apache Hamilton

```
from hamilton.function_modifiers import
extract_columns

NEWSTORIES_URL = "https://hacker-
news.firebaseio.com/v0/topstories.json"

def topstory_ids(newstories_url: str =
NEWSTORIES_URL) -> list[int]:
    """Query the id of the top HackerNews
stories"""
    return
requests.get(newstories_url).json()[:100]

@extract_columns("title")
def topstories(topstory_ids: list[int]) -
> pd.DataFrame:
    """Query the top HackerNews stories
based on ids"""
    results = []
    for item_id in topstory_ids:
        item = requests.get(
            f"https://hacker-
news.firebaseio.com/v0/item/
{item_id}.json"
```

Dagster

```
from dagster import AssetExecut
MetadataValue, asset, Materiali

@asset
def topstory_ids() -> None:
    newstories_url = "https://h
news.firebaseio.com/v0/topstori
    top_new_story_ids =
requests.get(newstories_url).js

    os.makedirs("data", exist_o
    with open("data/topstory_id
as f:
        json.dump(top_new_story

@asset(deps=[topstory_ids])
def topstories(context: AssetEx
-> MaterializeResult:
    with open("data/topstory_id
as f:
        topstory_ids = json.loa

    results = []
    for item_id in topstory_ids
```

Apache Hamilton

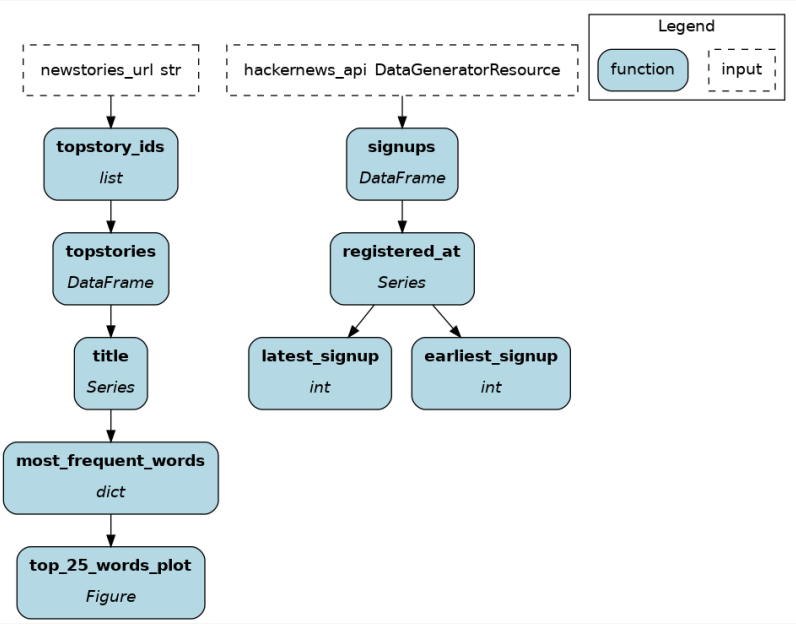
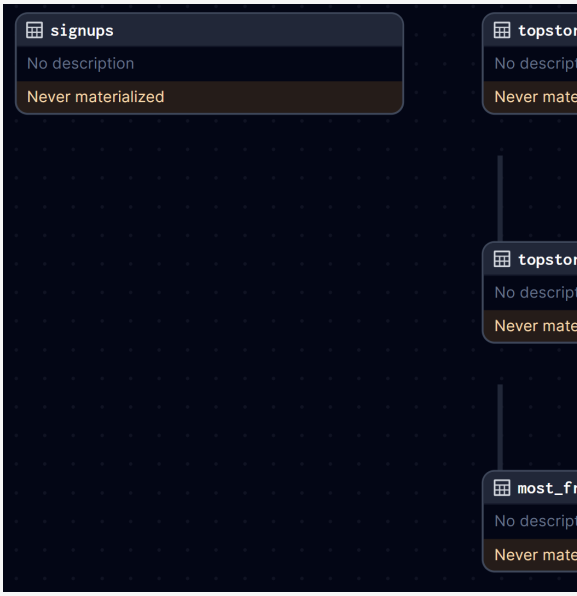
```
        ).json()
        results.append(item)
    return pd.DataFrame(results)

def most_frequent_words(title: pd.Series)
-> dict[str, int]:
    """Compute word frequency in
HackerNews story titles"""
    STOPWORDS = ["a", "the", "an", "of",
"to", "in",
                 "for", "and", "with",
"on", "is", "\u2013"]
    word_counts = {}
    for raw_title in title:
        for word in
raw_title.lower().split():
            word = word.strip(".,-!?:;()
[]'\"-")
            if len(word) == 0:
                continue

            if word in STOPWORDS:
                continue

            word_counts[word] =
word_counts.get(word, 0) + 1
    return word_counts

def
top_25_words_plot(most_frequent_words:
dict[str, int]) -> Figure:
    """Bar plot of the frequency of the
top 25 words in HackerNews titles"""
    top_words = {
        pair[0]: pair[1]
        for pair in sorted(
            most_frequent_words.items(),
key=lambda x: x[1], reverse=True
        )[:25]
    }

    fig = plt.figure(figsize=(10, 6))
    plt.bar(list(top_words.keys()),
list(top_words.values()))
    plt.xticks(rotation=45, ha="right")

plt.title("Top 25 Words in Hacker News
Titles")
```

Dagster

```
        item = requests.get(
            f"https://hacker-
news.firebaseio.com/v0/item/{it
        ).json()
        results.append(item)

        if len(results) % 20 ==
            context.log.info(f"
{len(results)} items so far.")

    df = pd.DataFrame(results)
    df.to_csv("data/topstories.

    return MaterializeResult(
        metadata={
            "num_records": len(
            "preview":
MetadataValue.md(df.head().to_m
        }
    )

@asset(deps=[topstories])
def most_frequent_words() -> Ma
    stopwords = ["a", "the", "a
"in",
                 "for", "and",
"is"]
    topstories = pd.read_csv("d
topstories.csv")

    word_counts = {}
    for raw_title in topstories
        title = raw_title.lower
        for word in title.split
            word = word.strip("
[]'\"-")
            if cleaned_word in
len(cleaned_word) < 0:
                continue

            word_counts[cleaned
word_counts.get(word, 0) + 1

    top_words = {
        pair[0]: pair[1]
        for pair in sorted(
            word_counts.items()
x[1], reverse=True
```

```
    plt.tight_layout()
    return fig

@extract_columns("registered_at")
def signups(hackernews_api:
DataGeneratorResource) -> pd.DataFrame:
    """Query HackerNews signups using a
mock API endpoint"""
    return
pd.DataFrame(hackernews_api.get_signups())

def earliest_signup(registered_at:
pd.Series) -> int:
    """Earliest signup on HackerNews"""
    return registered_at.min()

def latest_signup(registered_at:
pd.Series) -> int:
    """Latest signup on HackerNews"""
    return registered_at.min()
```

```
        )[:25]
    }

    plt.figure(figsize=(10, 6))
    plt.bar(list(top_words.keys
list(top_words.values()))
    plt.xticks(rotation=45, ha=
    plt.title("Top 25 Words in
Titles")
    plt.tight_layout()

    buffer = BytesIO()
    plt.savefig(buffer, format=
    image_data =
base64.b64encode(buffer.getvalu

    md_content = f"![img](data:
png;base64,{image_data.decode()

    with open("data/most_freque
"w") as f:
        json.dump(top_words, f)

    return MaterializeResult(
        metadata={"plot":
MetadataValue.md(md_content)}
    )

@asset
def signups(hackernews_api:
DataGeneratorResource) -> Mater
    signups =
pd.DataFrame(hackernews_api.get

    signups.to_csv("data/signup

    return MaterializeResult(
        metadata={
            "Record Count": len
            "Preview":
MetadataValue.md(signups.head()
            "Earliest Signup":
signups["registered_at"].min(),
            "Latest Signup":
signups["registered_at"].max(),
        }
    )
```

Apache Hamilton	Dagster
	

Key points

Trait	Apache Hamilton	Dagster
Define operations	Uses the native Python function signature. The dataflow is assembled based on function/parameter names and type annotations.	Uses the `@asset` decorator to transform function in operations and specify dependencies by passing functions.
Data I/O	Loading/Saving is decoupled from the dataflow definition. The code becomes more portable and facilitates moving from dev to prod.	Each asset code operations is coupled with I/O. Hard-coding this behavior reduces maintainability.
Lineage	Favors granular operations and fine-grained lineage. For example, `most_frequent_words()` operates on a single column and the `top_25_words_plot` is its own function.	Favors chunking dataflow into meaningful assets to reduce the orchestration and I/O overhead per operation. Finer lineage is complex to achieve and requires

Trait	Apache Hamilton	Dagster
		using `@op`, `@graph`, `@job`, and `@asset` (ref)
Documentation	Uses the native Python docstrings. Further metadata can be added using the `@tag` decorator.	Uses `MaterializeResult` to store metadata.

Dataflow execution

HackerNews top stories

Apache Hamilton

```
import os

from hamilton import driver
from hamilton.io.materialization import to
from hamilton.plugins import matplotlib_extensions

import dataflow  # import module with dataflow
definition
from mock_api import DataGeneratorResource

def main():
    dr = (
        driver.Builder()
        .with_modules(dataflow)  # pass the module
        .build()
    )

    # load environment variable
    num_days =
os.environ.get("HACKERNEWS_NUM_DAYS_WINDOW")
    inputs = dict( # mock an API connection

hackernews_api=DataGeneratorResource(num_days=num_days),
    )

    # define I/O operations; decoupled from dataflow def
```

Dagster

```
from dagster impo
    AssetSelectio
    Definitions,
    define_asset_
    load_assets_f
    EnvVar,
)

from . import ass
from .resources i
DataGeneratorReso

# load assets fro
all_assets =
load_assets_from_
# select assets t
hackernews_job =
define_asset_job(
selection=AssetSe

# load environmen
num_days =
EnvVar.int("HACKE
defs = Definition
    assets=all_as
    jobs=[hackern
```

Apache Hamilton

```
    materializers = [
        to.json(  # JSON file type
            id="most_frequent_words.json",
            dependencies=["most_frequent_words"],
            path="data/most_frequent_words.json",
        ),
        to.csv(  # CSV file type
            id="topstories.csv",
            dependencies=["topstories"],
            path="data/topstories.csv",
        ),
        to.csv(
            id="signups.csv",
            dependencies=["signups"],
            path="data/signups.csv",
        ),
        to.plt(  # Use matplotlib.pyplot to render
            id="top_25_words_plot.plt",
            dependencies=["top_25_words_plot"],
            path="data/top_25_words_plot.png",
        ),
    ]

    # visualize materialization plan without executing
code
    dr.visualize_materialization(
        *materializers,
        inputs=inputs,
        output_file_path="dataflow.png"
    )
    # pass I/O operations and inputs to materialize
dataflow
    dr.materialize(*materializers, inputs=inputs)

if __name__ == "__main__":
    main()
```

Dagster

```
    resources={
connection
        "hackerne
DataGeneratorReso
    },
)
```

Key points

Trait	Apache Hamilton	Dagster
Execution instructions	Define a `Driver` using the `Builder` object. It automatically assembles the graph from the	Load assets from Python modules using `load_assets_from_modules` then create an asset job by

Trait	Apache Hamilton	Dagster
	dataflow definition found in `dataflow.py`	selecting assets to include. Finally, create a `Definitions` object to register on the orchestrator.
Execution plane	`Driver.materialize()` executes the dataflow in a Python process. Can be called as a script, using the CLI, or programmatically.	The asset job is executed by the orchestrator, either through Dagster UI, by a scheduler/sensor/trigger, or via the CLI.
Data I/O	I/O is decoupled from dataflow definition. People responsible for deployment can manage data sources without refactoring the dataflow. (Data I/O can be coupled if wanted.)	Data I/O is coupled with data assets which simplifies the execution code at the code of reusability.
Framework code	Leverages a maximum of standard Python mechanisms (imports, env variables, etc.).	Most constructs requires Dagster-specific code to leverage protobuf serialization.

More information

For a full side-by-side example of Dagster and Apache Hamilton, visit this GitHub repository

For more questions, join our Slack Channel!

LangChain

Here we have some code snippets that help compare a vanilla code implementation with LangChain and Apache Hamilton.

LangChain's focus is on hiding details and making code terse.

Apache Hamilton's focus instead is on making code more readable, maintainable, and importantly customizeable.

So don't be surprised that Apache Hamilton's code is "longer" - that's by design. There is also little abstraction between you, and the underlying libraries with Apache Hamilton. With LangChain they're abstracted away, so you can't really see easily what's going on underneath.

Rhetorical question: which code would you rather maintain, change, and update?

A simple joke example

Simple Invocation

Apache Hamilton

```
# hamilton_invoke.py
from typing import List

import openai


def llm_client() -> openai.OpenAI:
    return openai.OpenAI()


def joke_prompt(topic: str) -> str:
    return f"Tell me a short joke
about {topic}"


def joke_messages(joke_prompt:
str) -> List[dict]:
    return [{"role": "user",
"content": joke_prompt}]


def joke_response(llm_client:
openai.OpenAI,
                  joke_messages:
List[dict]) -> str:
    response =
llm_client.chat.completions.create(
        model="gpt-3.5-turbo",
        messages=joke_messages,
    )
    return
```

Vanilla

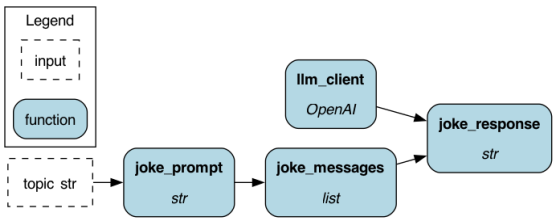
```
from typing import List

import openai

prompt_template = "Tell me a short
joke about {topic}"
client = openai.OpenAI()


def call_chat_model(messages:
List[dict]) -> str:
    response =
client.chat.completions.create(
        model="gpt-3.5-turbo",
        messages=messages,
    )
    return
response.choices[0].message.content


def invoke_chain(topic: str) ->
str:
    prompt_value =
prompt_template.format(topic=topic)
    messages = [{"role": "user",
"content": prompt_value}]
    return
call_chat_model(messages)


if __name__ == "__main__":
```

Apache Hamilton

```
response.choices[0].message.content


if __name__ == "__main__":
    import hamilton_invoke

    from hamilton import driver

    dr = (
        driver.Builder()
        .with_modules(hamilton_invoke)
        .build()
    )

dr.display_all_functions("hamilton-
invoke.png")

print(dr.execute(["joke_response"],

inputs={"topic": "ice cream"}))
```

Vanilla

```
    print(invoke_chain("ice
cream"))
```



The Hamilton DAG visualized.

A streamed joke example

With Apache Hamilton we can just swap the call function to return a streamed response. Note: you could use @config.when to include both streamed and non-streamed versions in the same DAG.

Streamed Version

Apache Hamilton

```
# hamilton_streamed.py
from typing import Iterator, List
```

Vanilla

```
from typing import List
from typing import Iterator
```

Apache Hamilton

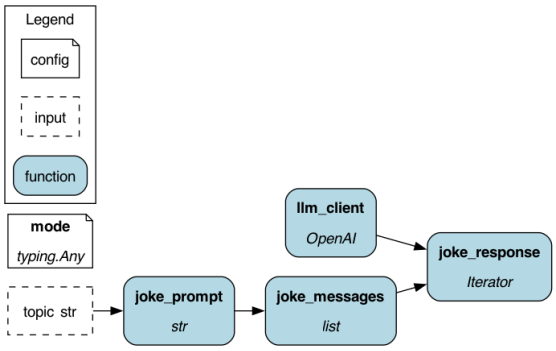
```
import openai


def llm_client() -> openai.OpenAI:
    return openai.OpenAI()


def joke_prompt(topic: str) -> str:
    return (
        f"Tell me a short joke
about {topic}"
    )


def joke_messages(
        joke_prompt: str) ->
List[dict]:
    return [{"role": "user",
             "content":
joke_prompt}]


def joke_response(
        llm_client: openai.OpenAI,
        joke_messages: List[dict])
-> Iterator[str]:
    stream =
llm_client.chat.completions.create(
        model="gpt-3.5-turbo",
        messages=joke_messages,
        stream=True
    )
    for response in stream:
        content =
response.choices[0].delta.content
        if content is not None:
            yield content


if __name__ == "__main__":
    import hamilton_streaming
    from hamilton import driver

    dr = (
        driver.Builder()
        .with_modules(hamilton_streaming)
        .build()
    )
```

Vanilla

```
import openai

prompt_template = "Tell me a short
joke about {topic}"
client = openai.OpenAI()


def stream_chat_model(
        messages: List[dict]) ->
Iterator[str]:
    stream =
client.chat.completions.create(
        model="gpt-3.5-turbo",
        messages=messages,
        stream=True,
    )
    for response in stream:
        content =
response.choices[0].delta.content
        if content is not None:
            yield content


def stream_chain(topic: str) ->
Iterator[str]:
    prompt_value =
prompt_template.format(topic=topic)
    return stream_chat_model(
        [{"role": "user",
"content": prompt_value}])


if __name__ == "__main__":
    for chunk in stream_chain("ice
cream"):
        print(chunk, end="",
flush=True)
```

Apache Hamilton **Vanilla**

```
    dr.display_all_functions(
        "hamilton-streaming.png"
    )
    result = dr.execute(
        ["joke_response"],
        inputs={"topic": "ice
cream"}
    )
    for chunk in
result["joke_response"]:
        print(chunk, end="",
flush=True)
```



The Hamilton DAG visualized.

A “batch” parallel joke example

In this batch example, the joke requests are parallelized. Note: with Apache Hamilton you can delegate to many different backends for parallelization, e.g. Ray, Dask, etc. We use multi-threading here.

Batch Parallel Version

Apache Hamilton **Vanilla**

```
# hamilton_batch.py
from typing import List

import openai

from hamilton.execution import
```

```
from concurrent.futures import
ThreadPoolExecutor
from typing import List

import openai
```

Apache Hamilton

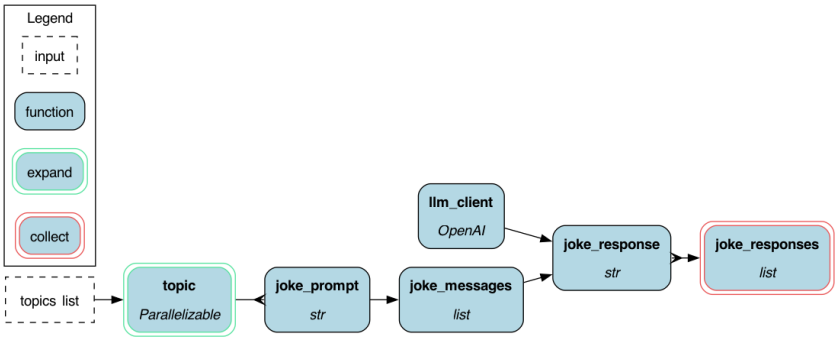
```
executors
from hamilton.htypes import Collect
from hamilton.htypes import
Parallelizable


def llm_client() -> openai.OpenAI:
    return openai.OpenAI()


def topic(
        topics: list[str]) ->
Parallelizable[str]:
    for _topic in topics:
        yield _topic


def joke_prompt(topic: str) -> str:
    return f"Tell me a short joke
about {topic}"


def joke_messages(
        joke_prompt: str) ->
List[dict]:
    return [{"role": "user",
             "content":
joke_prompt}]


def joke_response(llm_client:
openai.OpenAI,
                  joke_messages:
List[dict]) -> str:
    response =
llm_client.chat.completions.create(
        model="gpt-3.5-turbo",
        messages=joke_messages,
    )
    return
response.choices[0].message.content


def joke_responses(
        joke_response:
Collect[str]) -> List[str]:
    return list(joke_response)
```

Vanilla

```
prompt_template = "Tell me a short
joke about {topic}"
client = openai.OpenAI()


def call_chat_model(messages:
List[dict]) -> str:
    response =
client.chat.completions.create(
        model="gpt-3.5-turbo",
        messages=messages,
    )
    return
response.choices[0].message.content


def invoke_chain(topic: str) ->
str:
    prompt_value =
prompt_template.format(topic=topic)
    messages = [{"role": "user",
                 "content":
prompt_value}]
    return
call_chat_model(messages)


def batch_chain(topics: list) ->
list:
    with
ThreadPoolExecutor(max_workers=5)
as executor:
        return list(

executor.map(invoke_chain, topics)
        )


if __name__ == "__main__":
    print(
        batch_chain(
            ["ice cream",
"spaghetti", "dumplings"]
        )
    )
```

Apache Hamilton	Vanilla

```
if __name__ == "__main__":
    import hamilton_batch

    from hamilton import driver

    dr = (
        driver.Builder()
        .with_modules(hamilton_batch)
        .enable_dynamic_execution(

allow_experimental_mode=True
        )
        .with_remote_executor(

executors.MultiThreadingExecutor(5)
        )
        .build()
    )

dr.display_all_functions("hamilton-
batch.png")
    print(
        dr.execute(
            ["joke_responses"],
            inputs={
                "topics": ["ice
cream",

"spaghetti",

"dumplings"]
            }
        )
    )

    # can still run single chain
with overrides
    # and getting just one response
    print(
        dr.execute(
            ["joke_response"],
            overrides={"topic":
"lettuce"}
        )
    )
```



The Hamilton DAG visualized.

A “async” joke example

Here we show how to make the joke using async constructs. With Apache Hamilton you can mix and match async and regular functions, the only change is that you need to use the async Hamilton Driver.

Async Version

Apache Hamilton

```
# hamilton_async.py
from typing import List

import openai


def llm_client() ->
openai.AsyncOpenAI:
    return openai.AsyncOpenAI()


def joke_prompt(topic: str) -> str:
    return (
        f"Tell me a short joke
about {topic}"
    )


def joke_messages(
        joke_prompt: str) ->
List[dict]:
    return [{"role": "user",
             "content":
joke_prompt}]
```

Vanilla

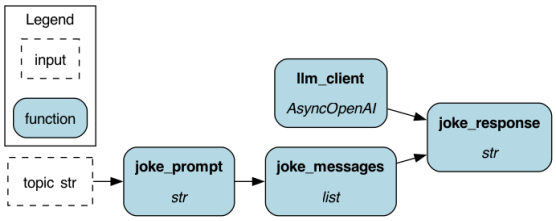
```
from typing import List

import openai


prompt_template = "Tell me a short
joke about {topic}"
client = openai.OpenAI()


async_client = openai.AsyncOpenAI()

async def acall_chat_model(
        messages: List[dict]) -> str:
    response = await (

async_client.chat.completions.create(
            model="gpt-3.5-turbo",
            messages=messages,
        )
    )
    return
response.choices[0].message.content

async def ainvoke_chain(topic: str) -
```

Apache Hamilton

```
async def joke_response(
        llm_client:
openai.AsyncOpenAI,
        joke_messages: List[dict])
-> str:
    response = await (

llm_client.chat.completions.create(
            model="gpt-3.5-turbo",
            messages=joke_messages,
        )
    )
    return
response.choices[0].message.content


if __name__ == "__main__":
    import asyncio

    import hamilton_async

    from hamilton import base
    from hamilton import
async_driver

    dr = async_driver.AsyncDriver(
        {},
        hamilton_async,

result_builder=base.DictResult()
    )

dr.display_all_functions("hamilton-
async.png")
    loop = asyncio.get_event_loop()
    result =
loop.run_until_complete(
        dr.execute(
            ["joke_response"],
            inputs={"topic": "ice
cream"}
        )
    )
    print(result)
```

Vanilla

```
> str:
    prompt_value =
prompt_template.format(
        topic=topic
    )
    messages = [{"role": "user",
                 "content":
prompt_value}]
    return await
acall_chat_model(messages)


if __name__ == "__main__":
    import asyncio

    loop = asyncio.get_event_loop()
    result = loop.run_until_complete(
        ainvoke_chain("ice cream")
    )
    print(result)
```



The Hamilton DAG visualized.

Switch LLM to completion for joke

Here we show how to make the joke switching to a different openAI model that is for completion. Note: we use the @config.when construct to augment the original DAG and add a new function that uses the different OpenAI model.

Completion Version

Apache Hamilton

```
# hamilton_completion.py
from typing import List

import openai

from hamilton.function_modifiers
import config


def llm_client() -> openai.OpenAI:
    return openai.OpenAI()


def joke_prompt(topic: str) -> str:
    return f"Tell me a short joke
about {topic}"


def joke_messages(
        joke_prompt: str) ->
List[dict]:
    return [{"role": "user",
             "content":
joke_prompt}]


@config.when(type="completion")
def joke_response__completion(
```

Vanilla

```
import openai

prompt_template = "Tell me a short
joke about {topic}"
client = openai.OpenAI()


def call_llm(prompt_value: str) ->
str:
    response =
client.completions.create(
        model="gpt-3.5-turbo-
instruct",
        prompt=prompt_value,
    )
    return response.choices[0].text

def invoke_llm_chain(topic: str) -
> str:
    prompt_value =
prompt_template.format(topic=topic)
    return call_llm(prompt_value)



if __name__ == "__main__":
```

Apache Hamilton

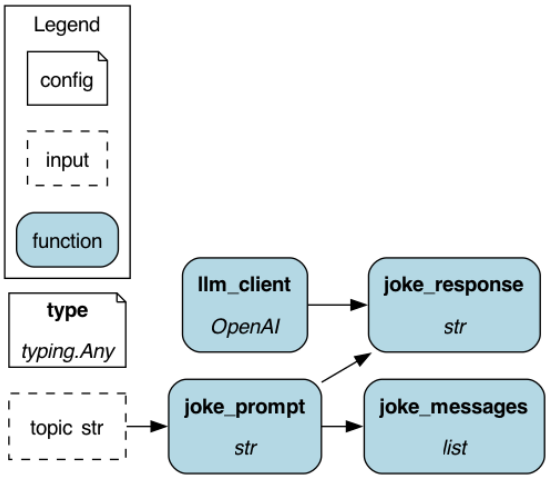
```
        llm_client: openai.OpenAI,
        joke_prompt: str) -> str:
    response =
llm_client.completions.create(
        model="gpt-3.5-turbo-
instruct",
        prompt=joke_prompt,
    )
    return response.choices[0].text


@config.when(type="chat")
def joke_response__chat(
        llm_client: openai.OpenAI,
        joke_messages: List[dict])
-> str:
    response =
llm_client.chat.completions.create(
        model="gpt-3.5-turbo",
        messages=joke_messages,
    )
    return
response.choices[0].message.content


if __name__ == "__main__":
    import hamilton_completion

    from hamilton import driver

    dr = (
        driver.Builder()
        .with_modules(hamilton_completion)
        .with_config({"type":
"completion"})
        .build()
    )
    dr.display_all_functions(
        "hamilton-completion.png"
    )
    print(
        dr.execute(
            ["joke_response"],
            inputs={"topic": "ice
cream"}
        )
    )
```

Vanilla

```
    print(invoke_llm_chain("ice
cream"))
```

Apache Hamilton **Vanilla**

```
    dr = (
        driver.Builder()
        .with_modules(hamilton_completion)
        .with_config({"type":
"chat"})
        .build()
    )

dr.display_all_functions("hamilton-
chat.png")
    print(
        dr.execute(
            ["joke_response"],
            inputs={"topic": "ice
cream"}
        )
    )
```



The Hamilton DAG visualized with configuration provided for the completion path. Note the dangling node - that's normal, it's not used in the completion path.

Switch to using Anthropic

Here we show how to make the joke switching to use a different model provider, in this case it's Anthropic. Note: we use the @config.when construct to augment the original DAG and add a new functions to use Anthropic.

Anthropic Version

Apache Hamilton

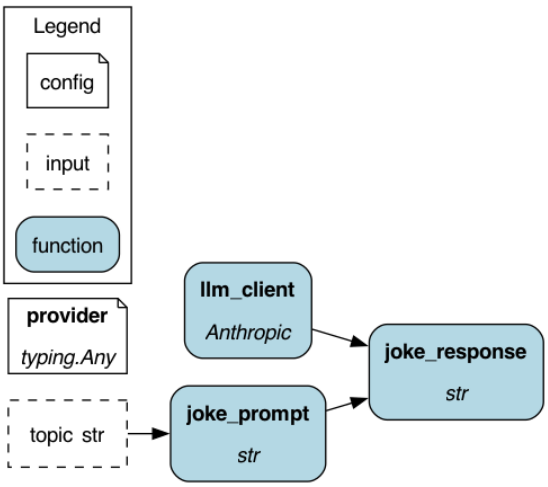
```
# hamilton_anthropic.py
import anthropic
import openai

from hamilton.function_modifiers import
config


@config.when(provider="openai")
def llm_client__openai() ->
openai.OpenAI:
    return openai.OpenAI()


@config.when(provider="anthropic")
def llm_client__anthropic() ->
anthropic.Anthropic:
    return anthropic.Anthropic()


def joke_prompt(topic: str) -> str:
    return (
        "Human:\n\n"
        "Tell me a short joke about
{topic}\n\n"
        "Assistant:"
    ).format(topic=topic)


@config.when(provider="openai")
def joke_response__openai(
        llm_client: openai.OpenAI,
        joke_prompt: str) -> str:
    response =
llm_client.completions.create(
        model="gpt-3.5-turbo-instruct",
        prompt=joke_prompt,
    )
    return response.choices[0].text


@config.when(provider="anthropic")
def joke_response__anthropic(
        llm_client: anthropic.Anthropic,
        joke_prompt: str) -> str:
    response =
```

Vanilla

```
import anthropic

prompt_template = "Tell me a shor
joke about {topic}"
anthropic_template = f"Human:
\n\n{prompt_template}\n\nAssistan
anthropic_client =
anthropic.Anthropic()


def call_anthropic(prompt_value:
-> str:
    response =
anthropic_client.completions.crea
        model="claude-2",
        prompt=prompt_value,
        max_tokens_to_sample=256,
    )
    return response.completion


def invoke_anthropic_chain(topic:
str) -> str:
    prompt_value =
anthropic_template.format(topic=t
    return
call_anthropic(prompt_value)


if __name__ == "__main__":
    print(invoke_anthropic_chain(
cream"))
```

Apache Hamilton **Vanilla**

```
llm_client.completions.create(
        model="claude-2",
        prompt=joke_prompt,
        max_tokens_to_sample=256
    )
    return response.completion


if __name__ == "__main__":
    import hamilton_invoke_anthropic

    from hamilton import driver

    dr = (
        driver.Builder()
        .with_modules(hamilton_invoke_anthropic)
        .with_config({"provider":
"anthropic"})
        .build()
    )
    dr.display_all_functions(
        "hamilton-anthropic.png"
    )
    print(
        dr.execute(
        ["joke_response"],
        inputs={"topic": "ice cream"}
        )
    )

    dr = (
        driver.Builder()
        .with_modules(hamilton_invoke_anthropic)
        .with_config({"provider":
"openai"})
        .build()
    )
    print(
        dr.execute(
            ["joke_response"],
            inputs={"topic": "ice
cream"}
        )
    )
```



The Hamilton DAG visualized with configuration provided to use Anthropic.

Logging

Here we show how to log more information about the joke request. Apache Hamilton has lots of customization options, and one out of the box is to log more information via printing.

Logging

Apache Hamilton	Vanilla

```python
# run.py
from hamilton import driver, lifecycle
import hamilton_anthropic

dr = (
    driver.Builder()
    .with_modules(hamilton_anthropic)
    .with_config({"provider": "anthropic"})
    # we just need to add this line to get
things printing
    # to the console; see DAGWorks for a more
off-the-shelf
    # solution.
    .with_adapters(lifecycle.PrintLn(verbosity=2))
    .build()
)
print(
    dr.execute(
        ["joke_response"],
        inputs={"topic": "ice cream"}
```

```python
import anthropic

prompt_template = "Tell me
{topic}"
anthropic_template = f"Huma
\n\n{prompt_template}\n\nAs
anthropic_client = anthropi


def call_anthropic(prompt_v
    response =
anthropic_client.completion
        model="claude-2",
        prompt=prompt_value
        max_tokens_to_sampl
    )
    return response.complet


def invoke_anthropic_chain_
str) -> str:
    print(f"Input: {topic}"
```

Apache Hamilton

```
    )
)
```

Vanilla

```
    prompt_value =
anthropic_template.format(t
    print(f"Formatted promp
    output = call_anthropic
    print(f"Output: {output
    return output


if __name__ == "__main__":

print(invoke_anthropic_chai
cream"))
```

Fallbacks

Fallbacks are pretty situation and context dependent. It's not that hard to wrap a function in a try/except block. The key is to make sure you know what's going on, and that a fallback was triggered. So in our opinion it's better to be explicit about it.

Logging

Apache Hamilton

```
import hamilton_anthropic
from hamilton import driver

anthropic_driver = (
    driver.Builder()
    .with_modules(hamilton_anthropic)
    .with_config({"provider":
"anthropic"})
    .build()
```

Vanilla

```
def invoke_chain_with_fallback(topic:
str) -> str:
    try:
        return invoke_chain(topic)  #
noqa: F821
    except Exception:
        return
invoke_anthropic_chain(topic)
# noqa: F821
```

Apache Hamilton

```
)
openai_driver = (
    driver.Builder()
    .with_modules(hamilton_anthropic)
    .with_config({"provider":
"openai"})
    .build()
)
try:
    print(
        anthropic_driver.execute(
            ["joke_response"],
            inputs={"topic":
"ice cream"}
        )
    )
except Exception:
    # this is the current way to
do fall backs
    print(
        openai_driver.execute(
            ["joke_response"],
            inputs={"topic":
"ice cream"}
        )
    )
```

Vanilla

```
if __name__ == '__main__':

print(invoke_chain_with_fallback("ice
cream"))
```

Airflow

For more details see this Apache Hamilton + Airflow blog post.

TL;DR:

1. Apache Hamilton complements Airflow. It'll help you write better, more modular, and testable code.

2. Apache Hamilton does not replace Airflow.

High-level differences:

- Apache Hamilton is a micro-orchestator. Airflow is a macro-orchestrator.
- Apache Hamilton is a Python library standardizing how you express python pipelines, while Airflow is a complete platform and system for scheduling and executing pipelines.
- Apache Hamilton focuses on providing a lightweight, low dependency, flexible way to define data pipelines as Python functions, whereas Airflow is a whole system that comes with a web-based UI, scheduler, and executor.
- Apache Hamilton pipelines are defined using pure Python code, that can be run anywhere that Python runs. While Airflow uses Python to describe a DAG, this DAG can only be run by the Airflow system.
- Apache Hamilton complements Airflow, and you can use Apache Hamilton within Airflow. But the reverse is not true.
- You can use Apache Hamilton directly in a Jupyter Notebook, or Python web-service. You can't do this with Airflow.

Code examples:

Looking at the two examples below, you can see that Apache Hamilton is a more lightweight and flexible way to define data pipelines. There is no scheduling information, etc required to run the code because Apache Hamilton runs the pipeline in the same process as the caller. This makes it easier to test and debug pipelines. Airflow, on the other hand, is a complete system for scheduling and executing pipelines. It is more complex to set up and run. Note: If you stuck the contents of *run.py* in a function within the *example_dag.py*, the Apache Hamilton pipeline could be used in the Airflow PythonOperator!

Apache Hamilton:

The below code here shows how you can define a simple data pipeline using Apache Hamilton. The pipeline consists of three functions that are executed in sequence. The pipeline is defined in a module called *pipeline.py*, and then executed in a separate script called *run.py*, which imports the pipeline module and executes it.

```
# pipeline.py
def raw_data() -> list[int]:
    return [1, 2, 3]

def processed_data(raw_data: list[int]) -> list[int]:
    return [x * 2 for x in data]
```

```
def load_data(process_data: list[int], client: SomeClient) -> dict:
    metadata = client.send_data(process_data)
    return metadata

# run.py -- this is the script that executes the pipeline
import pipeline
from hamilton import driver
dr = driver.Builder().with_modules(pipeline).build()
metadata = dr.execute(['load_data'],
inputs=dict(client=SomeClient()))
```

Airflow:

The below code shows how you can define the same pipeline using Airflow. The pipeline consists of three tasks that are executed in sequence. The entire pipeline is defined in a module called *example_dag.py*, and then executed by the Airflow scheduler.

```
# example_dag.py
from airflow import DAG
from airflow.operators.python_operator import PythonOperator
from datetime import datetime, timedelta

default_args = {
    'owner': 'airflow',
    'depends_on_past': False,
    'start_date': datetime(2023, 1, 1),
    'email_on_failure': False,
    'email_on_retry': False,
    'retries': 1,
    'retry_delay': timedelta(minutes=5),
}

dag = DAG(
    'example_dag',
    default_args=default_args,
    description='A simple DAG',
    schedule_interval=timedelta(days=1),
)

def extract_data():
    return [1, 2, 3]

def transform_data(data):
    return [x * 2 for x in data]

def load_data(data):
    client = SomeClient()
    client.send_data(data)
```

```
extract_task = PythonOperator(
    task_id='extract_data',
    python_callable=extract_data,
    dag=dag,
)

transform_task = PythonOperator(
    task_id='transform_data',
    python_callable=transform_data,
    op_args=['{{ ti.xcom_pull(task_ids="extract_data") }}'],
    dag=dag,
)

load_task = PythonOperator(
    task_id='load_data',
    python_callable=load_data,
    op_args=['{{ ti.xcom_pull(task_ids="transform_data") }}'],
    dag=dag,
)

extract_task >> transform_task >> load_task
```

Meet-ups

We have an active global meetup group that meets virtually once a month.

You can sign up for the group here.

Past Meet-ups

The below will be out of date. Please see our youtube for the latest recordings on past meetups.

July 2025

- **Topic**: Apache Transition

December 2024

- **Topic**: Community spotlight by Jernej Frank on decorators framework

October 2024

- **Topic**: “Building a Decisioning Engine for Data Scientists” by Sholto Armstrong

August 2024

- **Topic**: “Finding Optimal DAGs for Machine Learning/AI/RAG projects” by Gilad Rubin
- Recording

June 2024

- Recording

April 2024

- Recording

March 2024

- Recording

February 2024

- Recording
- Slides
- Community Spotlight: Arthur Andres and slides

Ecosystem

Welcome to the Apache Hamilton Ecosystem page! This page showcases the integrations, plugins, and external resources available for Apache Hamilton users.

🚀 Interactive Tutorials

tryhamilton.dev

Learn Apache Hamilton concepts through interactive, browser-based tutorials.

Built-in Integrations

Apache Hamilton provides first-class support for many popular data science and engineering tools through built-in plugins and adapters. These integrations are maintained by the Apache Hamilton community and included in the core project.

Data Frameworks

Apache Hamilton integrates seamlessly with popular data manipulation libraries:

Integration	Description	Documentation
pandas	DataFrame operations and transformations	Examples | ResultBuilder
Polars	High-performance DataFrame library	Examples | ResultBuilder
PySpark	Distributed data processing with Spark	Examples | GraphAdapter
Dask	Parallel computing and distributed arrays	Examples | GraphAdapter

Integration	Description	Documentation
Ray	Distributed computing framework	Examples | GraphAdapter
Ibis	Portable DataFrame API across backends	Integration Guide
Vaex	Out-of-core DataFrame library	Examples
Narwhals	DataFrame-agnostic interface	Examples | Lifecycle Hook
NumPy	Numerical computing arrays	ResultBuilder
PyArrow	Columnar in-memory data	ResultBuilder

Machine Learning & Data Science

Build and deploy ML workflows with Apache Hamilton:

Integration	Description	Documentation
MLflow	Experiment tracking and model registry	Examples | Lifecycle Hook
scikit-learn	Machine learning algorithms	Examples
XGBoost	Gradient boosting framework	IO Adapters
LightGBM	Gradient boosting framework	IO Adapters
Hugging Face	Transformers and NLP models	IO Adapters
Pandera	DataFrame validation	Examples

Integration	Description	Documentation
Pydantic	Data validation and settings	Decorator

Orchestration & Workflow Systems

Use Apache Hamilton within your existing orchestration infrastructure:

Integration	Description	Documentation
Airflow	Workflow orchestration platform	Examples
Dagster	Data orchestrator	Examples
Prefect	Workflow orchestration	Examples
Kedro	Data science pipelines	Examples
Metaflow	ML infrastructure	Integration
dbt	Data transformation tool	Integration Guide

Data Engineering & ETL

Tools for building robust data pipelines:

Integration	Description	Documentation
dlt	Data loading and transformation	Integration Guide

Integration	Description	Documentation
Feast	Feature store	Examples
FastAPI	Web service framework	Integration Guide
Streamlit	Interactive web applications	Integration Guide

Observability & Monitoring

Track and monitor your Apache Hamilton dataflows:

Integration	Description	Documentation
Datadog	Monitoring and analytics	Lifecycle Hook
OpenTelemetry	Observability framework	Examples
OpenLineage	Data lineage tracking	Examples | Lifecycle Hook
Hamilton UI	Built-in execution tracking	UI Guide
Experiment Manager	Lightweight experiment tracking	Examples

Visualization

Create visualizations from your dataflows:

Integration	Description	Documentation
Plotly	Interactive plotting	Examples
Matplotlib	Static plotting	IO Adapters
Rich	Terminal formatting and progress	Lifecycle Hook

Developer Tools

Improve your development workflow:

Integration	Description	Documentation
Jupyter	Notebook magic commands	Examples
VS Code	Language server and extension	VS Code Guide
tqdm	Progress bars	Lifecycle Hook

Cloud Providers & Infrastructure

Deploy Apache Hamilton to the cloud:

Integration	Description	Documentation
AWS	Amazon Web Services	Examples
Google Cloud	Google Cloud Platform	Scale-up Guide

Integration	Description	Documentation
Modal	Serverless cloud functions	Scale-up Guide

Storage & Caching

Persist and cache your data:

Integration	Description	Documentation
DiskCache	Disk-based caching	Examples
File-based caching	Local file caching	Caching Guide

Other Utilities

Integration	Description	Documentation
Slack	Notifications and integrations	Examples | Lifecycle Hook
GeoPandas	Geospatial data analysis	Type extension for GeoDataFrame support
YAML	Configuration management	IO Adapters

External Resources

The following resources and services are provided by third parties and the broader Apache Hamilton community.

⚠️ **Important Notice:**

These resources and services are **not maintained, nor endorsed** by the Apache Hamilton Community and Apache Hamilton project (maintained by the Committers and the Apache Hamilton PMC). Use them at your sole discretion. The community does not verify the licenses nor validity of these tools, so it's your responsibility to verify them.

Community Resources

📚 Dataflow Hub

hub.dagworks.io

A repository of reusable Apache Hamilton dataflows contributed by the community. Browse and download pre-built dataflows for common use cases.

Note: It's WIP to move the domain to be under Apache. DAGWorks Inc., which donated Hamilton, is not an operating entity anymore.

📝 Blog & Tutorials

blog.dagworks.io

Articles covering Apache Hamilton use cases, design patterns, reference architectures, and best practices.

Note: It's WIP to move the contents to be under Apache. DAGWorks Inc., which donated Hamilton, is not an operating entity anymore.

🎥 Video Content

YouTube @DAGWorks-Inc

Video tutorials, talks, and meetup recordings about Apache Hamilton.

Note: It's WIP to move the contents to be under Apache. DAGWorks Inc., which donated Hamilton, is not an operating entity anymore.

Contributing to the Ecosystem

Adding a New Integration

If you've created a plugin or integration for Apache Hamilton, we'd love to include it in our ecosystem!

For Built-in Integrations (maintained by the Apache Hamilton project):

1. Create a plugin in the `hamilton/plugins/` directory
2. Add documentation and examples
3. Submit a pull request to the Apache Hamilton repository
4. Follow the contribution guidelines

For External Resources (maintained by third parties):

1. Submit a pull request to add your resource to this page under "External Resources"
2. Include a clear description and link
3. Ensure your resource is relevant to Apache Hamilton users
4. Your resource must be properly licensed and actively maintained

Support & Questions

- 💬 Slack Community - Real-time chat and community support
- 🐛 GitHub Issues - Bug reports and feature requests
- 📖 Documentation - Comprehensive guides and API reference
- 📧 **Mailing List** - Join the Apache Hamilton users mailing list for discussions and announcements
  - **How to Subscribe**: Send an empty email to users-subscribe@hamilton.apache.org. Use a subject line like "subscribe" to avoid spam filters. Await a confirmation message and follow the instructions to complete the process.
  - **How to Unsubscribe**: Send an empty message to users-unsubscribe@hamilton.apache.org from the same email address used to subscribe.
  - **How to Post**: Once subscribed, post messages to users@hamilton.apache.org
  - **Archives**: View past discussions

Stay Updated

- ⭐ Star us on GitHub

- 🐦 Follow @hamilton_os on Twitter/X
- 📧 Join the mailing lists for announcements

Decorators

Apache Hamilton implements several decorators to promote business-logic deduplication, configuratibility, and add a layer of capabilities. These decorators can be found in the `hamilton.function_modifiers` submodule GitHub.

Custom Decorators

If you have a use case for a custom decorator, tell us on Slack or via a GitHub issues. Knowing about your use case and talking through help ensures we aren't duplicating effort, and that it'll be using part of the API we don't intend to change.

Reference

check_output*

The `@check_output` decorator enables you to add simple data quality checks to your code.

For example:

```
import pandas as pd
import numpy as np
from hamilton.function_modifiers import check_output

@check_output(
    data_type=np.int64,
    range=(0,100),
)
def some_int_data_between_0_and_100() -> pd.Series:
    pass
```

The check_output validator takes in arguments that each correspond to one of the default validators. These arguments tell it to add the default validator to the list. The above thus creates two validators, one that checks the datatype of the series, and one that checks whether the data is in a certain range.

Note that you can also specify custom decorators using the `@check_output_custom` decorator.

See data_quality for more information on available validators and how to build custom ones.

Note we also have a plugins that allow for validation with the pandera and pydantic libraries. There are two ways to access these:

1. `@check_output(schema=pandera_schema)` or `@check_output(model=pydantic_model)`
2. `@h_pandera.check_output()` or `@h_pydantic.check_output()` on the function that declares either a typed dataframe or a pydantic model.

Reference Documentation

class hamilton.function_modifiers.check_output(*importance: str = 'warn'*, *default_validator_candidates: List[Type[BaseDefaultValidator]] = None*, *target_: str | Collection[str] | None | EllipsisType = None*, ***default_validator_kwargs: Any*)

The `@check_output` decorator enables you to add simple data quality checks to your code.

For example:

```
import pandas as pd
import numpy as np
from hamilton.function_modifiers import check_output

@check_output(
    data_type=np.int64,
    data_in_range=(0,100),
    importance="warn",
)
def some_int_data_between_0_and_100() -> pd.Series:
    ...
```

The check_output decorator takes in arguments that each correspond to one of the default validators. These arguments tell it to add the default validator to the list. The above thus creates two validators, one that checks the datatype of the series, and one that checks whether the data is in a certain range.

Pandera example that shows how to use the check_output decorator with a Pandera schema:

```
import pandas as pd
import pandera as pa
from hamilton.function_modifiers import check_output
from hamilton.function_modifiers import extract_columns

schema = pa.DataFrameSchema(...)

@extract_columns('col1', 'col2')
```

```
@check_output(schema=schema, target_="builds_dataframe",
importance="fail")
def builds_dataframe(...) -> pd.DataFrame:
    ...
```

__init__(*importance: str = 'warn'*, *default_validator_candidates: List[Type[BaseDefaultValidator]] = None*, *target_: str | Collection[str] | None | EllipsisType = None*, ***default_validator_kwargs: Any*)

Creates the check_output validator.

This constructs the default validator class.

Note: that this creates a whole set of default validators. TODO – enable construction of custom validators using check_output.custom(*validators).

Parameters:

- **importance** – For the default validator, how important is it that this passes.
- **default_validator_candidates** – List of validators to be considerred for this check.
- **default_validator_kwargs** – keyword arguments to be passed to the validator.
- **target_** – a target specifying which nodes to decorate. See the docs in check_output_custom for a quick overview and the docs in function_modifiers.base.NodeTransformer for more detail.

class hamilton.function_modifiers.check_output_custom(**validators: DataValidator*, *target_: str | Collection[str] | None | EllipsisType = None*)

Class to use if you want to implement your own custom validators.

Come chat to us in slack if you're interested in this!

__init__(**validators: DataValidator*, *target_: str | Collection[str] | None | EllipsisType = None*)

Creates a check_output_custom decorator. This allows passing of custom validators that implement the DataValidator interface.

Parameters:

- **validators** – Validator to use.

- **target_** –

  The nodes to check the output of. For more detail read the docs in function_modifiers.base.NodeTransformer, but your options are:

  1. **None**: This will check just the "final node" (the node that is returned by the decorated function).
  2. **... (Ellipsis)**: This will check all nodes in the subDAG created by this.
  3. **string**: This will check the node with the given name.
  4. **Collection[str]**: This will check all nodes specified in the list.

  In all likelihood, you *don't* want `...`, but the others are useful.

  Note: you cannot stack *@check_output_custom* decorators. If you want to use multiple custom validators, you should pass them all in as arguments to a single *@check_output_custom* decorator.

class hamilton.plugins.h_pandera.check_output(*importance: str = 'warn'*, *target: str | Collection[str] | None | EllipsisType = None*)

__init__(*importance: str = 'warn'*, *target: str | Collection[str] | None | EllipsisType = None*)

Specific output-checker for pandera schemas. This decorator utilizes the output type of the function, which has to be of type pandera.typing.pandas.DataFrame or pandera.typing.pandas.Series, with an annotation argument.

Parameters:

- **schema** – The schema to use for validation. If this is not provided, then the output type of the function is used.
- **importance** – Importance level (either "warn" or "fail") – see documentation for check_output for more details.
- **target** – The target of the decorator – see documentation for check_output for more details.

Let's look at equivalent examples to demonstrate:

```
import pandera as pa
import pandas as pd
from hamilton.plugins import h_pandera
from pandera.typing.pandas import DataFrame


class MySchema(pa.DataFrameModel):
    a: int
    b: float
    c: str = pa.Field(nullable=True)  # For example, allow
None values
    d: float  # US dollars


@h_pandera.check_output()
def foo() -> DataFrame[MySchema]:
    return pd.DataFrame()  # will fail
```

```
from hamilton import function_modifiers

schema = pa.DataFrameSchema({
    "a": pa.Column(pa.Int),
    "b": pa.Column(pa.Float),
    "c": pa.Column(pa.String, nullable=True),
    "d": pa.Column(pa.Float),
})


@function_modifiers.check_output(schema=schema)
def foo() -> pd.DataFrame:
    return pd.DataFrame()  # will fail
```

These two are functionally equivalent. Note that we do not (yet) support modification of the output.

class hamilton.plugins.h_pydantic.check_output(*importance: str = 'warn'*, *target: str | Collection[str] | None | EllipsisType = None*)

__init__(*importance: str = 'warn'*, *target: str | Collection[str] | None | EllipsisType = None*)

Specific output-checker for pydantic models (requires `pydantic>=2.0`). This decorator utilizes the output type of the function, which can be any subclass of pydantic.BaseModel. The function output must be declared with a type hint.

Parameters:

- **model** – The pydantic model to use for validation. If this is not provided, then the output type of the function is used.
- **importance** – Importance level (either "warn" or "fail") – see documentation for check_output for more details.
- **target** – The target of the decorator – see documentation for check_output for more details.

Here is an example of how to use this decorator with a function that returns a pydantic model:

```
from hamilton.plugins import h_pydantic
from pydantic import BaseModel

class MyModel(BaseModel):
    a: int
    b: float
    c: str

@h_pydantic.check_output()
def foo() -> MyModel:
    return MyModel(a=1, b=2.0, c="hello")
```

Alternatively, you can return a dictionary from the function (type checkers will probably complain about this):

```
from hamilton.plugins import h_pydantic
from pydantic import BaseModel

class MyModel(BaseModel):
    a: int
    b: float
    c: str

@h_pydantic.check_output()
def foo() -> MyModel:
    return {"a": 1, "b": 2.0, "c": "hello"}
```

You can also use pydantic validation through `function_modifiers.check_output` by providing the model as an argument:

```python
from typing import Any

from hamilton import function_modifiers
from pydantic import BaseModel

class MyModel(BaseModel):
    a: int
    b: float
    c: str

@function_modifiers.check_output(model=MyModel)
def foo() -> dict[str, Any]:
    return {"a": 1, "b": 2.0, "c": "hello"}
```

Note, that because we do not (yet) support modification of the output, the validation is performed in strict mode, meaning that no data coercion is performed. For example, the following function will *fail* validation:

```python
from hamilton.plugins import h_pydantic
from pydantic import BaseModel

class MyModel(BaseModel):
    a: int  # Defined as an int

@h_pydantic.check_output()  # This will fail validation!
def foo() -> MyModel:
    return MyModel(a="1")  # Assigned as a string
```

For more information about strict mode see the pydantic docs: https://docs.pydantic.dev/latest/concepts/strict_mode/

config.when*

`@config.when` allows you to specify different implementations depending on configuration parameters.

Note the following:

- The function cannot have the same name in the same file (or python gets unhappy), so we name it with a __ (dunderscore) as a suffix. The dunderscore is removed before it goes into the DAG.
- There is currently no `@config.otherwise(...)` decorator, so make sure to have `config.when` specify set of configuration possibilities. Any missing cases will not have that output column

(and subsequent downstream nodes may error out if they ask for it). To make this easier, we have a few more `@config` decorators:

- `@config.when_not(param=value)` Will be included if the parameter is _not_ equal to the value specified.
- `@config.when_in(param=[value1, value2, ...])` Will be included if the parameter is equal to one of the specified values.
- `@config.when_not_in(param=[value1, value2, ...])` Will be included if the parameter is not equal to any of the specified values.
- `@config` If you're feeling adventurous, you can pass in a lambda function that takes in the entire configuration and resolves to `True` or `False`. You probably don't want to do this.

- To always exclude a function (such as helper functions) from the DAG the most straightforward and preferred pattern is to prefix it with "_", but you can also use `@hamilton_exclude`.

Reference Documentation

class hamilton.function_modifiers.config(*resolves: Callable[[Dict[str, Any]], bool]*, *target_name: str = None*, *config_used: List[str] = None*)

Decorator class that determines whether a function should be in the DAG based on some configuration variable.

Notes:

1. Currently, functions that exist in all configurations have to be disjoint.
2. There is currently no `@config.otherwise(...)` decorator, so make sure to have `config.when` specify set of configuration possibilities. Any missing cases will not have that output (and subsequent downstream functions may error out if they ask for it).
3. **To make this easier, we have a few more `@config` decorators:**
    - `@config.when_not(param=value)` Will be included if the parameter is _not_ equal to the value specified.
    - `@config.when_in(param=[value1, value2, ...])` Will be included if the parameter is equal to one of the specified values.
    - `@config.when_not_in(param=[value1, value2, ...])` Will be included if the parameter is not equal to any of the specified values.
    - `@config` If you're feeling adventurous, you can pass in a lambda function that takes in the entire configuration and resolves to `True` or `False`. You probably don't want to do this.

Example:

```
@config.when_in(business_line=["mens","kids"], region=["uk"])
def LEAD_LOG_BASS_MODEL_TIMES_TREND(
     TREND_BSTS_WOMENS_ACQUISITIONS: pd.Series,
     LEAD_LOG_BASS_MODEL_SIGNUPS_NON_REFERRAL: pd.Series) ->
pd.Series:
     # logic
     ...
```

Example - use of *__suffix* to differentiate between functions with the same name. This is required if you want to use the same function name in multiple configurations. Hamilton will automatically drop the suffix for you. The following will ensure only one function is registered with the name *my_transform*:

```
@config.when(region="us")
def my_transform__us(some_input: pd.Series, some_input_b:
pd.Series) -> pd.Series:
     # logic
     ...

@config.when(region="uk")
def my_transform__uk(some_input: pd.Series, some_input_c:
pd.Series) -> pd.Series:
     # logic
     ...
```

`@config` If you're feeling adventurous, you can pass in a lambda function that takes in the entire configuration and resolves to `True` or `False`. You probably don't want to do this.

__init__(*resolves: Callable[[Dict[str, Any]], bool]*, *target_name: str = None*, *config_used: List[str] = None*)

Decorator that resolves a function based on the configuration...

Parameters:

- **resolves** – the python function to use to resolve whether the wrapped function should exist in the graph or not.
- **target_name** – Optional. The name of the "function"/"node" that we want to attach @config to.
- **config_used** – Optional. The list of config names that this function uses.

class hamilton.function_modifiers.configuration.hamilton_exclude

Decorator class that excludes a function from the DAG.

The preferred way to hide functions from the Hamilton DAG is to prefix them with "_". However, for the exceptional case, it can be useful for decorating helper functions without the need to prefix them with "_" and use them either inside other nodes or in conjunction with `step` or `apply_to`.

```
@hamilton_exclude
def helper(...) -> ...:
    '''This will not be part of the DAG'''
    ...
```

You may also want to use this decorator for excluding functions in legacy code that would raise and error in Hamilton (for example missing type hints).

dataloader

Reference Documentation

class hamilton.function_modifiers.dataloader

Decorator for specifying a data loading function within the Hamilton framework. This decorator is used to annotate functions that load data, allowing them to be treated specially in the Hamilton DAG (Directed Acyclic Graph). The decorated function should return a tuple containing the loaded data and a dictionary of metadata about the loading process.

The *dataloader* decorator captures loading data metadata and ensures the function's return type is correctly annotated to be a tuple, where the first element is the loaded data and the second element is a dictionary containing metadata about the data loading process.

Downstream functions need only to depend on the type of data loaded.

Example Usage:

Assuming you have a function that loads data from a JSON file and you want to expose the metadata in your Hamilton DAG to be captured in the Hamilton UI / adapters:

```
import pandas as pd
from hamilton.function_modifiers import dataloader


@dataloader()  # you need ()
def load_json_data(json_path: str = "data/my_data.json") ->
tuple[pd.DataFrame, dict]:
```

```
    '''Loads a dataframe from a JSON file.

    :return: A tuple containing two dictionaries:
        - The first dictionary contains the loaded JSON data as a
dataframe
        - The second dictionary contains metadata about the
loading process.
    '''
    # Load the data
    data = pd.read_json(json_path)

    # Metadata about the loading process
    metadata = {"source": json_path, "format": "json"}

    return data, metadata
```

generate_nodes(*fn: Callable*, *config*) → List[Node]

Generates two nodes. We have to add tags appropriately.

The first one is just the fn - with a slightly different name. The second one uses the proper function name, but only returns the first part of the tuple that the first returns.

Parameters:

· **fn**

· **config**

Returns:

validate(*fn: Callable*)

Validates that the output type is correctly annotated.

datasaver

Reference Documentation

class hamilton.function_modifiers.datasaver

Decorator for specifying a data saving function within the Hamilton framework. This decorator is used to annotate functions that save data, allowing them to be treated specially in the Hamilton DAG (Directed Acyclic Graph). The decorated function should return a dictionary containing metadata about the saving process.

The *datasaver* decorator captures saving data metadata and ensures the function's return type is correctly annotated to be a dictionary, where the dictionary contains metadata about the data saving process, that then is exposed / captures for the Hamilton UI / adapters.

Example Usage:

Assuming you have a function that saves data to a JSON file and you want to expose the metadata in your Hamilton DAG to be captured in the Hamilton UI / adapters:

```
import pandas as pd
from hamilton.function_modifiers import datasaver


@datasaver()  # you need ()
def save_json_data(data: pd.DataFrame, json_path: str = "data/
my_saved_data.json") -> dict:
    '''Saves data to a JSON file and returns metadata about the
saving process.

    :param data: The data to save.
    :param json_path: The path to save the data to.
    :return: metadata about what was saved.
    '''
    # Save the data
    with open(json_path, "w") as file:
        data.to_json(json_path)

    # Metadata about the saving process
    metadata = {"destination": json_path, "format": "json"}

    return metadata
```

This function can now be used within the Hamilton framework as a node that saves data to a JSON file. The metadata returned alongside the data can be used for logging, debugging, or any other purpose that requires information about the data saving process as it can be pulled out by the Hamilton Tracker for the Hamilton UI or other adapters.

generate_nodes(*fn: Callable*, *config*) → List[Node]

Generates same node but all this does is add tags to it. :param fn: :param config: :return:

validate(*fn: Callable*)

Validates that the function output is a dict type.

does

`@does` is a decorator that essentially allows you to run a function over all the input parameters. So you can't pass any old function to `@does`, instead the function passed has to take any amount of inputs and process them all in the same way.

```
import pandas as pd
from hamilton.function_modifiers import does
import internal_package_with_logic

def sum_series(**series: pd.Series) -> pd.Series:
    """This function takes any number of inputs and sums them all
together."""
    ...

@does(sum_series)
def D_XMAS_GC_WEIGHTED_BY_DAY(D_XMAS_GC_WEIGHTED_BY_DAY_1: pd.Series,
                              D_XMAS_GC_WEIGHTED_BY_DAY_2:
pd.Series) -> pd.Series:
    """Adds D_XMAS_GC_WEIGHTED_BY_DAY_1 and
D_XMAS_GC_WEIGHTED_BY_DAY_2"""
    pass

@does(internal_package_with_logic.identity_function)
def copy_of_x(x: pd.Series) -> pd.Series:
    """Just returns x"""
    pass
```

The example here is a function, that all that it does, is sum all the parameters together. So we can annotate it with the `@does` decorator and pass it the `sum_series` function. The `@does` decorator is currently limited to just allow functions that consist only of one argument, a generic **kwargs.

Reference Documentation

class hamilton.function_modifiers.does(*replacing_function: Callable*, ***argument_mapping: str | List[str]*)

> `@does` is a decorator that essentially allows you to run a function over all the input parameters. So you can't pass any old function to `@does`, instead the function passed has to take any amount of inputs and process them all in the same way.
>
> ```
> import pandas as pd
> from hamilton.function_modifiers import does
> import internal_package_with_logic
>
> def sum_series(**series: pd.Series) -> pd.Series:
> ```

```
    '''This function takes any number of inputs and sums them all
    together.'''
        ...

    @does(sum_series)
    def D_XMAS_GC_WEIGHTED_BY_DAY(D_XMAS_GC_WEIGHTED_BY_DAY_1:
    pd.Series,
                                  D_XMAS_GC_WEIGHTED_BY_DAY_2:
    pd.Series) -> pd.Series:
        '''Adds D_XMAS_GC_WEIGHTED_BY_DAY_1 and
    D_XMAS_GC_WEIGHTED_BY_DAY_2'''
        pass

    @does(internal_package_with_logic.identity_function)
    def copy_of_x(x: pd.Series) -> pd.Series:
        '''Just returns x'''
        pass
```

The example here is a function, that all that it does, is sum all the parameters together. So we can annotate it with the `@does` decorator and pass it the `sum_series` function. The `@does` decorator is currently limited to just allow functions that consist only of one argument, a generic **kwargs.

__init__(*replacing_function: Callable*, ***argument_mapping: str | List[str]*)

Constructor for a modifier that replaces the annotated functions functionality with something else. Right now this has a very strict validation requirements to make compliance with the framework easy.

Parameters:

- **replacing_function** – The function to replace the original function with.
- **argument_mapping** – A mapping of argument name in the replacing function to argument name in the decorating function.

unpack_fields

This decorator works on a function that outputs a tuple and unpacks its elements to make them individually available for consumption. Essentially, it expands the original function into n separate functions, each of which takes the original output tuple and, in return, outputs a specific field based on the index supplied to the `unpack_fields` decorator.

```
import pandas as pd
from hamilton.function_modifiers import unpack_fields

@unpack_fields('X_train', 'X_test', 'y_train', 'y_test')
def train_test_split_func(
    feature_matrix: np.ndarray,
    target: np.ndarray,
    test_size_fraction: float,
    shuffle_train_test_split: bool,
) -> Tuple[np.ndarray, np.ndarray, np.ndarray, np.ndarray]:
    ...  # Calculate the train-test split
    return X_train, X_test, y_train, y_test
```

The arguments to the decorator not only represent the names of the resulting fields but also determine their position in the output tuple. This means you can choose to unpack a subset of the fields or declare an indeterminate number of fields — as long as the number of requested fields does not exceed the number of elements in the output tuple.

```
import pandas as pd
from hamilton.function_modifiers import unpack_fields

@unpack_fields('X_train', 'X_test', 'y_train', 'y_test')
def train_test_split_func(
    feature_matrix: np.ndarray,
    target: np.ndarray,
    test_size_fraction: float,
    shuffle_train_test_split: bool,
) -> Tuple[np.ndarray, ...]:  # indeterminate number of fields
    ...  # Calculate the train-test split
    return X_train, X_test, y_train, y_test
```

Reference Documentation

class hamilton.function_modifiers.unpack_fields(**fields: str*)

Unpacks fields from a tuple output.

Expands a single function into the following nodes:

- 1 function that outputs the original tuple
- n functions, each of which take in the original tuple and output a specific field

The decorated function must have an return type of either *tuple* (python 3.9+) or *typing.Tuple*, and must specify either: - An explicit length tuple (e.g.`tuple[int, str]`, *typing.Tuple[int, str]*) - An indeterminate length tuple (e.g. *tuple[int, ...]*, *typing.Tuple[int, ...]*)

Parameters:

fields – Fields to unpack from the return value of the decorated function.

__init__(**fields: str*)

Initializes the node transformer to only allow a single node to be transformed. Note this passes target=None to the superclass, which means that it will only apply to the 'sink' nodes produced.

extract_columns

This works on a function that outputs a dataframe, that we want to extract the columns from and make them individually available for consumption. So it expands a single function into *n functions*, each of which take in the output dataframe and output a specific column as named in the `extract_columns` decorator.

```
import pandas as pd
from hamilton.function_modifiers import extract_columns

@extract_columns('fiscal_date', 'fiscal_week_name', 'fiscal_month',
'fiscal_quarter', 'fiscal_year')
def fiscal_columns(date_index: pd.Series, fiscal_dates:
pd.DataFrame) -> pd.DataFrame:
    """Extracts the fiscal column data.
    We want to ensure that it has the same spine as date_index.
    :param fiscal_dates: the input dataframe to extract.
    :return:
    """
    df = pd.DataFrame({'date_index': date_index},
index=date_index.index)
    merged = df.join(fiscal_dates, how='inner')
    return merged
```

Note: if you have a list of columns to extract, then when you call `@extract_columns` you should call it with an asterisk like this:

```
import pandas as pd
from hamilton.function_modifiers import extract_columns

@extract_columns(*my_list_of_column_names)
def my_func(...) -> pd.DataFrame:
    """..."""
```

Reference Documentation

class hamilton.function_modifiers.extract_columns(**columns: Tuple[str, str] | str*, *fill_with: Any = None*)

__init__(**columns: Tuple[str, str] | str*, *fill_with: Any = None*)

Constructor for a modifier that expands a single function into the following nodes:

- n functions, each of which take in the original dataframe and output a specific column
- 1 function that outputs the original dataframe

Parameters:

- **columns** – Columns to extract, that can be a list of tuples of (name, documentation) or just names.
- **fill_with** – If you want to extract a column that doesn't exist, do you want to fill it with a default value? Or do you want to error out? Leave empty/None to error out, set fill_value to dynamically create a column.

extract_fields

This works on a function that outputs a dictionary, that we want to extract the fields from and make them individually available for consumption. So it expands a single function into *n functions*, each of which take in the output dictionary and output a specific field as named in the `extract_fields` decorator.

```
import pandas as pd
from hamilton.function_modifiers import extract_columns

@function_modifiers.extract_fields(
    {'X_train': np.ndarray, 'X_test': np.ndarray, 'y_train':
np.ndarray, 'y_test': np.ndarray})
def train_test_split_func(feature_matrix: np.ndarray,
                          target: np.ndarray,
                          test_size_fraction: float,
                          shuffle_train_test_split: bool) ->
Dict[str, np.ndarray]:
    ...
    return {'X_train': ... }
```

The input to the decorator is a dictionary of `field_name` to `field_type` – this information is used for static compilation to ensure downstream uses are expecting the right type.

Reference Documentation

class hamilton.function_modifiers.extract_fields(*fields: Dict[str, Any] | List[str] | Any | None = None*, **others*, *fill_with: Any = None*)

Extracts fields from a dictionary of output.

__init__(*fields: Dict[str, Any] | List[str] | Any | None = None*, **others*, *fill_with: Any = None*)

Constructor for a modifier that expands a single function into the following nodes:

- n functions, each of which take in the original dict and output a specific field
- 1 function that outputs the original dict

Parameters:

- **fields** – Fields to extract. Can be a dict of field names to types, a list of field names, or a single field name.
- **others** – Additional fields names to extract - argument unpacking. Ignored if *fields* is a dict.
- **fill_with** – If you want to extract a field that doesn't exist, do you want to fill it with a default value? Or do you want to error out? Leave empty/None to error out, set fill_value to dynamically create a field value.

inject

Reference Documentation

class hamilton.function_modifiers.inject(***key_mapping: ParametrizedDependency*)

@inject allows you to replace parameters with values passed in. You can think of it as a *@parameterize* call that has only one parameterization, the result of which is the name of the function. See the following examples:

```
import pandas as pd
from function_modifiers import inject, source, value, group

@inject(nums=group(source('a'), value(10), source('b'),
value(2)))
def a_plus_10_plus_b_plus_2(nums: List[int]) -> int:
```

```
    return sum(nums)

This would be equivalent to:

@parameterize(
    a_plus_10_plus_b_plus_2={
        'nums': group(source('a'), value(10), source('b'),
value(2))
    })
def sum_numbers(nums: List[int]) -> int:
    return sum(nums)
```

Something to note – we currently do not support the case in which the same parameter is utilized multiple times as an injection. E.G. two lists, a list and a dict, two sources, etc...

This is considered undefined behavior, and should be avoided.

__init__(***key_mapping: ParametrizedDependency*)

Instantiates an @inject decorator with the given key_mapping.

Parameters:

key_mapping – A dictionary of string to dependency spec. This is the same as the input mapping in *@parameterize*.

load_from

Reference Documentation

class hamilton.function_modifiers.load_from

Decorator to inject externally loaded data into a function. Ideally, anything that is not a pure transform should either call this, or accept inputs from an external location.

This decorator functions by "injecting" a parameter into the function. For example, the following code will load the json file, and inject it into the function as the parameter *input_data*. Note that the path for the JSON file comes from another node called raw_data_path (which could also be passed in as an external input).

```
@load_from.json(path=source("raw_data_path"))
def raw_data(input_data: dict) -> dict:
    return input_data
```

The decorator can also be used with *value* to inject a constant value into the loader. In the following case, we use the literal value "some/path.json" as the path to the JSON file.

```
@load_from.json(path=value("some/path.json"))
def raw_data(input_data: dict) -> dict:
    return input_data
```

Note that, if neither *source* nor *value* is specified, the value will be passed in as a literal value.

```
@load_from.json(path="some/path.json")
def raw_data(input_data: dict) -> dict:
    return input_data
```

You can also utilize the *inject_* parameter in the loader if you want to inject the data into a specific param. For example, the following code will load the json file, and inject it into the function as the parameter *data*.

```
@load_from.json(path=source("raw_data_path"), inject_="data")
def raw_data(data: dict, valid_keys: List[str]) -> dict:
    return [item for item in data if item in valid_keys]
```

You can also utilize multiple data loaders with separate *inject_* parameters to load from multiple files. data loaders to a single function:

```
@load_from.json(path=source("raw_data_path"), inject_="data")
@load_from.json(path=source("raw_data_path2"), inject_="data2")
def raw_data(data: dict, data2: dict) -> dict:
    return [item for item in data if item in data2]
```

This is a highly pluggable functionality – here's the basics of how it works:

1. Every "key" (json above, but others include csv, literal, file, pickle, etc...) corresponds to a set of loader classes. For example, the json key corresponds to the JSONLoader class in default_data_loaders. They implement the classmethod *name*. Once they are registered with the central registry they pick

2. Every data loader class (which are all dataclasses) implements the *load_targets* method, which returns a list of types it can load to. For example, the JSONLoader class can load data of type *dict*. Note that the set of potential loading candidate classes are evaluated in reverse order, so the most recently registered loader class is the one that is used. That way, you can register custom ones.

3. The loader class is instantiated with the kwargs passed to the decorator. For example, the JSONLoader class takes a *path* kwarg, which is the path to the JSON file.

4. The decorator then creates a node that loads the data, and modifies the node that runs the function to accept that. It also returns metadata (customizable at the loader-class-level) to enable debugging after the fact. This is unstructured, but can be used down the line to describe any metadata to help debug.

The "core" hamilton library contains a few basic data loaders that can be implemented within the confines of python's standard library. pandas_extensions contains a few more that require pandas to be installed.

Note that these can have *default* arguments, specified by defaults in the dataclass fields. For the full set of "keys" and "types" (e.g. load_from.json, etc...), look for all classes that inherit from *DataLoader* in the hamilton library. We plan to improve documentation shortly to make this discoverable.

__init__()

parameterize

Expands a single function into n, each of which correspond to a function in which the parameter value is replaced either by:

1. A specified value `value()`

2. The value from a specified upstream node `source()`.

Note if you're confused by the other *@paramterize_** decorators, don't worry, they all delegate to this base decorator.

```
import pandas as pd
from hamilton.function_modifiers import parameterize
from hamilton.function_modifiers import value, source


@parameterize(

D_ELECTION_2016_shifted=dict(n_off_date=source('D_ELECTION_2016'),
shift_by=value(3)),
    SOME_OUTPUT_NAME=dict(n_off_date=source('SOME_INPUT_NAME'),
shift_by=value(1)),
)
def date_shifter(n_off_date: pd.Series, shift_by: int = 1) ->
pd.Series:
```

```
    """{one_off_date} shifted by shift_by to create {output_name}"""
    return n_off_date.shift(shift_by)
```

By choosing `value()` or `source()`, you can determine the source of your dependency. Note that you can also pass documentation. If you don't, it will use the parameterized docstring.

```
@parameterize(

D_ELECTION_2016_shifted=(dict(n_off_date=source('D_ELECTION_2016'),
shift_by=value(3)), "D_ELECTION_2016 shifted by 3"),
    SOME_OUTPUT_NAME=(dict(n_off_date=source('SOME_INPUT_NAME'),
shift_by=value(1)),"SOME_INPUT_NAME shifted by 1")
)
def date_shifter(n_off_date: pd.Series, shift_by: int=1) ->
pd.Series:
    """{one_off_date} shifted by shift_by to create {output_name}"""
    return n_off_date.shift(shift_by)
```

Reference Documentation

Classes to help with @parameterize (also can be used with @inject):

class hamilton.function_modifiers.ParameterizedExtract(*outputs: Tuple[str, ...]*, *input_mapping: Dict[str, ParametrizedDependency]*)

Dataclass to hold inputs for @parameterize and @parameterize_extract_columns.

Parameters:

- **outputs** – A tuple of strings, each of which is the name of an output.
- **input_mapping** – A dictionary of string to ParametrizedDependency. The string is the name of the python parameter of the decorated function, and the value is a "source"/"value" which will be passed as input for that parameter to the function.

class hamilton.function_modifiers.source(*dependency_on: Any*)

Specifies that a parameterized dependency comes from an *upstream* source.

This means that it comes from a node somewhere else. E.G. source("foo") means that it should be assigned the value that "foo" outputs.

Parameters:

dependency_on – Upstream function (i.e. node) to come from.

Returns:

An UpstreamDependency object – a signifier to the internal framework of the dependency type.

class hamilton.function_modifiers.value(*literal_value: Any*)

Specifies that a parameterized dependency comes from a "literal" source.

E.G. value("foo") means that the value is actually the string value "foo".

Parameters:

literal_value – Python literal value to use.

Returns:

A LiteralDependency object – a signifier to the internal framework of the dependency type.

class hamilton.function_modifiers.group(**dependency_args: ParametrizedDependency*, ***dependency_kwargs: ParametrizedDependency*)

Specifies that a parameterized dependency comes from a "grouped" source.

This means that it gets injected into a list of dependencies that are grouped together. E.G. dep=group(source("foo"), source("bar")) for the function:

```
@inject(dep=group(source("foo"), source("bar")))
def f(dep: List[pd.Series]) -> pd.Series:
    return ...
```

Would result in dep getting foo and bar dependencies injected.

Parameters:

- **dependency_args** – Dependencies, list of dependencies (e.g. source("foo"), source("bar"))
- **dependency_kwargs** – Dependencies, kwarg dependencies (e.g. foo=source("foo"))

Returns:

Parameterize documentation:

class hamilton.function_modifiers.parameterize(**parametrization: Dict[str, ParametrizedDependency] | Tuple[Dict[str, ParametrizedDependency], str]*)

Decorator to use to create many functions.

Expands a single function into n, each of which correspond to a function in which the parameter value is replaced either by:

1. A specified literal value, denoted value('literal_value').
2. The output from a specified upstream function (i.e. node), denoted source('upstream_function_name').

Note that `parameterize` can take the place of `@parameterize_sources` or `@parameterize_values` decorators below. In fact, they delegate to this!

Examples expressing different syntax:

```
@parameterize(
    # tuple of assignments (consisting of literals/upstream
specifications), and docstring.
    replace_no_parameters=({}, 'fn with no parameters replaced'),
)
def no_param_function() -> Any:
    ...

@parameterize(
    # tuple of assignments (consisting of literals/upstream
specifications), and docstring.
    replace_just_upstream_parameter=(
        {'upstream_source': source('foo_source')},
        'fn with upstream_parameter set to node foo'
    ),
)
def param_is_upstream_function(upstream_source: Any) -> Any:
    '''Doc string that can also be parameterized:
{upstream_source}.'''
    ...

@parameterize(
    replace_just_literal_parameter={'literal_parameter':
value('bar')},
)
def param_is_literal_value(literal_parameter: Any) -> Any:
    '''Doc string that can also be parameterized:
```

```
{literal_parameter}.'''
    ...

@parameterize(
    replace_both_parameters={
        'upstream_parameter': source('foo_source'),
        'literal_parameter': value('bar')
    }
)
def concat(upstream_parameter: Any, literal_parameter: str) ->
Any:
    '''Adding {literal_parameter} to {upstream_parameter} to
create {output_name}.'''
    return upstream_parameter + literal_parameter
```

You also have the capability to "group" parameters, which will combine them into a list.

```
@parameterize(
    a_plus_b_plus_c={
        'to_concat' : group(source('a'), value('b'), source('c'))
    }
)
def concat(to_concat: List[str]) -> Any:
    '''Adding {literal_parameter} to {upstream_parameter} to
create {output_name}.'''
    return sum(to_concat, '')
```

__init__(***parametrization: Dict[str, ParametrizedDependency] | Tuple[Dict[str, ParametrizedDependency], str]*)

Decorator to use to create many functions.

Parameters:

parametrization –

***kwargs* with one of two things:

- a tuple of assignments (consisting of literals/ upstream specifications), and docstring.
- just assignments, in which case it parametrizes the existing docstring.

parameterize_extract_columns

Reference Documentation

class hamilton.function_modifiers.parameterize_extract_columns(**extract_config: ParameterizedExtract*, *reassign_columns: bool = True*)

@parameterize_extract_columns gives you the power of both *@extract_columns* and *@parameterize* in one decorator.

It takes in a list of *Parameterized_Extract* objects, each of which is composed of: 1. A list of columns to extract, and 2. A parameterization that gets used

In the following case, we produce four columns, two for each parameterization:

```
import pandas as pd
from function_modifiers import parameterize_extract_columns,
ParameterizedExtract, source, value
@parameterize_extract_columns(
    ParameterizedExtract(
        ("outseries1a", "outseries2a"),
        {"input1": source("inseries1a"), "input2":
source("inseries1b"), "input3": value(10)},
    ),
    ParameterizedExtract(
        ("outseries1b", "outseries2b"),
        {"input1": source("inseries2a"), "input2":
source("inseries2b"), "input3": value(100)},
    ),
)
def fn(input1: pd.Series, input2: pd.Series, input3: float) ->
pd.DataFrame:
    return pd.concat([input1 * input2 * input3, input1 + input2
+ input3], axis=1)
```

__init__(**extract_config: ParameterizedExtract*, *reassign_columns: bool = True*)

Initializes a *parameterized_extract* decorator. Note this currently works for series, but the plan is to extend it to fields as well...

Parameters:

- **extract_config** – A configuration consisting of a list ParameterizedExtract classes These contain the information of a *@parameterized* and *@extract...* together.

- **reassign_columns** – Whether we want to reassign the columns as part of the function.

parameterize_frame

Reference Documentation

class hamilton.experimental.decorators.parameterize_frame.parameterize_frame(*parameterization: DataFrame*)

EXPERIMENTAL! Instantiates a parameterize_extract decorator using a dataframe to specify a set of extracts + parameterizations.

This is an experimental decorator and the API may change in the future; please provide feedback whether this API does or does not work for you.

Parameters:

parameterization – Parameterization dataframe. See below.

This is of a specific shape:

1. Index - Level 0: list of parameter names
2. Index - Level 1: types of things to inject, either:
    - "out" (meaning this is an output),
    - "value" (meaning this is a literal value)
    - "source" (meaning this node comes from an upstream value)
3. Contents:
    - Each row corresponds to the index. Each of these corresponds to an output node from this.

Note your function has to take in the column-names and output a dataframe with those names – we will likely change it so that's not the case, and it can just use the position of the columns.

Example usage:

```
from hamilton.experimental.decorators.parameterize_frame import
parameterize_frame
df = pd.DataFrame(
[
   ["outseries1a", "outseries2a", "inseries1a", "inseries2a",
5.0],
   ["outseries1b", "outseries2b", "inseries1b", "inseries2b",
0.2],
],
# specify column names corresponding to function arguments and
# if outputting multiple columns, output dataframe columns.
columns=[
   ["output1", "output2", "input1", "input2", "input3"],
   ["out", "out", "source", "source", "value"],
])

@parameterize_frame(df)
def my_func(
    input1: pd.Series, input2: pd.Series, input3: float
) -> pd.DataFrame:
   ...
```

__init__(*parameterization: DataFrame*)

Initializes a *parameterized_extract* decorator. Note this currently works for series, but the plan is to extend it to fields as well...

Parameters:

- **extract_config** – A configuration consisting of a list ParameterizedExtract classes These contain the information of a *@parameterized* and *@extract...* together.
- **reassign_columns** – Whether we want to reassign the columns as part of the function.

parameterize_sources

Expands a single function into *n*, each of which corresponds to a function in which the parameters specified are mapped to the specified inputs. Note this decorator and `@parameterize_values` are quite similar, except that the input here is another DAG node(s), i.e. column/input, rather than a specific scalar/static value.

```
import pandas as pd
from hamilton.function_modifiers import parameterize_sources

@parameterize_sources(
    D_ELECTION_2016_shifted=dict(one_off_date='D_ELECTION_2016'),
    SOME_OUTPUT_NAME=dict(one_off_date='SOME_INPUT_NAME')
)
def date_shifter(one_off_date: pd.Series) -> pd.Series:
    """{one_off_date} shifted by 1 to create {output_name}"""
    return one_off_date.shift(1)
```

We see here that `parameterize_sources` allows you to keep your code DRY by reusing the same function to create multiple distinct outputs. The key word arguments passed have to have the following structure:

```
OUTPUT_NAME = Mapping of function argument to input that should go
into it.
```

So in the example, `D_ELECTION_2016_shifted` is an _output_ that will correspond to replacing `one_off_date` with `D_ELECTION_2016`. Then similarly `SOME_OUTPUT_NAME` is an _output_ that will correspond to replacing `one_off_date` with `SOME_INPUT_NAME`. The documentation for both uses the same function doc and will replace values that are templatized with the input parameter names, and the reserved value `output_name`.

To help visualize what the above is doing, it is equivalent to writing the following two function definitions:

```
def D_ELECTION_2016_shifted(D_ELECTION_2016: pd.Series) -> pd.Series:
    """D_ELECTION_2016 shifted by 1 to create
D_ELECTION_2016_shifted"""
    return D_ELECTION_2016.shift(1)

def SOME_OUTPUT_NAME(SOME_INPUT_NAME: pd.Series) -> pd.Series:
    """SOME_INPUT_NAME shifted by 1 to create SOME_OUTPUT_NAME"""
    return SOME_INPUT_NAME.shift(1)
```

Note: that the different input variables must all have compatible types with the original decorated input variable.

Reference Documentation

class hamilton.function_modifiers.parameterize_sources(***parameterization: Dict[str, str]*)

Expands a single function into *n*, each of which corresponds to a function in which the parameters specified are mapped to the specified inputs. Note this decorator and

`@parameterize_values` are quite similar, except that the input here is another DAG node(s), i.e. column/input, rather than a specific scalar/static value.

```
import pandas as pd
from hamilton.function_modifiers import parameterize_sources

@parameterize_sources(
   D_ELECTION_2016_shifted=dict(one_off_date='D_ELECTION_2016'),
   SOME_OUTPUT_NAME=dict(one_off_date='SOME_INPUT_NAME')
)
def date_shifter(one_off_date: pd.Series) -> pd.Series:
   '''{one_off_date} shifted by 1 to create {output_name}'''
   return one_off_date.shift(1)
```

__init__(***parameterization: Dict[str, str]*)

Constructor for a modifier that expands a single function into n, each of which corresponds to replacing some subset of the specified parameters with specific upstream nodes.

Note this decorator and *@parametrized_input* are similar, except this one allows multiple parameters to be mapped to multiple function arguments (and it fixes the spelling mistake).

parameterized_sources **allows you keep your code DRY by reusing the same function but replace the inputs to create multiple corresponding distinct outputs. We see here that** *parameterized_inputs* **allows you to keep your code DRY by reusing the same function to create multiple distinct outputs. The key word arguments passed have to have the following structure:**

> OUTPUT_NAME = Mapping of function argument to input that should go into it.

The documentation for the output is taken from the function. The documentation string can be templatized with the parameter names of the function and the reserved value *output_name* - those will be replaced with the corresponding values from the parameterization.

Parameters:

****parameterization** – kwargs of output name to dict of parameter mappings.

Note: this was previously called *@parameterized_inputs*.

parameterized_subdag

Reference Documentation

class hamilton.function_modifiers.parameterized_subdag(**load_from: ModuleType | Callable*, *inputs: Dict[str, ParametrizedDependency | LiteralDependency] = None*, *config: Dict[str, Any] = None*, *external_inputs: List[str] = None*, ***parameterization: SubdagParams*)

parameterized subdag is when you want to create multiple subdags at one time. Why might you want to do this?

1. You have multiple data sets you want to run the same feature engineering pipeline on.
2. You want to run some sort of optimization routine with a variety of results
3. You want to run some sort of pipeline over slightly different configuration (E.G. region/ business line)

Note that this really is just syntactic sugar for creating multiple subdags, just as *@parameterize is syntactic sugar for creating multiple nodes from a function. That said, it is common that you won't know what you want until compile time (E.G. when you have the config available), so this decorator along with the `@resolve* decorator is a good way to make that feasible. Note that we are getting into *advanced* Hamilton here – we don't recommend starting with this. In fact, we generally recommend repeating subdags multiple times if you don't have too many. That said, that can get cumbersome if you have a lot, so this decorator is a good way to help with that.

Let's take a look at an example:

```
@parameterized_subdag(
    feature_modules,
    from_datasource_1={"inputs" : {"data" :
value("datasource_1.csv")}},
    from_datasource_2={"inputs" : {"data" :
value("datasource_2.csv")}},
    from_datasource_3={
        "inputs" : {"data" : value("datasource_3.csv")},
        "config" : {"filter" : "only_even_client_ids"}
    }
)
def feature_engineering(feature_df: pd.DataFrame) ->
pd.DataFrame:
    return feature_df
```

This is (obviously) contrived, but what it does is create three subdags, each with a different data source. The third one also applies a configuration to that subdags. Note that we can also pass in inputs/config to the decorator itself, which will be applied to all subdags.

This is effectively the same as the example above.

```
@parameterized_subdag(
    feature_modules,
    inputs={"data" : value("datasource_1.csv")},
    from_datasource_1={},
    from_datasource_2={
            "inputs" : {"data" : value("datasource_2.csv")}
    },
    from_datasource_3={
            "inputs" : {"data" : value("datasource_3.csv")},
            "config" : {"filter" : "only_even_client_ids"},
    }
)
```

Again, think about whether this feature is really the one you want – often times, verbose, static DAGs are far more readable than very concise, highly parameterized DAGs.

__init__(*load_from: ModuleType | Callable*, *inputs: Dict[str, ParametrizedDependency | LiteralDependency] = None*, *config: Dict[str, Any] = None*, *external_inputs: List[str] = None*, ***parameterization: SubdagParams*)

Initializes a parameterized_subdag decorator.

Parameters:

- **load_from** – Modules to load from
- **inputs** – Inputs for each subdag generated by the decorated function
- **config** – Config for each subdag generated by the decorated function
- **external_inputs** – External inputs to all parameterized subdags. Note that if you pass in any external inputs from local subdags, it overrides this (does not merge).
- **parameterization** –

  Parameterizations for each subdag generated. Note that this *overrides* any inputs/config passed to the decorator itself.

  Furthermore, note the following:

  1. The parameterizations passed to the constructor are **kwargs, so you are not allowed to name these

load_from, *inputs*, or *config*. That's a good thing, as these are not good names for variables anyway.

2. Any empty items (not included) will default to an empty dict (or an empty list in the case of parameterization)

parameterize_values

Expands a single function into n, each of which corresponds to a function in which the parameter value is replaced by that *specific value*.

```
import pandas as pd
from hamilton.function_modifiers import parameterize_values
import internal_package_with_logic

ONE_OFF_DATES = {
     #output name        # doc string                # input value to
function
    ('D_ELECTION_2016', 'US Election 2016 Dummy'): '2016-11-12',
    ('SOME_OUTPUT_NAME', 'Doc string for this thing'):
'value to pass to function',
}
            # parameter matches the name of the argument in the
function below
@parameterize_values(parameter='one_off_date',
assigned_output=ONE_OFF_DATES)
def create_one_off_dates(date_index: pd.Series, one_off_date: str) -
> pd.Series:
    """Given a date index, produces a series where a 1 is placed at
the date index that would contain that event."""
    one_off_dates =
internal_package_with_logic.get_business_week(one_off_date)
    return
internal_package_with_logic.bool_to_int(date_index.isin([one_off_dates]))
```

We see here that `parameterize` allows you keep your code DRY by reusing the same function to create multiple distinct outputs. The *parameter* key word argument has to match one of the arguments in the function. The rest of the arguments are pulled from outside the DAG. The _assigned_output_ key word argument takes in a dictionary of tuple(Output Name, Documentation string) -> value.

Reference Documentation

class hamilton.function_modifiers.parameterize_values(*parameter: str*, *assigned_output: Dict[Tuple[str, str], Any]*)

Expands a single function into n, each of which corresponds to a function in which the parameter value is replaced by that *specific value*.

```
import pandas as pd
from hamilton.function_modifiers import parameterize_values
import internal_package_with_logic

ONE_OFF_DATES = {

#output name        # doc string               # input value to
function
    ('D_ELECTION_2016', 'US Election 2016 Dummy'): '2016-11-12',
    ('SOME_OUTPUT_NAME', 'Doc string for this thing'): 'value to
pass to function',
}
            # parameter matches the name of the argument in the
function below
@parameterize_values(parameter='one_off_date',
assigned_output=ONE_OFF_DATES)
def create_one_off_dates(date_index: pd.Series, one_off_date:
str) -> pd.Series:
    '''Given a date index, produces a series where a 1 is placed
at the date index that would contain that event.'''
    one_off_dates =
internal_package_with_logic.get_business_week(one_off_date)
    return
internal_package_with_logic.bool_to_int(date_index.isin([one_off_dates]))
```

__init__(*parameter: str*, *assigned_output: Dict[Tuple[str, str], Any]*)

Constructor for a modifier that expands a single function into n, each of which corresponds to a function in which the parameter value is replaced by that *specific value*.

Parameters:

- **parameter** – Parameter to expand on.
- **assigned_output** – A map of tuple of [parameter names, documentation] to values

Note: this was previously called *@parametrized*.

pipe family

We have a family of decorators that represent a chained set of transformations. This specifically solves the "node redefinition" problem, and is meant to represent a pipeline of chaining/redefinitions. This is similar (and can happily be used in conjunction with) `pipe` in pandas. In Pyspark this is akin to the common operation of redefining a dataframe with new columns.

For some examples have a look at: https://github.com/apache/hamilton/tree/main/examples/scikit-learn/species_distribution_modeling

While it is generally reasonable to contain constructs within a node's function, you should consider the pipe family for any of the following reasons:

1. You want the transformations to display as nodes in the DAG, with the possibility of storing or visualizing the result.

1. You want to pull in functions from an external repository, and build the DAG a little more procedurally.

3. You want to use the same function multiple times, but with different parameters – while `@does` / `@parameterize` can do this, this presents an easier way to do this, especially in a chain.

Reference Documentation

pipe

DeprecationWarning from 2.0.0: use pipe_input instead

class hamilton.function_modifiers.macros.pipe(**transforms: Applicable*, *namespace: str | EllipsisType | None = Ellipsis*, *on_input: str | Collection[str] | None | EllipsisType = None*, *collapse=False*, *_chain=False*)

__init__(**transforms: Applicable*, *namespace: str | EllipsisType | None = Ellipsis*, *on_input: str | Collection[str] | None | EllipsisType = None*, *collapse=False*, *_chain=False*)

Instantiates a `@pipe_input` decorator.

Parameters:

- **transforms** – step transformations to be applied, in order
- **namespace** – namespace to apply to all nodes in the pipe. This can be "..." (the default), which resolves to the name of the decorated function, None (which means no namespace), or a string (which means that all nodes will be namespaced with that string). Note

that you can either use this *or* namespaces inside `pipe_input()`...

- **on_input** – setting the target parameter for all steps in the pipe. Leave empty to select only the first argument.
- **collapse** – Whether to collapse this into a single node. This is not currently supported.
- **_chain** – Whether to chain the first parameter. This is the only mode that is supported. Furthermore, this is not externally exposed. `@flow` will make use of this.

pipe_input

class hamilton.function_modifiers.macros.pipe_input(**transforms: Applicable*, *namespace: str | EllipsisType | None = Ellipsis*, *on_input: str | Collection[str] | None | EllipsisType = None*, *collapse=False*, *_chain=False*)

Running a series of transformations on the input of the function.

To demonstrate the rules for chaining nodes, we'll be using the following example. This is using primitives to demonstrate, but as hamilton is just functions of any python objects, this works perfectly with dataframes, series, etc...

```
from hamilton.function_modifiers import step, pipe_input, value,
source


def _add_one(x: int) -> int:
    return x + 1


def _sum(x: int, y: int) -> int:
    return x + y


def _multiply(x: int, y: int, z: int = 10) -> int:
    return x * y * z


@pipe_input(
    step(_add_one),
    step(_multiply, y=2),
    step(_sum, y=value(3)),
    step(_multiply, y=source("upstream_node_to_multiply")),
```

```
)
def final_result(upstream_int: int) -> int:
    return upstream_int
```

```
upstream_int = ...  # result from upstream
upstream_node_to_multiply = ...  # result from upstream

output = final_result(
    _multiply(
        _sum(
            _multiply(
                _add_one(upstream_int),
                y=2
            ),
            y=3
        ),
        y=upstream_node_to_multiply
    )
)
```

```
upstream_int = ...  # result from upstream
upstream_node_to_multiply = ...  # result from upstream

one_added = _add_one(upstream_int)
multiplied = _multiply(one_added, y=2)
summed = _sum(multiplied, y=3)
multiplied_again = _multiply(summed, y=upstream_node_to_multiply)
output = final_result(multiplied_again)
```

Note that functions must have no position-only arguments (this is rare in python, but hamilton does not handle these). This basically means that the functions must be defined similarly to `def fn(x, y, z=10)` and not `def fn(x, y, /, z=10)`. In fact, all arguments must be named and "kwarg-friendly", meaning that the function can happily be called with `**kwargs`, where kwargs are some set of resolved upstream values. So, no `*args` are allowed, and `**kwargs` (variable keyword-only) are not permitted. Note that this is not a design limitation, rather an implementation detail – if you feel like you need this, please reach out.

Furthermore, the function should be typed, as a Hamilton function would be.

One has three ways to tune the shape/implementation of the subsequent nodes:

1. **`when` / `when_not` / `when_in` / `when_not_in` – these are used to filter the application of the function.**

This is valuable to reflect if/else conditions in the structure of the DAG, pulling it out of functions, rather than buried within the logic itself. It is functionally equivalent to `@config.when`.

For instance, if you want to include a function in the chain only when a config parameter is set to a certain value, you can do:

```
@pipe_input(
    step(_add_one).when(foo="bar"),
    step(_add_two,
y=source("other_node_to_add").when(foo="baz"),
)
def final_result(upstream_int: int) -> int:
    return upstream_int
```

This will only apply the first function when the config parameter `foo` is set to `bar`, and the second when it is set to `baz`.

2. **`named` – this is used to name the node. This is useful if you want to refer to intermediate results.**

   If this is left out, hamilton will automatically name the functions in a globally unique manner. The names of these functions will not necessarily be stable/guaranteed by the API, so if you want to refer to them, you should use `named`. The default namespace will always be the name of the decorated function (which will be the last node in the chain).

   `named` takes in two parameters – required is the `name` – this will assign the nodes with a single name and *no* global namespace. For instance:

   ```
   @pipe_input(
       step(_add_one).named("a"),
       step(_add_two, y=source("upstream_node")).named("b"),
   )
   def final_result(upstream_int: int) -> int:
       return upstream_int
   ```

   The above will create two nodes, `a` and `b`. `a` will be the result of `_add_one`, and `b` will be the result of `_add_two`. `final_result` will then be called with the output of `b`. Note that, if these are part of a namespaced operation (a subdag, in particular), they *will* get the same namespace as the subdag.

   The second parameter is `namespace`. This is used to specify a namespace for the node. This is useful if you want to either (a) ensure that the nodes are namespaced but share a common one to avoid name clashes (usual case), or (b) if you want a

custom namespace (unusual case). To indicate a custom namespace, one need simply pass in a string.

To indicate that a node should share a namespace with the rest of the step(...) operations in a pipe, one can pass in `...` (the ellipsis).

```
  @pipe_input(
      step(_add_one).named("a", namespace="foo"),  # foo.a
      step(_add_two, y=source("upstream_node")).named("b",
namespace=...),  # final_result.b
  )
  def final_result(upstream_int: int) -> int:
      return upstream_int
```

Note that if you pass a namespace argument to the `pipe_input` function, it will set the namespace on each step operation. This is useful if you want to ensure that all the nodes in a pipe have a common namespace, but you want to rename them.

```
@pipe_input(
    step(_add_one).named("a"), # a
    step(_add_two, y=source("upstream_node")).named("b"),
# foo.b
    namespace=..., # default -- final_result.a and
final_result.b, OR
    namespace=None, # no namespace -- a and b are exposed
as that, OR
    namespace="foo", # foo.a and foo.b
)
def final_result(upstream_int: int) -> int:
    return upstream_int
```

In all likelihood, you should not be using this, and this is only here in case you want to expose a node for consumption/output later. Setting the namespace in individual nodes as well as in `pipe_input` is not yet supported.

3. `on_input` **– this selects which input we will run the pipeline on.**
   In case `on_input` is set to None (default), we apply `pipe_input` on the first parameter. Let us know if you wish to expand to other use-cases. You can track the progress on this topic via: https://github.com/apache/hamilton/issues/1177

   The following would apply function *_add_one* and *_add_two* to `p2`:

```
@pipe_input(
    step(_add_one)
    step(_add_two, y=source("upstream_node")),
    on_input = "p2"
```

```
)
def final_result(p1: int, p2: int, p3: int) -> int:
    return upstream_int
```

__init__(**transforms: Applicable*, *namespace: str | EllipsisType | None = Ellipsis*, *on_input: str | Collection[str] | None | EllipsisType = None*, *collapse=False*, *_chain=False*)

Instantiates a `@pipe_input` decorator.

Parameters:

- **transforms** – step transformations to be applied, in order
- **namespace** – namespace to apply to all nodes in the pipe. This can be "..." (the default), which resolves to the name of the decorated function, None (which means no namespace), or a string (which means that all nodes will be namespaced with that string). Note that you can either use this *or* namespaces inside `pipe_input()`...
- **on_input** – setting the target parameter for all steps in the pipe. Leave empty to select only the first argument.
- **collapse** – Whether to collapse this into a single node. This is not currently supported.
- **_chain** – Whether to chain the first parameter. This is the only mode that is supported. Furthermore, this is not externally exposed. `@flow` will make use of this.

pipe_output

class hamilton.function_modifiers.macros.pipe_output(**transforms: Applicable*, *namespace: str | EllipsisType | None = Ellipsis*, *on_output: str | Collection[str] | None | EllipsisType = None*, *collapse=False*, *_chain=False*)

Running a series of transformation on the output of the function.

The decorated function declares the dependency, the body of the function gets executed, and then we run a series of transformations on the result of the function specified by `pipe_output`.

If we have nodes **A –> B –> C** in the DAG and decorate **B** with `pipe_output` like

```
@pipe_output(
    step(B1),
    step(B2)
)
def B(...):
    return ...
```

we obtain the new DAG **A –> B.raw –> B1 –> B2 –> B –> C**, where we can think of the **B.raw –> B1 –> B2 –> B** as a "pipe" that takes the raw output of **B**, applies to it **B1**, takes the output of **B1** applies to it **B2** and then gets renamed to **B** to re-connect to the rest of the DAG.

The rules for chaining nodes are the same as for `pipe_input`.

For extra control in case of multiple output nodes, for example after `extract_field` / `extract_columns` we can also specify the output node that we wish to mutate. The following apply *A* to all fields while *B* only to `field_1`

```
@extract_columns("col_1", "col_2")
def A(...):
    return ...

def B(...):
    return ...


 @pipe_output(
    step(A),
    step(B).on_output("field_1"),
)
@extract_fields(
        {"field_1":int, "field_2":int, "field_3":int}
)
def foo(a:int)->Dict[str,int]:
    return {"field_1":1, "field_2":2, "field_3":3}
```

We can also do this on the global level (but cannot do on both levels at the same time). The following would apply function *A* and function *B* to only `field_1` and `field_2`

```
@pipe_output(
    step(A),
    step(B),
    on_output = ["field_1","field_2]
)
@extract_fields(
        {"field_1":int, "field_2":int, "field_3":int}
)
```

```
def foo(a:int)->Dict[str,int]:
    return {"field_1":1, "field_2":2, "field_3":3}
```

__init__(*transforms: Applicable*, *namespace: str | EllipsisType | None = Ellipsis*, *on_output: str | Collection[str] | None | EllipsisType = None*, *collapse=False*, *_chain=False*)

Instantiates a `@pipe_output` decorator.

Warning: if there is a global pipe_output target, the individual `step(...).target` would only choose from the subset pre-selected from the global pipe_output target. We have disabled this for now to avoid confusion. Leave global pipe_output target empty if you want to choose between all the nodes on the individual step level.

Parameters:

- **transforms** – step transformations to be applied, in order
- **namespace** – namespace to apply to all nodes in the pipe. This can be "..." (the default), which resolves to the name of the decorated function, None (which means no namespace), or a string (which means that all nodes will be namespaced with that string). Note that you can either use this *or* namespaces inside `pipe_output()`...
- **on_output** – setting the target node for all steps in the pipe. Leave empty to select all the output nodes.
- **collapse** – Whether to collapse this into a single node. This is not currently supported.
- **_chain** – Whether to chain the first parameter. This is the only mode that is supported. Furthermore, this is not externally exposed. `@flow` will make use of this.

mutate

class hamilton.function_modifiers.macros.mutate(**target_functions: Applicable | Callable*, *collapse: bool = False*, *_chain: bool = False*, ***mutating_function_kwargs: SingleDependency | Any*)

Running a transformation on the outputs of a series of functions.

This is closely related to `pipe_output` as it effectively allows you to run transformations on the output of a node without touching that node. We choose which target functions we wish

to mutate by the transformation we are decorating. For now, the target functions, that will be mutated, have to be in the same module (come speak to us if you need this capability over multiple modules).

We suggest you define them with an prefixed underscore to only have them displayed in the *transform pipeline* of the target node.

If we wish to apply `_transform1` to the output of **A** and **B** and `_transform2` only to the output of node **B**, we can do this like

```
def A(...):
    return ...

def B(...):
    return ...

@mutate(A, B)
def _transform1(...):
    return ...

@mutate(B)
def _transform2(...):
    return ...
```

we obtain the new pipe-like subDAGs **A.raw –> _transform1 –> A** and **B.raw –> _transform1 –> _transform2 –> B**, where the behavior is the same as `pipe_output`.

While it is generally reasonable to use `pipe_output`, you should consider `mutate` in the following scenarios:

1. Loading data and applying pre-cleaning step.

2. Feature engineering via joining, filtering, sorting, etc.

3. Experimenting with different transformations across nodes by selectively turning transformations on / off.

We assume the first argument of the decorated function to be the output of the function we are targeting. For transformations with multiple arguments you can use key word arguments coupled with `step` or `value` the same as with other `pipe`-family decorators

```
@mutate(A, B, arg2=step('upstream_node'),
arg3=value(some_literal), ...)
def _transform1(output_from_target:correct_type, arg2:arg2_type,
arg3:arg3_type,...):
    return ...
```

You can also select individual args that will be applied to each target node by adding `apply_to(...)`

```
@mutate(
        apply_to(A, arg2=step('upstream_node_1'),
arg3=value(some_literal_1)),
        apply_to(B, arg2=step('upstream_node_2'),
arg3=value(some_literal_2)),
        )
def _transform1(output_from_target:correct_type, arg2:arg2_type,
arg3:arg3_type, ...):
    return ...
```

In case of multiple output nodes, for example after `extract_field` / `extract_columns` we can also specify the output node that we wish to mutate. The following would mutate all columns of *A* individually while in the case of function *B* only `field_1`

```
@extract_columns("col_1", "col_2")
def A(...):
    return ...

@extract_fields(
        {"field_1":int, "field_2":int, "field_3":int}
)
def B(...):
    return ...

@mutate(
    apply_to(A),
    apply_to(B).on_output("field_1"),
    )

def foo(a:int)->Dict[str,int]:
    return {"field_1":1, "field_2":2, "field_3":3}
```

__init__(**target_functions: Applicable | Callable*, *collapse: bool = False*, *_chain: bool = False*, ***mutating_function_kwargs: SingleDependency | Any*)

Instantiates a `mutate` decorator.

We assume the first argument of the decorated function to be the output of the function we are targeting.

Parameters:

- **target_functions** – functions we wish to mutate the output of

- **collapse** – Whether to collapse this into a single node. This is not currently supported.
- **_chain** – Whether to chain the first parameter. This is the only mode that is supported. Furthermore, this is not externally exposed. `@flow` will make use of this.
- ****mutating_function_kwargs** – other kwargs that the decorated function has. Must be validly called as `f(**kwargs)`, and have a 1-to-1 mapping of kwargs to parameters. This will be applied for all `target_functions`, unless `apply_to` already has the mutator function kwargs, in which case it takes those.

resolve

Reference Documentation

class hamilton.function_modifiers.resolve(*, *when: ResolveAt*, *decorate_with: Callable[[...], NodeTransformLifecycle]*)

Decorator class to delay evaluation of decorators until after the configuration is available. Note: this is a power-user feature, and you have to enable power-user mode! To do so, you have to add the configuration hamilton.enable_power_user_mode=True to the config you pass into the driver.

If not, this will break when it tries to instantiate a DAG.

This is particularly useful when you don't know how you want your functions to resolve until configuration time. Say, for example, we want to add two series, and we need to pass the set of series to add as a configuration parameter, as we'll be changing it regularly. Without this, you would have to have them as part of the same dataframe. E.G.

```
@parameterize_values(
    series_sum_1={"s1": "series_1", "s2": "series_2"},
    series_sum_2={"s1": "series_3", "s2": "series_4"},
)
def summation(df: pd.DataFrame, s1: str, s2: str) -> pd.Series:
    return df[s1] + df[s2]
```

Note that there are a lot of benefits to this code, but it is a workaround for the fact that we cannot configure the dependencies. With the *@resolve* decorator, we can actually dynamically set the shape of the DAG based on config:

```
from hamilton.function_modifiers import resolve, ResolveAt


@resolve(
    when=ResolveAt.CONFIG_AVAILABLE,
    decorate_with=lambda first_series_sum, second_series_sum:
parameterize_sources(
        series_sum_1={"s1": first_series_sum[0], "s2":
second_series_sum[1]},
        series_sum_2={"s1": second_series_sum[1], "s2":
second_series_sum[2]},
    ),
)
def summation(s1: pd.Series, s2: pd.Series) -> pd.Series:
    return s1 + s2
```

Note how this works:

1. The *decorate_with* argument is a function that gives you the decorator you want to apply. Currently its "hamilton-esque" – while we do not require it to be typed, you can use a separate configuration-reoslver function (and include type information). This lambda function must return a decorator.

2. The *when* argument is the point at which you want to resolve the decorator. Currently, we only support *ResolveAt.CONFIG_AVAILABLE*, which means that the decorator will be resolved at compile time, E.G. when the driver is instantiated.

1. This is then run and dynamically resolved.

This is powerful, but the code is uglier. It's meant to be used in some very specific cases, E.G. When you want time-series data on a per-column basis (E.G. once per month), and don't want that hardcoded. While it is possible to store this up in a JSON file and run parameterization on the loaded result as a global variable, it is much cleaner to pass it through the DAG, which is why we support it. However, since the code goes against one of Hamilton's primary tenets (that all code is highly readable), we require that you enable power_user_mode.

We *highly* recommend that you put all functions decorated with this in their own module, keeping it separate from the rest of your functions. This way, you can import/build DAGs from the rest of your functions without turning on power-user mode.

__init__(*, *when: ResolveAt*, *decorate_with: Callable[[...], NodeTransformLifecycle]*)

Initializes a delayed decorator that gets called at some specific resolution time.

Parameters:

- **decorate_with** – Function that takes required and optional parameters/returns a decorator.
- **when** – When to resolve the decorator. Currently only supports *ResolveAt.CONFIG_AVAILABLE*.

class hamilton.function_modifiers.resolve_from_config(*, *decorate_with: Callable[[...], NodeTransformLifecycle]*)

Decorator class to delay evaluation of decorators until after the configuration is available. Note: this is a power-user feature, and you have to enable power-user mode! To do so, you have to add the configuration hamilton.enable_power_user_mode=True to the config you pass into the driver.

This is a convenience decorator that is a subclass of *resolve* and passes *ResolveAt.CONFIG_AVAILABLE* to the *when* argument such that the decorator is resoled at compile time, E.G. when the driver is instantiated.

```
from hamilton.function_modifiers import resolve, ResolveAt


@resolve_from_config(
    decorate_with=lambda first_series_sum, second_series_sum:
parameterize_sources(
        series_sum_1={"s1": first_series_sum[0], "s2":
second_series_sum[1]},
        series_sum_2={"s1": second_series_sum[1], "s2":
second_series_sum[2]},
    )
)
def summation(s1: pd.Series, s2: pd.Series) -> pd.Series:
    return s1 + s2
```

__init__(*, *decorate_with: Callable[[...], NodeTransformLifecycle]*)

Initializes a delayed decorator that gets called at some specific resolution time.

Parameters:

- **decorate_with** – Function that takes required and optional parameters/returns a decorator.
- **when** – When to resolve the decorator. Currently only supports *ResolveAt.CONFIG_AVAILABLE*.

save_to

Reference Documentation

class hamilton.function_modifiers.save_to

Decorator that outputs data to some external source. You can think about this as the inverse of load_from.

This decorates a function, takes the final node produced by that function and then appends an additional node that saves the output of that function.

As the load_from decorator does, this decorator can be referred to in a dynamic way. For instance, @save_to.json will save the output of the function to a json file. Note that this means that the output of the function must be a dictionary (or subclass thereof), otherwise the decorator will fail.

Looking at the json example:

```
@save_to.json(path=source("raw_data_path"),
output_name_="data_save_output")
def final_output(data: dict, valid_keys: List[str]) -> dict:
    return [item for item in data if item in valid_keys]
```

This adds a final node to the DAG with the name "data_save_output" that accepts the output of the function "final_output" and saves it to a json. In this case, the JSONSaver accepts a *path* parameter, which is provided by the upstream node (or input) named "raw_data_path". The *output_name_* parameter then says how to refer to the output of this node in the DAG.

If you called this with the driver:

```
dr = driver.Driver(my_module)
output = dr.execute(["final_output"], {"raw_data_path": "/path/
my_data.json"})
```

You would *just* get the final result, and nothing would be saved.

If you called this with the driver:

```
dr = driver.Driver(my_module)
output = dr.execute(["data_save_output"], {"raw_data_path": "/
path/my_data.json"})
```

You would get a dictionary of metadata (about the saving output), and the final result would be saved to a path.

Note that you can also hardcode the path, rather than using a dependency:

```
@save_to.json(path=value("/path/my_data.json"),
output_name_="data_save_output")
def final_output(data: dict, valid_keys: List[str]) -> dict:
    return [item for item in data if item in valid_keys]
```

Note that, like the loader function, you can use literal values as kwargs and they'll get interpreted as values. If you needs savers, you should also look into *.materialize* on the driver – it's a clean way to do this in a more ad-hoc/decoupled manner.

If you want to layer savers, you'll have to use the target_ parameter, which tells the saver which node to use.

```
@save_to.json(path=source("raw_data_path"),
output_name_="data_save_output", target_="data")
@save_to.json(path=source("raw_data_path2"),
output_name_="data_save_output2", target_="data")
def final_output(data: dict, valid_keys: List[str]) -> dict:
    return [item for item in data if item in valid_keys]
```

__init__()

subdag

Reference Documentation

class hamilton.function_modifiers.subdag(**load_from: ModuleType | Callable*, *inputs: Dict[str, ParametrizedDependency] = None*, *config: Dict[str, Any] = None*, *namespace: str = None*, *final_node_name: str = None*, *external_inputs: List[str] = None*)

The *@subdag* decorator enables you to rerun components of your DAG with varying parameters. That is, it enables you to "chain" what you could express with a driver into a single DAG.

That is, instead of using Hamilton within itself:

```
def feature_engineering(source_path: str) -> pd.DataFrame:
    '''You could recursively use Hamilton within itself.'''
    dr = driver.Driver({}, feature_modules)
```

```
    df = dr.execute(["feature_df"], inputs={"path": source_path})
    return df
```

You instead can use the *@subdag* decorator to do the same thing, with the added benefit of visibility into the whole DAG:

```
@subdag(
    feature_modules,
    inputs={"path": source("source_path")},
    config={}
)
def feature_engineering(feature_df: pd.DataFrame) ->
pd.DataFrame:
    return feature_df
```

Note that this is immensely powerful – if we draw analogies from Hamilton to standard procedural programming paradigms, we might have the following correspondence:

- *config.when* + friends – *if/else* statements
- *parameterize/extract_columns* – *for* loop
- *does* – effectively macros

And so on. *@subdag* takes this one step further:

- *@subdag* – subroutine definition

E.G. take a certain set of nodes, and run them with specified parameters.

@subdag declares parameters that are outputs of its subdags. Note that, if you want to use outputs of other components of the DAG, you can use the *external_inputs* parameter to declare the parameters that do *not* come from the subDAG.

Why might you want to use this? Let's take a look at some examples:

1. You have a feature engineering pipeline that you want to run on multiple datasets. If its exactly the same, this is perfect. If not, this works perfectly as well, you just have to utilize different functions in each or the *config.when* + *config* parameter to rerun it.
2. You want to train multiple models in the same DAG that share some logic (in features or training) – this allows you to reuse and continually add more.
3. You want to combine multiple similar DAGs (e.g. one for each business line) into one so you can build a cross-business line model.

This basically bridges the gap between the flexibility of non-declarative pipelining frameworks with the readability/maintainability of declarative ones.

__init__(*load_from: ModuleType | Callable*, *inputs: Dict[str, ParametrizedDependency] = None*, *config: Dict[str, Any] = None*, *namespace: str = None*, *final_node_name: str = None*, *external_inputs: List[str] = None*)

Adds a subDAG to the main DAG.

Parameters:

- **load_from** – The functions that will be used to generate this subDAG.
- **inputs** – Parameterized dependencies to inject into all sources of this subDAG. This should *not* be an intermediate node in the subDAG.
- **config** – A configuration dictionary for *just* this subDAG. Note that this passed in value takes precedence over the DAG's config.
- **namespace** – Namespace with which to prefix nodes. This is optional – if not included, this will default to the function name.
- **final_node_name** – Name of the final node in the subDAG. This is optional – if not included, this will default to the function name.
- **external_inputs** – Parameters in the function that are not produced by the functions passed to the subdag. This is useful if you want to perform some logic with other inputs in the subdag's processing function. Note that this is currently required to differentiate and clarify the inputs to the subdag.

schema

@schema is a function modifier that allows you to specify a schema for the function's inputs/outputs. This can be used to validate data at runtime, visualize, etc...

Reference Documentation

class hamilton.function_modifiers.schema

Container class for schema stuff. This is purely so we can have a nice API for it – E.G. Schema.output

static output(*fields: Tuple[str, str], target_: str | None = None) → SchemaOutput

Initializes a *@schema.output* decorator. This takes in a list of fields, which are tuples of the form *(field_name, field_type)*. The field type must be one of the function_modifiers.SchemaTypes types.

Parameters:

- **target** – Target node to decorate – if *None* it'll decorate all final nodes (E.G. sinks in the subdag), otherwise it will decorate the specified node.
- **fields** – List of fields to add to the schema. Each field is a tuple of the form *(field_name, field_type)*

This is implemented using tags, but that might change. Thus you should not rely on the tags created by this decorator (which is why they are prefixed with *internal*).

To use this, you should decorate a node with *@schema.output*

Example usage:

```
@schema.output(
    ("a", "int"),
    ("b", "float"),
    ("c", "str")
 )
def example_schema() -> pd.DataFrame:
    return pd.DataFrame.from_records({"a": [1], "b": [2.0],
"c": ["3"]})
```

Then, when drawing the DAG, the schema will be displayed as sub-elements in the node for the DAG (if *display_schema* is selected).

tag*

Allows you to attach metadata to a node (any node decorated with the function). A common use of this is to enable marking nodes as part of some data product, or for GDPR/privacy purposes.

For instance:

```
import pandas as pd
from hamilton.function_modifiers import tag

def intermediate_column() -> pd.Series:
```

```
    pass

@tag(data_product='final', pii='true')
def final_column(intermediate_column: pd.Series) -> pd.Series:
    pass
```

How do I query by tags?

Right now, we don't have a specific interface to query by tags, however we do expose them via the driver. Using the `list_available_variables()` capability exposes tags along with their names & types, enabling querying of the available outputs for specific tag matches. E.g.

```
from hamilton import driver
dr = driver.Driver(...)  # create driver as required
all_possible_outputs = dr.list_available_variables()
desired_outputs = [o.name for o in all_possible_outputs
                   if 'my_tag_value' == o.tags.get('my_tag_key')]
output = dr.execute(desired_outputs)
```

Using display_name for visualization

You can use the special `display_name` tag to provide a human-readable name for nodes in graphviz visualizations. This allows you to show user-friendly names in DAG diagrams while keeping valid Python identifiers as function names.

```
import pandas as pd
from hamilton.function_modifiers import tag

@tag(display_name="Customer Lifetime Value")
def customer_ltv(purchases: pd.DataFrame, tenure: pd.Series) ->
pd.Series:
    """Calculate customer lifetime value."""
    return purchases.sum() * tenure
```

When you visualize the DAG using `dr.display_all_functions()`, the node will display "Customer Lifetime Value" instead of "customer_ltv". This is useful for:

- Creating presentation-ready diagrams for stakeholders
- Adding business-friendly names for technical functions
- Making visualizations more readable for non-technical audiences

Note that `display_name` only affects visualization - the actual node name used in code remains the function name.

Reference Documentation

class hamilton.function_modifiers.tag(*, *target_: str | Collection[str] | None | EllipsisType = None*, *bypass_reserved_namespaces_: bool = False*, ***tags: str | List[str]*)

Decorator class that adds a tag to a node. Tags take the form of key/value pairings. Tags can have dots to specify namespaces (keys with dots), but this is usually reserved for special cases (E.G. subdecorators) that utilize them. Usually one will pass in tags as kwargs, so we expect tags to be un-namespaced in most uses.

That is using:

```
@tag(my_tag='tag_value')
def my_function(...) -> ...:
```

is un-namespaced because you cannot put a . in the keyword part (the part before the '=').

But using:

```
@tag(**{'my.tag': 'tag_value'})
def my_function(...) -> ...:
```

allows you to add dots that allow you to namespace your tags.

Currently, tag values are restricted to allowing strings only, although we may consider changing the in the future (E.G. thinking of lists).

Hamilton also reserves the right to change the following: * adding purely positional arguments * not allowing users to use a certain set of top-level prefixes (E.G. any tag where the top level is one of the values in RESERVED_TAG_PREFIX).

Example usage:

```
@tag(foo='bar', a_tag_key='a_tag_value', **{'namespace.tag_key':
'tag_value'})
def my_function(...) -> ...:
    ...
```

__init__(*, *target_: str | Collection[str] | None | EllipsisType = None*, *bypass_reserved_namespaces_: bool = False*, ***tags: str | List[str]*)

Constructor for adding tag annotations to a function.

Parameters:

- **bypass_reserved_namespaces_** – Whether to bypass Reserved Namespace checking.

- **target_** –

  Target nodes to decorate. This can be one of the following:

  - **None**: tag all nodes outputted by this that are "final" (E.g. do not have a node outputted by this that depend on them)
  - **Ellipsis (...)**: tag *all* nodes outputted by this
  - **Collection[str]**: tag *only* the nodes with the specified names
  - **str**: tag *only* the node with the specified name

- **tags** – the keys are always going to be strings, so the type annotation here means the values are strings or lists of values. Implicitly this is *Dict[str, Union[str, List[str]]]* but the PEP guideline is to only annotate it with the value *Union[str, List[str]]*.

class hamilton.function_modifiers.tag_outputs(***tag_mapping: Dict[str, str | List[str]]*)

__init__(***tag_mapping: Dict[str, str | List[str]]*)

Creates a tag_outputs decorator.

Note that this currently does not validate whether the nodes are spelled correctly as it takes in a superset of nodes.

Parameters:

tag_mapping – Mapping of output name to tags – this is akin to applying @tag to individual outputs produced by the function.

Example usage:

```
@tag_output(**{'a': {'a_tag': 'a_tag_value'}, 'b':
{'b_tag': 'b_tag_value'}})
@extract_columns("a", "b")
def example_tag_outputs() -> pd.DataFrame:
    return pd.DataFrame.from_records({"a": [1], "b": [2]})
```

with_columns

We support the *with_columns* operation that appends the results as new columns to the original dataframe for several libraries:

Pandas

Reference Documentation

class hamilton.plugins.h_pandas.with_columns(**load_from: Callable | ModuleType*, *columns_to_pass: List[str] = None*, *pass_dataframe_as: str = None*, *on_input: str = None*, *select: List[str] = None*, *namespace: str = None*, *config_required: List[str] = None*)

Initializes a with_columns decorator for pandas. This allows you to efficiently run groups of map operations on a dataframe.

Here's an example of calling it – if you've seen `@subdag`, you should be familiar with the concepts:

```
# my_module.py
def a(a_from_df: pd.Series) -> pd.Series:
    return _process(a)

def b(b_from_df: pd.Series) -> pd.Series:
    return _process(b)

def a_b_average(a_from_df: pd.Series, b_from_df: pd.Series) ->
pd.Series:
    return (a_from_df + b_from_df) / 2
```

```
# with_columns_module.py
def a_plus_b(a: pd.Series, b: pd.Series) -> pd.Series:
    return a + b


# the with_columns call
@with_columns(
    *[my_module], # Load from any module
    *[a_plus_b], # or list operations directly
    columns_to_pass=["a_from_df", "b_from_df"], # The columns to
pass from the dataframe to
    # the subdag
    select=["a", "b", "a_plus_b", "a_b_average"], # The columns
to select from the dataframe
)
def final_df(initial_df: pd.DataFrame) -> pd.DataFrame:
```

```
    # process, or just return unprocessed
    ...
```

In this instance the `initial_df` would get two columns added: `a_plus_b` and `a_b_average`.

The operations are applied in topological order. This allows you to express the operations individually, making it easy to unit-test and reuse.

Note that the operation is "append", meaning that the columns that are selected are appended onto the dataframe.

If the function takes multiple dataframes, the dataframe input to process will always be the first argument. This will be passed to the subdag, transformed, and passed back to the function. This follows the hamilton rule of reference by parameter name. To demonstarte this, in the code above, the dataframe that is passed to the subdag is *initial_df*. That is transformed by the subdag, and then returned as the final dataframe.

You can read it as:

"final_df is a function that transforms the upstream dataframe initial_df, running the transformations from my_module. It starts with the columns a_from_df and b_from_df, and then adds the columns a, b, and a_plus_b to the dataframe. It then returns the dataframe, and does some processing on it."

In case you need more flexibility you can alternatively use `on_input`, for example,

```
# with_columns_module.py
def a_from_df(initial_df: pd.Series) -> pd.Series:
    return initial_df["a_from_df"] / 100

def b_from_df(initial_df: pd.Series) -> pd.Series:
    return initial_df["b_from_df"] / 100


# the with_columns call
@with_columns(
    *[my_module],
    *[a_from_df],
    on_input="initial_df",
    select=["a_from_df", "b_from_df", "a", "b", "a_plus_b",
"a_b_average"],
)
def final_df(initial_df: pd.DataFrame, ...) -> pd.DataFrame:
    # process, or just return unprocessed
    ...
```

the above would output a dataframe where the two columns `a_from_df` and `b_from_df` get overwritten.

__init__(*load_from: Callable | ModuleType*, *columns_to_pass: List[str] = None*, *pass_dataframe_as: str = None*, *on_input: str = None*, *select: List[str] = None*, *namespace: str = None*, *config_required: List[str] = None*)

Instantiates a `@with_columns` decorator.

Parameters:

- **load_from** – The functions or modules that will be used to generate the group of map operations.
- **columns_to_pass** – The initial schema of the dataframe. This is used to determine which upstream inputs should be taken from the dataframe, and which shouldn't. Note that, if this is left empty (and external_inputs is as well), we will assume that all dependencies come from the dataframe. This cannot be used in conjunction with on_input.
- **on_input** – The name of the dataframe that we're modifying, as known to the subdag. If you pass this in, you are responsible for extracting columns out. If not provided, you have to pass columns_to_pass in, and we will extract the columns out on the first parameter for you.
- **select** – The end nodes that represent columns to be appended to the original dataframe via with_columns. Existing columns will be overridden. The selected nodes need to have the corresponding column type, in this case pd.Series, to be appended to the original dataframe.
- **namespace** – The namespace of the nodes, so they don't clash with the global namespace and so this can be reused. If its left out, there will be no namespace (in which case you'll want to be careful about repeating it/reusing the nodes in other parts of the DAG.)
- **config_required** – the list of config keys that are required to resolve any functions. Pass in None if you

want the functions/modules to have access to all possible config.

Polar (Eager)

Reference Documentation

class hamilton.plugins.h_polars.with_columns(**load_from: Callable | ModuleType*, *columns_to_pass: List[str] = None*, *pass_dataframe_as: str = None*, *on_input: str = None*, *select: List[str] = None*, *namespace: str = None*, *config_required: List[str] = None*)

Initializes a with_columns decorator for polars.

This allows you to efficiently run groups of map operations on a dataframe. We support both eager and lazy mode in polars. In case of using eager mode the type should be pl.DataFrame and the subsequent operations run on columns with type pl.Series.

Here's an example of calling in eager mode – if you've seen `@subdag`, you should be familiar with the concepts:

```
# my_module.py
def a_b_average(a: pl.Series, b: pl.Series) -> pl.Series:
    return (a + b) / 2
```

```
# with_columns_module.py
def a_plus_b(a: pl.Series, b: pl.Series) -> pl.Series:
    return a + b


# the with_columns call
@with_columns(
    *[my_module], # Load from any module
    *[a_plus_b], # or list operations directly
    columns_to_pass=["a", "b"], # The columns to pass from the
dataframe to
    # the subdag
    select=["a_plus_b", "a_b_average"], # The columns to append
to the dataframe
)
def final_df(initial_df: pl.DataFrame) -> pl.DataFrame:
    # process, or just return unprocessed
    ...
```

In this instance the `initial_df` would get two columns added: `a_plus_b` and `a_b_average`.

Note that the operation is "append", meaning that the columns that are selected are appended onto the dataframe.

If the function takes multiple dataframes, the dataframe input to process will always be the first argument. This will be passed to the subdag, transformed, and passed back to the function. This follows the hamilton rule of reference by parameter name. To demonstarte this, in the code above, the dataframe that is passed to the subdag is *initial_df*. That is transformed by the subdag, and then returned as the final dataframe.

You can read it as:

"final_df is a function that transforms the upstream dataframe initial_df, running the transformations from my_module. It starts with the columns a_from_df and b_from_df, and then adds the columns a, b, and a_plus_b to the dataframe. It then returns the dataframe, and does some processing on it."

In case you need more flexibility you can alternatively use `on_input`, for example,

```
# with_columns_module.py
def a_from_df() -> pl.Expr:
    return pl.col(a).alias("a") / 100

def b_from_df() -> pl.Expr:
    return pl.col(b).alias("b") / 100


# the with_columns call
@with_columns(
    *[my_module],
    on_input="initial_df",
    select=["a_from_df", "b_from_df", "a_plus_b", "a_b_average"],
)
def final_df(initial_df: pl.DataFrame) -> pl.DataFrame:
    # process, or just return unprocessed
    ...
```

the above would output a dataframe where the two columns `a` and `b` get overwritten.

__init__(*load_from: Callable | ModuleType*, *columns_to_pass: List[str] = None*, *pass_dataframe_as: str = None*, *on_input: str = None*, *select: List[str] = None*, *namespace: str = None*, *config_required: List[str] = None*)

Instantiates a `@with_columns` decorator.

Parameters:

- **load_from** – The functions or modules that will be used to generate the group of map operations.

- **columns_to_pass** – The initial schema of the dataframe. This is used to determine which upstream inputs should be taken from the dataframe, and which shouldn't. Note that, if this is left empty (and external_inputs is as well), we will assume that all dependencies come from the dataframe. This cannot be used in conjunction with on_input.

- **on_input** – The name of the dataframe that we're modifying, as known to the subdag. If you pass this in, you are responsible for extracting columns out. If not provided, you have to pass columns_to_pass in, and we will extract the columns out on the first parameter for you.

- **select** – The end nodes that represent columns to be appended to the original dataframe via with_columns. Existing columns will be overridden. The selected nodes need to have the corresponding column type, in this case pl.Series, to be appended to the original dataframe.

- **namespace** – The namespace of the nodes, so they don't clash with the global namespace and so this can be reused. If its left out, there will be no namespace (in which case you'll want to be careful about repeating it/reusing the nodes in other parts of the DAG.)

- **config_required** – the list of config keys that are required to resolve any functions. Pass in None if you want the functions/modules to have access to all possible config.

Polars (Lazy)

Reference Documentation

class hamilton.plugins.h_polars_lazyframe.with_columns(*load_from: Callable | ModuleType*, *columns_to_pass: List[str] = None*, *pass_dataframe_as: str = None*, *on_input: str = None*, *select: List[str] = None*, *namespace: str = None*, *config_required: List[str] = None*)

Initializes a with_columns decorator for polars.

This allows you to efficiently run groups of map operations on a dataframe. We support both eager and lazy mode in polars. For lazy execution, use pl.LazyFrame and the subsequent operations should be typed as pl.Expr. See examples/polars/with_columns for a practical implementation in both variations.

The lazy execution would be:

```
# my_module.py
def a_b_average(a: pl.Expr, b: pl.Expr) -> pl.Expr:
    return (a + b) / 2
```

```
# with_columns_module.py
def a_plus_b(a: pl.Expr, b: pl.Expr) -> pl.Expr:
    return a + b


# the with_columns call
@with_columns(
    *[my_module], # Load from any module
    *[a_plus_b], # or list operations directly
    columns_to_pass=["a_from_df", "b_from_df"], # The columns to
pass from the dataframe to
    # the subdag
    select=["a_plus_b", "a_b_average"], # The columns to append
to the dataframe
)
def final_df(initial_df: pl.LazyFrame) -> pl.LazyFrame:
    # process, or just return unprocessed
    ...
```

Note that the operation is “append”, meaning that the columns that are selected are appended onto the dataframe.

If the function takes multiple dataframes, the dataframe input to process will always be the first argument. This will be passed to the subdag, transformed, and passed back to the function. This follows the hamilton rule of reference by parameter name. To demonstarte this, in the code above, the dataframe that is passed to the subdag is *initial_df*. That is transformed by the subdag, and then returned as the final dataframe.

You can read it as:

“final_df is a function that transforms the upstream dataframe initial_df, running the transformations from my_module. It starts with the columns a_from_df and b_from_df, and then adds the columns a, b, and a_plus_b to the dataframe. It then returns the dataframe, and does some processing on it.”

In case you need more flexibility you can alternatively use `on_input`, for example,

```
# with_columns_module.py
def a_from_df() -> pl.Expr:
    return pl.col(a).alias("a") / 100

def b_from_df() -> pd.Expr:
    return pl.col(a).alias("b") / 100


# the with_columns call
@with_columns(
    *[my_module],
    on_input="initial_df",
    select=["a_from_df", "b_from_df", "a_plus_b", "a_b_average"],
)
def final_df(initial_df: pl.LazyFrame) -> pl.LazyFrame:
    # process, or just return unprocessed
    ...
```

the above would output a dataframe where the two columns `a` and `b` get overwritten.

__init__(*load_from: Callable | ModuleType*, *columns_to_pass: List[str] = None*, *pass_dataframe_as: str = None*, *on_input: str = None*, *select: List[str] = None*, *namespace: str = None*, *config_required: List[str] = None*)

Instantiates a `@with_columns` decorator.

Parameters:

- **load_from** – The functions or modules that will be used to generate the group of map operations.
- **columns_to_pass** – The initial schema of the dataframe. This is used to determine which upstream inputs should be taken from the dataframe, and which shouldn't. Note that, if this is left empty (and external_inputs is as well), we will assume that all dependencies come from the dataframe. This cannot be used in conjunction with on_input.
- **on_input** – The name of the dataframe that we're modifying, as known to the subdag. If you pass this in, you are responsible for extracting columns out. If not provided, you have to pass columns_to_pass in, and we will extract the columns out on the first parameter for you.
- **select** – The end nodes that represent columns to be appended to the original dataframe via with_columns.

Existing columns will be overridden. The selected nodes need to have the corresponding column type, in this case pl.Expr, to be appended to the original dataframe.

- **namespace** – The namespace of the nodes, so they don't clash with the global namespace and so this can be reused. If its left out, there will be no namespace (in which case you'll want to be careful about repeating it/reusing the nodes in other parts of the DAG.)

- **config_required** – the list of config keys that are required to resolve any functions. Pass in None if you want the functions/modules to have access to all possible config.

PySpark

This is part of the hamilton pyspark integration. To install, run:

pip install sf-hamilton[pyspark]

Reference Documentation

class hamilton.plugins.h_spark.with_columns(**load_from: Callable | ModuleType*, *columns_to_pass: List[str] = None*, *pass_dataframe_as: str = None*, *on_input: str = None*, *select: List[str] = None*, *namespace: str = None*, *mode: str = 'append'*, *config_required: List[str] = None*)

__init__(**load_from: Callable | ModuleType*, *columns_to_pass: List[str] = None*, *pass_dataframe_as: str = None*, *on_input: str = None*, *select: List[str] = None*, *namespace: str = None*, *mode: str = 'append'*, *config_required: List[str] = None*)

Initializes a with_columns decorator for spark. This allows you to efficiently run

groups of map operations on a dataframe, represented as pandas/primitives UDFs. This effectively "linearizes" compute – meaning that a DAG of map operations can be run as a set of *.withColumn* operations on a single dataframe – ensuring that you don't have to do a complex *extract* then *join* process on spark, which can be inefficient.

Here's an example of calling it – if you've seen *@subdag*, you should be familiar with the concepts:

```
# my_module.py
def a(a_from_df: pd.Series) -> pd.Series:
    return _process(a)
```

```
def b(b_from_df: pd.Series) -> pd.Series:
    return _process(b)

def a_plus_b(a_from_df: pd.Series, b_from_df:
pd.Series) -> pd.Series:
    return a + b


# the with_columns call
@with_columns(
    load_from=[my_module], # Load from any module
    columns_to_pass=["a_from_df", "b_from_df"], # The
columns to pass from the dataframe to
    # the subdag
    select=["a", "b", "a_plus_b"], # The columns to
select from the dataframe
)
def final_df(initial_df: ps.DataFrame) -> ps.DataFrame:
    # process, or just return unprocessed
    ...
```

You can think of the above as a series of withColumn calls on the dataframe, where the operations are applied in topological order. This is significantly more efficient than extracting out the columns, applying the maps, then joining, but *also* allows you to express the operations individually, making it easy to unit-test and reuse.

Note that the operation is "append", meaning that the columns that are selected are appended onto the dataframe. We will likely add an option to have this be either "select" or "append" mode.

If the function takes multiple dataframes, the dataframe input to process will always be the first one. This will be passed to the subdag, transformed, and passed back to the functions. This follows the hamilton rule of reference by parameter name. To demonstarte this, in the code above, the dataframe that is passed to the subdag is *initial_df*. That is transformed by the subdag, and then returned as the final dataframe.

You can read it as:

"final_df is a function that transforms the upstream dataframe initial_df, running the transformations from my_module. It starts with the columns a_from_df and b_from_df, and then adds the columns a, b, and a_plus_b to the dataframe. It then returns the dataframe, and does some processing on it."

Parameters:

- **load_from** – The functions that will be used to generate the group of map operations.
- **columns_to_pass** – The initial schema of the dataframe. This is used to determine which upstream inputs should be taken from the dataframe, and which shouldn't. Note that, if this is left empty (and external_inputs is as well), we will assume that all dependencies come from the dataframe. This cannot be used in conjunction with pass_dataframe_as.
- **pass_dataframe_as** – The name of the dataframe that we're modifying, as known to the subdag. If you pass this in, you are responsible for extracting columns out. If not provided, you have to pass columns_to_pass in, and we will extract the columns out for you.
- **select** – Outputs to select from the subdag, i.e. functions/module passed int. If this is left blank it will add all possible columns from the subdag to the dataframe.
- **namespace** – The namespace of the nodes, so they don't clash with the global namespace and so this can be reused. If its left out, there will be no namespace (in which case you'll want to be careful about repeating it/reusing the nodes in other parts of the DAG.)
- **mode** – The mode of the operation. This can be either "append" or "select". If it is "append", it will keep all original columns in the dataframe, and append what's in select. If it is "select", it will do a global select of columns in the dataframe from the *select* parameter. Note that, if the *select* parameter is left blank, it will add all columns in the dataframe that are in the subdag. This defaults to *append*. If you're using select, use the *@select* decorator instead.
- **config_required** – the list of config keys that are required to resolve any functions. Pass in None if you want the functions/modules to have access to all possible config.

Drivers

Currently, we have one main driver. It's highly parameterizable, allowing you to customize:

- The way the DAG is executed (how each node is executed), i.e. either locally, in parallel, or on a cluster!
- How the results are materialized back to you – e.g. a DataFrame, a dictionary, your custom object!

To tune the above, pass in a Graph Adapter, a Result Builder, and/or anotehr lifecycle method – see ResultBuilders, GraphAdapters.

Let's walk through how you might use the Hamilton Driver.

Instantiation

1. Determine the configuration required to setup the DAG.
2. Provide the python modules that should be crawled to create the DAG.
3. Optional. Determine the return type of the object you want `execute()` to return. Default is to create a Pandas DataFrame.

```
from hamilton import driver
from hamilton import base

# 1. Setup config. See the Parameterizing the DAG section for usage
config = {}

# 2. we need to tell hamilton where to load function definitions from
module_name = 'my_functions'
module = importlib.import_module(module_name)  # or simply "import
my_functions"

# 3. Determine the return type -- default is a pandas.DataFrame.
adapter = base.SimplePythonDataFrameGraphAdapter()
# See GraphAdapter docs for more details.

# These all feed into creating the driver & thus DAG.
dr = driver.Driver(config, module, adapter=adapter)
```

Execution

Using a DAG once

This approach assumes that all inputs were passed in with the `config` dictionary above.

```
output = ['output1', 'output2', ...]
df = dr.execute(output)
```

Using a DAG multiple times

This approach assumes that at least one input is not provided in the `config` dictionary provided to the constructor, and instead you provide that input to each `execute` invocation.

```
output = ['output1', 'output2', ...]
for data in dataset:  # if data is a dict of values.
    df = dr.execute(output, inputs=data)
```

Short circuiting some DAG computation

This will force Apache Hamilton to short circuit a particular computation path, and use the passed in override as a result of that particular node.

```
output = ['output1', 'output2', ...]
df = dr.execute(output, overrides={'intermediate_node':
intermediate_value})
```

Reference

Builder

Use this to instantiate a driver.

class hamilton.driver.Builder

__init__()

Constructs a driver builder. No parameters as you call methods to set fields.

allow_module_overrides() → Builder

Same named functions in different modules get overwritten. If multiple modules have same named functions, the later module overrides the previous one(s). The order of listing the modules is important, since later ones will overwrite the previous ones. This is a global call affecting all imported modules. See https://github.com/apache/hamilton/tree/main/examples/module_overrides for more info.

Returns:

self

build() → Driver

Builds the driver – note that this can return a different class, so you'll likely want to have a sense of what it returns.

Note: this defaults to a dictionary adapter if no adapter is set.

Returns:

The driver you specified.

property cache: HamiltonCacheAdapter | None

Attribute to check if a cache was set, either via *.with_cache()* or *.with_adapters(SmartCacheAdapter())*

Required for the check *._require_field_unset()*

copy() → Builder

Creates a copy of the current state of this Builder.

NOTE. The copied Builder currently holds reference of Builder attributes

enable_dynamic_execution(*, *allow_experimental_mode: bool = False*) → Builder

Enables the Parallelizable[] type, which in turn enables: 1. Grouped execution into tasks 2. Parallel execution :return: self

with_adapter(*adapter: HamiltonGraphAdapter*) → Builder

Sets the adapter to use.

Parameters:

adapter – Adapter to use.

Returns:

self

with_adapters(**adapters: BasePreDoAnythingHook | BaseDoCheckEdgeTypesMatch | BaseDoValidateInput | BaseValidateNode | BaseValidateGraph | BasePostGraphConstruct | BasePostGraphConstructAsync | BasePreGraphExecute | BasePreGraphExecuteAsync | BasePostTaskGroup | BasePostTaskExpand | BasePreTaskSubmission | BasePostTaskReturn | BasePreTaskExecute | BasePreTaskExecuteAsync | BasePreNodeExecute | BasePreNodeExecuteAsync | BaseDoNodeExecute | BaseDoNodeExecuteAsync | BasePostNodeExecute | BasePostNodeExecuteAsync | BasePostTaskExecute | BasePostTaskExecuteAsync | BasePostGraphExecute | BasePostGraphExecuteAsync | BaseDoBuildResult*) → Builder

Sets the adapter to use.

Parameters:

adapter – Adapter to use.

Returns:

self

with_cache(*path: str | Path = '.hamilton_cache'*, *metadata_store: MetadataStore | None = None*, *result_store: ResultStore | None = None*, *default: Literal[True] | Sequence[str] | None = None*, *recompute: Literal[True] | Sequence[str] | None = None*, *ignore: Literal[True] | Sequence[str] | None = None*, *disable: Literal[True] | Sequence[str] | None = None*, *default_behavior: Literal['default', 'recompute', 'disable', 'ignore'] = 'default'*, *default_loader_behavior: Literal['default', 'recompute', 'disable', 'ignore'] = 'default'*, *default_saver_behavior: Literal['default', 'recompute', 'disable', 'ignore'] = 'default'*, *log_to_file: bool = False*) → Builder

Add the caching adapter to the *Driver*

Parameters:

- **path** – path where the cache metadata and results will be stored
- **metadata_store** – BaseStore handling metadata for the cache adapter
- **result_store** – BaseStore caching dataflow execution results
- **default** – Set caching behavior to DEFAULT for specified node names. If True, apply to all nodes.

- **recompute** – Set caching behavior to RECOMPUTE for specified node names. If True, apply to all nodes.
- **ignore** – Set caching behavior to IGNORE for specified node names. If True, apply to all nodes.
- **disable** – Set caching behavior to DISABLE for specified node names. If True, apply to all nodes.
- **default_behavior** – Set the default caching behavior.
- **default_loader_behavior** – Set the default caching behavior *DataLoader* nodes.
- **default_saver_behavior** – Set the default caching behavior *DataSaver* nodes.

Log_to_file:

If True, the cache adapter logs will be stored in JSONL format under the metadata_store directory

Returns:

self

Learn more on the Caching Concepts page.

```
from hamilton import driver
import my_dataflow

dr = (
    driver.Builder()
    .with_module(my_dataflow)
    .with_cache()
    .build()
)

# execute twice
dr.execute([...])
dr.execute([...])

# view cache logs
dr.cache.logs()
```

with_config(*config: Dict[str, Any]*) → Builder

Adds the specified configuration to the config. This can be called multilple times – later calls will take precedence.

Parameters:

config – Config to use.

Returns:

self

with_execution_manager(*execution_manager: ExecutionManager*) → Builder

Sets the execution manager to use. Note that this cannot be used if local_executor or remote_executor are also set

Parameters:

execution_manager

Returns:

self

with_grouping_strategy(*grouping_strategy: GroupingStrategy*) → Builder

Sets a node grouper, which tells the driver how to group nodes into tasks for execution.

Parameters:

node_grouper – Node grouper to use.

Returns:

self

with_local_executor(*local_executor: TaskExecutor*) → Builder

Sets the execution manager to use. Note that this cannot be used if local_executor or remote_executor are also set

Parameters:

local_executor – Local executor to use

Returns:

self

with_materializers(**materializers: ExtractorFactory | MaterializerFactory*) → Builder

Add materializer nodes to the *Driver* The generated nodes can be referenced by name in *.execute()*

Parameters:

materializers – materializers to add to the dataflow

Returns:

self

with_modules(**modules: ModuleType*) → Builder

Adds the specified modules to the modules list. This can be called multiple times.

Parameters:

modules – Modules to use.

Returns:

self

with_remote_executor(*remote_executor: TaskExecutor*) → Builder

Sets the execution manager to use. Note that this cannot be used if local_executor or remote_executor are also set

Parameters:

remote_executor – Remote executor to use

Returns:

self

Driver

Use this driver in a general python context. E.g. batch, jupyter notebook, etc.

class hamilton.driver.Driver(*config: Dict[str, Any]*, **modules: ModuleType*, *adapter: BasePreDoAnythingHook | BaseDoCheckEdgeTypesMatch | BaseDoValidateInput | BaseValidateNode | BaseValidateGraph | BasePostGraphConstruct | BasePostGraphConstructAsync | BasePreGraphExecute | BasePreGraphExecuteAsync | BasePostTaskGroup | BasePostTaskExpand | BasePreTaskSubmission | BasePostTaskReturn | BasePreTaskExecute | BasePreTaskExecuteAsync | BasePreNodeExecute | BasePreNodeExecuteAsync | BaseDoNodeExecute | BaseDoNodeExecuteAsync | BasePostNodeExecute | BasePostNodeExecuteAsync | BasePostTaskExecute | BasePostTaskExecuteAsync | BasePostGraphExecute | BasePostGraphExecuteAsync | BaseDoBuildResult | List[BasePreDoAnythingHook | BaseDoCheckEdgeTypesMatch | BaseDoValidateInput | BaseValidateNode | BaseValidateGraph | BasePostGraphConstruct | BasePostGraphConstructAsync | BasePreGraphExecute | BasePreGraphExecuteAsync | BasePostTaskGroup | BasePostTaskExpand | BasePreTaskSubmission | BasePostTaskReturn | BasePreTaskExecute | BasePreTaskExecuteAsync | BasePreNodeExecute | BasePreNodeExecuteAsync | BaseDoNodeExecute | BaseDoNodeExecuteAsync | BasePostNodeExecute | BasePostNodeExecuteAsync | BasePostTaskExecute | BasePostTaskExecuteAsync | BasePostGraphExecute | BasePostGraphExecuteAsync | BaseDoBuildResult] | None = None*, *allow_module_overrides: bool = False*, *_materializers: Sequence[ExtractorFactory | MaterializerFactory] = None*, *_graph_executor: GraphExecutor = None*, *_use_legacy_adapter: bool = True*)

This class orchestrates creating and executing the DAG to create a dataframe.

```
from hamilton import driver
from hamilton import base

# 1. Setup config or invariant input.
config = {}

# 2. we need to tell hamilton where to load function definitions
from
import my_functions

# or programmatically (e.g. you can script module loading)
module_name = "my_functions"
my_functions = importlib.import_module(module_name)

# 3. Determine the return type -- default is a pandas.DataFrame.
adapter = base.SimplePythonDataFrameGraphAdapter()  # See
GraphAdapter docs for more details.

# These all feed into creating the driver & thus DAG.
dr = driver.Driver(config, module, adapter=adapter)
```

__init__(*config: Dict[str, Any]*, **modules: ModuleType*, *adapter: BasePreDoAnythingHook | BaseDoCheckEdgeTypesMatch | BaseDoValidateInput | BaseValidateNode | BaseValidateGraph | BasePostGraphConstruct | BasePostGraphConstructAsync |*

BasePreGraphExecute | BasePreGraphExecuteAsync | BasePostTaskGroup | BasePostTaskExpand | BasePreTaskSubmission | BasePostTaskReturn | BasePreTaskExecute | BasePreTaskExecuteAsync | BasePreNodeExecute | BasePreNodeExecuteAsync | BaseDoNodeExecute | BaseDoNodeExecuteAsync | BasePostNodeExecute | BasePostNodeExecuteAsync | BasePostTaskExecute | BasePostTaskExecuteAsync | BasePostGraphExecute | BasePostGraphExecuteAsync | BaseDoBuildResult | List[BasePreDoAnythingHook | BaseDoCheckEdgeTypesMatch | BaseDoValidateInput | BaseValidateNode | BaseValidateGraph | BasePostGraphConstruct | BasePostGraphConstructAsync | BasePreGraphExecute | BasePreGraphExecuteAsync | BasePostTaskGroup | BasePostTaskExpand | BasePreTaskSubmission | BasePostTaskReturn | BasePreTaskExecute | BasePreTaskExecuteAsync | BasePreNodeExecute | BasePreNodeExecuteAsync | BaseDoNodeExecute | BaseDoNodeExecuteAsync | BasePostNodeExecute | BasePostNodeExecuteAsync | BasePostTaskExecute | BasePostTaskExecuteAsync | BasePostGraphExecute | BasePostGraphExecuteAsync | BaseDoBuildResult] | None = None, allow_module_overrides: bool = False, _materializers: Sequence[ExtractorFactory | MaterializerFactory] = None, _graph_executor: GraphExecutor = None, _use_legacy_adapter: bool = True)

Constructor: creates a DAG given the configuration & modules to crawl.

Parameters:

- **config** – This is a dictionary of initial data & configuration. The contents are used to help create the DAG.
- **modules** – Python module objects you want to inspect for Hamilton Functions.
- **adapter** – Optional. A way to wire in another way of "executing" a hamilton graph. Defaults to using original Hamilton adapter which is single threaded in memory python.
- **allow_module_overrides** – Optional. Same named functions get overridden by later modules. The order of listing the modules is important, since later ones will overwrite the previous ones. This is a global call affecting all imported modules. See https://github.com/apache/hamilton/tree/main/examples/module_overrides for more info.
- **_materializers** – Not public facing, do not use this parameter. This is injected by the builder.

- **_graph_executor** – Not public facing, do not use this parameter. This is injected by the builder. If you need to tune execution, use the builder to do so.
- **_use_legacy_adapter** – Not public facing, do not use this parameter. This represents whether or not to use the legacy adapter. Defaults to True, as this should be backwards compatible. In Hamilton 2.0.0, this will be removed.

property cache: HamiltonCacheAdapter

Directly access the cache adapter

capture_constructor_telemetry(*error: str | None*, *modules: Tuple[ModuleType]*, *config: Dict[str, Any]*, *adapter: LifecycleAdapterSet*)

Captures constructor telemetry. Notes: (1) we want to do this in a way that does not break. (2) we need to account for all possible states, e.g. someone passing in None, or assuming that the entire constructor code ran without issue, e.g. *adapter* was assigned to *self*.

Parameters:

- **error** – the sanitized error string to send.
- **modules** – the list of modules, could be None.
- **config** – the config dict passed, could be None.
- **adapter** – the adapter passed in, might not be attached to *self* yet.

capture_execute_telemetry(*error: str | None*, *final_vars: List[str]*, *inputs: Dict[str, Any]*, *overrides: Dict[str, Any]*, *run_successful: bool*, *duration: float*)

Captures telemetry after execute has run.

Notes: (1) we want to be quite defensive in not breaking anyone's code with things we do here. (2) thus we want to double-check that values exist before doing something with them.

Parameters:

- **error** – the sanitized error string to capture, if any.
- **final_vars** – the list of final variables to get.
- **inputs** – the inputs to the execute function.

- **overrides** – any overrides to the execute function.
- **run_successful** – whether this run was successful.
- **duration** – time it took to run execute.

display_all_functions(*output_file_path: str = None*, *render_kwargs: dict = None*, *graphviz_kwargs: dict = None*, *show_legend: bool = True*, *orient: str = 'LR'*, *hide_inputs: bool = False*, *deduplicate_inputs: bool = False*, *show_schema: bool = True*, *custom_style_function: Callable = None*, *keep_dot: bool = False*) → graphviz.Digraph | None

Displays the graph of all functions loaded!

Parameters:

- **output_file_path** – the full URI of path + file name to save the dot file to. E.g. 'some/path/graph-all.dot'. Optional. No need to pass it in if you're in a Jupyter Notebook.
- **render_kwargs** – a dictionary of values we'll pass to graphviz render function. Defaults to viewing. If you do not want to view the file, pass in *{'view':False}*. See https://graphviz.readthedocs.io/en/stable/api.html#graphviz.Graph.render for other options.
- **graphviz_kwargs** – Optional. Kwargs to be passed to the graphviz graph object to configure it. E.g. dict(graph_attr={'ratio': '1'}) will set the aspect ratio to be equal of the produced image. See https://graphviz.org/doc/info/attrs.html for options.
- **show_legend** – If True, add a legend to the visualization based on the DAG's nodes.
- **orient** – *LR* stands for "left to right". Accepted values are TB, LR, BT, RL. *orient* will be overwridden by the value of *graphviz_kwargs['graph_attr']['rankdir']* see (https://graphviz.org/docs/attr-types/rankdir/)
- **hide_inputs** – If True, no input nodes are displayed.
- **deduplicate_inputs** – If True, remove duplicate input nodes. Can improve readability depending on the specifics of the DAG.

- **show_schema** – If True, display the schema of the DAG if the nodes have schema data provided
- **custom_style_function** – Optional. Custom style function. See example in repository for example use.
- **keep_dot** – If true, produce a DOT file (ref: https://graphviz.org/doc/info/lang.html)

Returns:

the graphviz object if you want to do more with it. If returned as the result in a Jupyter Notebook cell, it will render.

display_downstream_of(*node_names: str, output_file_path: str = None, render_kwargs: dict = None, graphviz_kwargs: dict = None, show_legend: bool = True, orient: str = 'LR', hide_inputs: bool = False, deduplicate_inputs: bool = False, show_schema: bool = True, custom_style_function: Callable = None, keep_dot: bool = False) → graphviz.Digraph | None

Creates a visualization of the DAG starting from the passed in function name(s).

Note: for any "node" visualized, we will also add its parents to the visualization as well, so there could be more nodes visualized than strictly what is downstream of the passed in function name(s).

Parameters:

- **node_names** – names of function(s) that are starting points for traversing the graph.
- **output_file_path** – the full URI of path + file name to save the dot file to. E.g. 'some/path/graph.dot'. Optional. No need to pass it in if you're in a Jupyter Notebook.
- **render_kwargs** – a dictionary of values we'll pass to graphviz render function. Defaults to viewing. If you do not want to view the file, pass in *{'view':False}*.
- **graphviz_kwargs** – Kwargs to be passed to the graphviz graph object to configure it. E.g. dict(graph_attr={'ratio': '1'}) will set the aspect ratio to be equal of the produced image.
- **show_legend** – If True, add a legend to the visualization based on the DAG's nodes.

- **orient** – *LR* stands for “left to right”. Accepted values are TB, LR, BT, RL. *orient* will be overwridden by the value of *graphviz_kwargs['graph_attr']['rankdir']* see (https://graphviz.org/docs/attr-types/rankdir/)
- **hide_inputs** – If True, no input nodes are displayed.
- **deduplicate_inputs** – If True, remove duplicate input nodes. Can improve readability depending on the specifics of the DAG.
- **show_schema** – If True, display the schema of the DAG if nodes have schema data provided
- **custom_style_function** – Optional. Custom style function.
- **keep_dot** – If true, produce a DOT file (ref: https://graphviz.org/doc/info/lang.html)

Returns:

the graphviz object if you want to do more with it. If returned as the result in a Jupyter Notebook cell, it will render.

display_upstream_of(**node_names: str*, *output_file_path: str = None*, *render_kwargs: dict = None*, *graphviz_kwargs: dict = None*, *show_legend: bool = True*, *orient: str = 'LR'*, *hide_inputs: bool = False*, *deduplicate_inputs: bool = False*, *show_schema: bool = True*, *custom_style_function: Callable = None*, *keep_dot: bool = False*) → graphviz.Digraph | None

Creates a visualization of the DAG going backwards from the passed in function name(s).

Note: for any “node” visualized, we will also add its parents to the visualization as well, so there could be more nodes visualized than strictly what is upstream of the passed in function name(s).

Parameters:

- **node_names** – names of function(s) that are starting points for traversing the graph.
- **output_file_path** – the full URI of path + file name to save the dot file to. E.g. ‘some/path/graph.dot’. Optional. No need to pass it in if you’re in a Jupyter Notebook.

- **render_kwargs** – a dictionary of values we'll pass to graphviz render function. Defaults to viewing. If you do not want to view the file, pass in *{'view':False}*. Optional.
- **graphviz_kwargs** – Kwargs to be passed to the graphviz graph object to configure it. E.g. dict(graph_attr={'ratio': '1'}) will set the aspect ratio to be equal of the produced image. Optional.
- **show_legend** – If True, add a legend to the visualization based on the DAG's nodes.
- **orient** – *LR* stands for "left to right". Accepted values are TB, LR, BT, RL. *orient* will be overwridden by the value of *graphviz_kwargs['graph_attr']['rankdir']* see (https://graphviz.org/docs/attr-types/rankdir/)
- **hide_inputs** – If True, no input nodes are displayed.
- **deduplicate_inputs** – If True, remove duplicate input nodes. Can improve readability depending on the specifics of the DAG.
- **show_schema** – If True, display the schema of the DAG if nodes have schema data provided
- **custom_style_function** – Optional. Custom style function.
- **keep_dot** – If true, produce a DOT file (ref: https://graphviz.org/doc/info/lang.html)

Returns:

the graphviz object if you want to do more with it. If returned as the result in a Jupyter Notebook cell, it will render.

execute(*final_vars: List[str | Callable | HamiltonNode]*, *overrides: Dict[str, Any] = None*, *display_graph: bool = False*, *inputs: Dict[str, Any] = None*) → Any

Executes computation.

Parameters:

- **final_vars** – the final list of outputs we want to compute.
- **overrides** – values that will override “nodes” in the DAG.
- **display_graph** – DEPRECATED. Whether we want to display the graph being computed.
- **inputs** – Runtime inputs to the DAG.

Returns:

an object consisting of the variables requested, matching the type returned by the GraphAdapter. See constructor for how the GraphAdapter is initialized. The default one right now returns a pandas dataframe.

export_execution(*final_vars: List[str]*, *inputs: Dict[str, Any] = None*, *overrides: Dict[str, Any] = None*) → str

Method to create JSON representation of the Graph.

Parameters:

- **final_vars** – The final variables to compute.
- **inputs** – Optional. The inputs to the DAG.
- **overrides** – Optional. Overrides to the DAG.

Returns:

JSON string representation of the graph.

has_cycles(*final_vars: List[str | Callable | HamiltonNode]*, *_fn_graph: FunctionGraph = None*) → bool

Checks that the created graph does not have cycles.

Parameters:

- **final_vars** – the outputs we want to compute.
- **_fn_graph** – the function graph to check for cycles, used internally

Returns:

boolean True for cycles, False for no cycles.

list_available_variables(*, *tag_filter: Dict[str, str | None | List[str]] = None*) → List[HamiltonNode]

Returns available variables, i.e. outputs.

These variables correspond 1:1 with nodes in the DAG, and contain the following information:

1. name: the name of the node
2. tags: the tags associated with this node
3. type: The type of data this node returns
4. is_external_input: Whether this node represents an external input (required from outside), or not (has a function specifying its behavior).

```
# gets all
dr.list_available_variables()
# gets exact matching tag name and tag value
dr.list_available_variables({"TAG_NAME": "TAG_VALUE"})
# gets all matching tag name and at least one of the values
in the list
dr.list_available_variables({"TAG_NAME": ["TAG_VALUE1",
"TAG_VALUE2"]})
# gets all with matching tag name, irrespective of value
dr.list_available_variables({"TAG_NAME": None})
# AND query between the two tags (i.e. both need to match)
dr.list_available_variables({"TAG_NAME": "TAG_VALUE",
"TAG_NAME2": "TAG_VALUE2"}
```

Parameters:

tag_filter – A dictionary of tags to filter by. Only nodes matching the tags and their values will be returned. If the value for a tag is None, then we will return all nodes

with that tag. If the value is non-empty we will return all nodes with that tag and that value.

Returns:

list of available variables (i.e. outputs).

materialize(**materializers: MaterializerFactory | ExtractorFactory*, *additional_vars: List[str | Callable | HamiltonNode] = None*, *overrides: Dict[str, Any] = None*, *inputs: Dict[str, Any] = None*) → Tuple[Any, Dict[str, Any]]

Executes and materializes with ad-hoc materializers (*to*) and extractors (*from_*).This does the following:

1. Creates a new graph, appending the desired materialization nodes and prepending the desired extraction nodes
2. Runs the portion of the DAG upstream of the materialization nodes outputted, as well as any additional nodes requested (which can be empty)
3. Returns a Tuple[Materialization metadata, additional vars result]

For instance, say you want to load data, process it, then materialize the output of a node to CSV:

```
from hamilton import driver, base
from hamilton.io.materialization import to
dr = driver.Driver(my_module, {})
# foo, bar are pd.Series
metadata, result = dr.materialize(
    from_.csv(
        target="input_data",
        path="./input.csv"
    ),
    to.csv(
        path="./output.csv"
        id="foo_bar_csv",
        dependencies=["foo", "bar"],
        combine=base.PandasDataFrameResult()
    ),
    additional_vars=["foo", "bar"]
)
```

The code above will do the following:

1. Load the CSV at "./input.csv" and inject it into he DAG as input_data
2. Run the nodes in the DAG on which "foo" and "bar" depend

3. Materialize the dataframe with "foo" and "bar" as columns, saving it as a CSV file at "./output.csv". The metadata will contain any additional relevant information, and result will be a dictionary with the keys "foo" and "bar" containing the original data.

Note that we pass in a *ResultBuilder* as the *combine* argument to *to*, as we may be materializing several nodes. This is not relevant in *from_* as we are only loading one dataset.

additional_vars is used for debugging – E.G. if you want to both realize side-effects and return an output for inspection. If left out, it will return an empty dictionary.

You can bypass the *combine* keyword for *to* if only one output is required. In this circumstance "combining/joining" isn't required, e.g. you do that yourself in a function and/or the output of the function can be directly used. In the case below the output can be turned in to a CSV.

```
from hamilton import driver, base
from hamilton.io.materialization import to
dr = driver.Driver(my_module, {})
# foo, bar are pd.Series
metadata, _ = dr.materialize(
    from_.csv(
        target="input_data",
        path="./input.csv"
    ),
    to.csv(
        path="./output.csv"
        id="foo_bar_csv",
        dependencies=["foo_bar_already_joined],
    ),
)
```

This will just save it to a csv.

Note that materializers can be any valid DataSaver – these have an isomorphic relationship with the *@save_to* decorator, which means that any key utilizable in *save_to* can be used in a materializer. The constructor arguments for a materializer are the same as the arguments for *@save_to*, with an additional trick – instead of requiring everything to be a *source* or *value*, you can pass in a literal, and it will be interpreted as a value.

That said, if you want to parameterize your materializer based on input or some node in the DAG, you can easily do that as well:

```
from hamilton import driver, base
from hamilton.function_modifiers import source
from hamilton.io.materialization import to
```

```
dr = driver.Driver(my_module, {})
# foo, bar are pd.Series
metadata, result = dr.Materialize(
    from_.csv(
       target="input_data",
       path=source("load_path")
    ),
    to.csv(
        path=source("save_path"),
        id="foo_bar_csv",
        dependencies=["foo", "bar"],
        combine=base.PandasDataFrameResult(),
    ),
    additional_vars=["foo", "bar"],
    inputs={"save_path": "./output.csv"},
)
```

While this is a contrived example, you could imagine something more powerful. Say, for instance, say you have created and registered a custom *model_registry* materializer that applies to an argument of your model class, and requires *training_data* to infer the signature. You could call it like this:

```
from hamilton import driver, base
from hamilton.function_modifiers import source
from hamilton.io.materialization import to
dr = driver.Driver(my_module, {})
metadata, _ = dr.Materialize(
    to.model_registry(
        training_data=source("training_data"),
        id="foo_model_registry",
        tags={"run_id" : ..., "training_date" : ..., ...},
        dependencies=["foo_model"]
    ),
)
```

In this case, we bypass a result builder (as there's only one model), the single node we depend on gets saved, and we pass in the training data as an input so the materializer can infer the signature.

You could also imagine a driver that loads up a model, runs inference, then saves the result:

```
from hamilton import driver, base
from hamilton.function_modifiers import source
from hamilton.io.materialization import to

dr = driver.Driver(my_module, {})
```

```
metadata, _ = dr.Materialize(
    from_.model_registry(
        target="input_model",
        query_tags={
            "training_date": ...,
            model_version: ...,
        },  # query based on run_id, model_version
    ),
    to.csv(
        path=source("save_path"),
        id="save_inference_data",
        dependencies=["inference_data"],
    ),
)
```

Note that the "from" extractor has an interesting property – it effectively functions as overrides. This means that it can *replace* nodes within a DAG, short-circuiting their behavior. Similar to passing overrides, but they are dynamically computed with the DAG, rather than statically included from the beginning.

This is customizable through a few APIs:

1. Custom data savers (Function modifiers)
2. Custom result builders
3. Custom data loaders (Function modifiers)

If you find yourself writing these, please consider contributing back! We would love to round out the set of available materialization tools.

Parameters:

- **materializers** – Materializer/extractors to use, created with to.xyz or *from.xyz*
- **additional_vars** – Additional variables to return from the graph
- **overrides** – Overrides to pass to execution
- **inputs** – Inputs to pass to execution

Returns:

Tuple[Materialization metadata|data, additional_vars result]

static normalize_adapter_input(*adapter: BasePreDoAnythingHook | BaseDoCheckEdgeTypesMatch | BaseDoValidateInput | BaseValidateNode | BaseValidateGraph | BasePostGraphConstruct | BasePostGraphConstructAsync | BasePreGraphExecute | BasePreGraphExecuteAsync | BasePostTaskGroup | BasePostTaskExpand | BasePreTaskSubmission | BasePostTaskReturn | BasePreTaskExecute | BasePreTaskExecuteAsync | BasePreNodeExecute | BasePreNodeExecuteAsync | BaseDoNodeExecute | BaseDoNodeExecuteAsync | BasePostNodeExecute | BasePostNodeExecuteAsync | BasePostTaskExecute | BasePostTaskExecuteAsync | BasePostGraphExecute | BasePostGraphExecuteAsync | BaseDoBuildResult | List[BasePreDoAnythingHook | BaseDoCheckEdgeTypesMatch | BaseDoValidateInput | BaseValidateNode | BaseValidateGraph | BasePostGraphConstruct | BasePostGraphConstructAsync | BasePreGraphExecute | BasePreGraphExecuteAsync | BasePostTaskGroup | BasePostTaskExpand | BasePreTaskSubmission | BasePostTaskReturn | BasePreTaskExecute | BasePreTaskExecuteAsync | BasePreNodeExecute | BasePreNodeExecuteAsync | BaseDoNodeExecute | BaseDoNodeExecuteAsync | BasePostNodeExecute | BasePostNodeExecuteAsync | BasePostTaskExecute | BasePostTaskExecuteAsync | BasePostGraphExecute | BasePostGraphExecuteAsync | BaseDoBuildResult] | LifecycleAdapterSet | None*, *use_legacy_adapter: bool = True*) → LifecycleAdapterSet

Normalizes the adapter argument in the driver to a list of adapters. Adds back the legacy adapter if needed.

Note that, in the past, hamilton required a graph adapter. Now it is only required to be included in the legacy case default behavior has been modified to handle anything a result builder did.

Parameters:

- **adapter** – Adapter to include
- **use_legacy_adapter** – Whether to use the legacy adapter. Defaults to True.

Returns:

A lifecycle adapter set.

raw_execute(*final_vars: List[str]*, *overrides: Dict[str, Any] = None*, *display_graph: bool = False*, *inputs: Dict[str, Any] = None*, *_fn_graph: FunctionGraph = None*) → Dict[str, Any]

Raw execute function that does the meat of execute.

Don't use this entry point for execution directly. Always go through *.execute()* or *.materialize()*. In case you are using *.raw_execute()* directly, please switch to *.execute()*

using a *base.DictResult()*. Note: *base.DictResult()* is the default return of execute if you are using the *driver.Builder()* class to create a *Driver()* object.

Parameters:

- **final_vars** – Final variables to compute
- **overrides** – Overrides to run.
- **display_graph** – DEPRECATED. DO NOT USE. Whether or not to display the graph when running it
- **inputs** – Runtime inputs to the DAG

Returns:

validate_execution(*final_vars: List[str | Callable | HamiltonNode]*, *overrides: Dict[str, Any] = None*, *inputs: Dict[str, Any] = None*)

Validates execution of the graph. One can call this to validate execution, independently of actually executing. Note this has no return – it will raise a ValueError if there is an issue.

Parameters:

- **final_vars** – Final variables to compute
- **overrides** – Overrides to pass to execution.
- **inputs** – Inputs to pass to execution.

Raises:

ValueError – if any issues with executino can be detected.

static validate_inputs(*fn_graph: FunctionGraph*, *adapter: BasePreDoAnythingHook | BaseDoCheckEdgeTypesMatch | BaseDoValidateInput | BaseValidateNode | BaseValidateGraph | BasePostGraphConstruct | BasePostGraphConstructAsync | BasePreGraphExecute | BasePreGraphExecuteAsync | BasePostTaskGroup | BasePostTaskExpand | BasePreTaskSubmission | BasePostTaskReturn | BasePreTaskExecute | BasePreTaskExecuteAsync | BasePreNodeExecute | BasePreNodeExecuteAsync | BaseDoNodeExecute | BaseDoNodeExecuteAsync | BasePostNodeExecute | BasePostNodeExecuteAsync | BasePostTaskExecute | BasePostTaskExecuteAsync | BasePostGraphExecute | BasePostGraphExecuteAsync | BaseDoBuildResult |*

List[BasePreDoAnythingHook | BaseDoCheckEdgeTypesMatch | BaseDoValidateInput | BaseValidateNode | BaseValidateGraph | BasePostGraphConstruct | BasePostGraphConstructAsync | BasePreGraphExecute | BasePreGraphExecuteAsync | BasePostTaskGroup | BasePostTaskExpand | BasePreTaskSubmission | BasePostTaskReturn | BasePreTaskExecute | BasePreTaskExecuteAsync | BasePreNodeExecute | BasePreNodeExecuteAsync | BaseDoNodeExecute | BaseDoNodeExecuteAsync | BasePostNodeExecute | BasePostNodeExecuteAsync | BasePostTaskExecute | BasePostTaskExecuteAsync | BasePostGraphExecute | BasePostGraphExecuteAsync | BaseDoBuildResult] | LifecycleAdapterSet, *user_nodes: Collection[Node]*, *inputs: Dict[str, Any] | None = None*, *nodes_set: Collection[Node] = None*)

Validates that inputs meet our expectations. This means that: 1. The runtime inputs don't clash with the graph's config 2. All expected graph inputs are provided, either in config or at runtime

Parameters:

- **fn_graph** – The function graph to validate.
- **adapter** – The adapter to use for validation.
- **user_nodes** – The required nodes we need for computation.
- **inputs** – the user inputs provided.
- **nodes_set** – the set of nodes to use for validation; Optional.

validate_materialization(**materializers: MaterializerFactory*, *additional_vars: List[str | Callable | HamiltonNode] = None*, *overrides: Dict[str, Any] = None*, *inputs: Dict[str, Any] = None*)

Validates materialization of the graph. Effectively .materialize() with a dry-run. Note this has no return – it will raise a ValueError if there is an issue.

Parameters:

- **materializers** – Materializers to use, see the materialize() function
- **additional_vars** – Additional variables to compute (in addition to materializers)
- **overrides** – Overrides to pass to execution. Optional.
- **inputs** – Inputs to pass to execution. Optional.

Raises:

ValueError – if any issues with materialization can be detected.

visualize_execution(*final_vars: List[str | Callable | HamiltonNode]*, *output_file_path: str = None*, *render_kwargs: dict = None*, *inputs: Dict[str, Any] = None*, *graphviz_kwargs: dict = None*, *overrides: Dict[str, Any] = None*, *show_legend: bool = True*, *orient: str = 'LR'*, *hide_inputs: bool = False*, *deduplicate_inputs: bool = False*, *show_schema: bool = True*, *custom_style_function: Callable = None*, *bypass_validation: bool = False*, *keep_dot: bool = False*) → graphviz.Digraph | None

Visualizes Execution.

Note: overrides are not handled at this time.

Shapes:

- ovals are nodes/functions
- rectangles are nodes/functions that are requested as output
- shapes with dotted lines are inputs required to run the DAG.

Parameters:

- **final_vars** – the outputs we want to compute. They will become rectangles in the graph.
- **output_file_path** – the full URI of path + file name to save the dot file to. E.g. 'some/path/graph.dot'. Optional. No need to pass it in if you're in a Jupyter Notebook.
- **render_kwargs** – a dictionary of values we'll pass to graphviz render function. Defaults to viewing. If you do not want to view the file, pass in *{'view':False}*. See https://graphviz.readthedocs.io/en/stable/api.html#graphviz.Graph.render for other options.
- **inputs** – Optional. Runtime inputs to the DAG.
- **graphviz_kwargs** – Optional. Kwargs to be passed to the graphviz graph object to configure it. E.g. dict(graph_attr={'ratio': '1'}) will set the aspect ratio to be equal of the produced image. See https://graphviz.org/doc/info/attrs.html for options.

- **overrides** – Optional. Overrides to the DAG.
- **show_legend** – If True, add a legend to the visualization based on the DAG's nodes.
- **orient** – *LR* stands for "left to right". Accepted values are TB, LR, BT, RL. *orient* will be overwridden by the value of *graphviz_kwargs['graph_attr']['rankdir']* see (https://graphviz.org/docs/attr-types/rankdir/)
- **hide_inputs** – If True, no input nodes are displayed.
- **deduplicate_inputs** – If True, remove duplicate input nodes. Can improve readability depending on the specifics of the DAG.
- **show_schema** – If True, display the schema of the DAG if nodes have schema data provided
- **custom_style_function** – Optional. Custom style function.
- **keep_dot** – If true, produce a DOT file (ref: https://graphviz.org/doc/info/lang.html)

Returns:

the graphviz object if you want to do more with it. If returned as the result in a Jupyter Notebook cell, it will render.

visualize_materialization(**materializers: MaterializerFactory | ExtractorFactory*, *output_file_path: str = None*, *render_kwargs: dict = None*, *additional_vars: List[str | Callable | HamiltonNode] = None*, *inputs: Dict[str, Any] = None*, *graphviz_kwargs: dict = None*, *overrides: Dict[str, Any] = None*, *show_legend: bool = True*, *orient: str = 'LR'*, *hide_inputs: bool = False*, *deduplicate_inputs: bool = False*, *show_schema: bool = True*, *custom_style_function: Callable = None*, *bypass_validation: bool = False*, *keep_dot: bool = False*) → graphviz.Digraph | None

Visualizes materialization. This helps give you a sense of how materialization will impact the DAG.

Parameters:

- **materializers** – Materializers/Extractors to use, see the materialize() function

- **additional_vars** – Additional variables to compute (in addition to materializers)
- **output_file_path** – Path to output file. Optional. Skip if in a Jupyter Notebook.
- **render_kwargs** – Arguments to pass to render. Optional.
- **inputs** – Inputs to pass to execution. Optional.
- **graphviz_kwargs** – Arguments to pass to graphviz. Optional.
- **overrides** – Overrides to pass to execution. Optional.
- **show_legend** – If True, add a legend to the visualization based on the DAG's nodes.
- **orient** – *LR* stands for "left to right". Accepted values are TB, LR, BT, RL. *orient* will be overwridden by the value of *graphviz_kwargs['graph_attr']['rankdir']* see (https://graphviz.org/docs/attr-types/rankdir/)
- **hide_inputs** – If True, no input nodes are displayed.
- **deduplicate_inputs** – If True, remove duplicate input nodes. Can improve readability depending on the specifics of the DAG.
- **show_schema** – If True, show the schema of the materialized nodes if nodes have schema metadata attached.
- **custom_style_function** – Optional. Custom style function.
- **bypass_validation** – If True, bypass validation. Optional.

Returns:

The graphviz graph, if you want to do something with it

visualize_path_between(*upstream_node_name: str*, *downstream_node_name: str*, *output_file_path: str | None = None*, *render_kwargs: dict = None*, *graphviz_kwargs: dict = None*, *strict_path_visualization: bool = False*, *show_legend: bool = True*, *orient: str = 'LR'*,

hide_inputs: bool = False, *deduplicate_inputs: bool = False*, *show_schema: bool = True*, *custom_style_function: Callable = None*, *keep_dot: bool = False*) → graphviz.Digraph | None

Visualizes the path between two nodes.

This is useful for debugging and understanding the path between two nodes.

Parameters:

- **upstream_node_name** – the name of the node that we want to start from.
- **downstream_node_name** – the name of the node that we want to end at.
- **output_file_path** – the full URI of path + file name to save the dot file to. E.g. 'some/path/graph.dot'. Pass in None to skip saving any file.
- **render_kwargs** – a dictionary of values we'll pass to graphviz render function. Defaults to viewing. If you do not want to view the file, pass in *{'view':False}*.
- **graphviz_kwargs** – Kwargs to be passed to the graphviz graph object to configure it. E.g. dict(graph_attr={'ratio': '1'}) will set the aspect ratio to be equal of the produced image.
- **strict_path_visualization** – If True, only the nodes in the path will be visualized. If False, the nodes in the path and their dependencies, i.e. parents, will be visualized.
- **show_legend** – If True, add a legend to the visualization based on the DAG's nodes.
- **orient** – *LR* stands for "left to right". Accepted values are TB, LR, BT, RL. *orient* will be overwridden by the value of *graphviz_kwargs['graph_attr']['rankdir']* see (https://graphviz.org/docs/attr-types/rankdir/)
- **hide_inputs** – If True, no input nodes are displayed.
- **deduplicate_inputs** – If True, remove duplicate input nodes. Can improve readability depending on the specifics of the DAG.

- **show_schema** – If True, display the schema of the DAG if nodes have schema data provided
- **custom_style_function** – Optional. Custom style function.
- **keep_dot** – If true, produce a DOT file (ref: https://graphviz.org/doc/info/lang.html)

Returns:

graphviz object.

Raises:

ValueError – if the upstream or downstream node names are not found in the graph, or there is no path between them.

what_is_downstream_of(**node_names: str*) → List[HamiltonNode]

Tells you what is downstream of this function(s), i.e. node(s).

Parameters:

node_names – names of function(s) that are starting points for traversing the graph.

Returns:

list of "variables" (i.e. nodes), inclusive of the function names, that are downstream of the passed in function names.

what_is_the_path_between(*upstream_node_name: str*, *downstream_node_name: str*) → List[HamiltonNode]

Tells you what nodes are on the path between two nodes.

Note: this is inclusive of the two nodes, and returns an unsorted list of nodes.

Parameters:

- **upstream_node_name** – the name of the node that we want to start from.

- **downstream_node_name** – the name of the node that we want to end at.

Returns:

Nodes representing the path between the two nodes, inclusive of the two nodes, unsorted. Returns empty list if no path exists.

Raises:

ValueError – if the upstream or downstream node name is not in the graph.

what_is_upstream_of(**node_names: str*) → List[HamiltonNode]

Tells you what is upstream of this function(s), i.e. node(s).

Parameters:

node_names – names of function(s) that are starting points for traversing the graph backwards.

Returns:

list of "variables" (i.e. nodes), inclusive of the function names, that are upstream of the passed in function names.

DefaultGraphExecutor

This is the default graph executor. It can handle limited parallelism through graph adapters, and conducts execution using a simple recursive depth first traversal. Note this cannot handle parallelism with *Parallelizable[]*/*Collect[]*. Note that this is only exposed through the *Builder* (and it comes default on *Driver* instantiation) – it is here purely for documentation, and you should never need to instantiate it directly.

class hamilton.driver.DefaultGraphExecutor(*adapter: LifecycleAdapterSet | None = None*)

__init__(*adapter: LifecycleAdapterSet | None = None*)

Constructor for the default graph executor.

Parameters:

adapter – Adapter to use for execution (optional).

execute(*fg: FunctionGraph*, *final_vars: List[str]*, *overrides: Dict[str, Any]*, *inputs: Dict[str, Any]*, *run_id: str*) → Dict[str, Any]

Basic executor for a function graph. Does no task-based execution, just does a DFS and executes the graph in order, in memory.

validate(*nodes_to_execute: List[Node]*)

The default graph executor cannot handle parallelizable[]/collect[] nodes.

Parameters:

nodes_to_execute

Raises:

InvalidExecutorException – if the graph contains parallelizable[]/collect[] nodes.

TaskBasedGraphExecutor

This is a task based graph executor. It can handle parallelism with the *Parallelizable/Collect* constructs, allowing it to spawn dynamic tasks and execute them as a group. Note that this is only exposed through the *Builder* when called with *enable_dynamic_execution(allow_experimental_mode: bool)* – it is here purely for documentation, and you should never need to instantiate it directly.

class hamilton.driver.TaskBasedGraphExecutor(*execution_manager: ExecutionManager*, *grouping_strategy: GroupingStrategy*, *adapter: LifecycleAdapterSet*)

__init__(*execution_manager: ExecutionManager*, *grouping_strategy: GroupingStrategy*, *adapter: LifecycleAdapterSet*)

Executor for task-based execution. This enables grouping of nodes into tasks, as well as parallel execution/dynamic spawning of nodes.

Parameters:

- **execution_manager** – Utility to assign task executors to node groups
- **grouping_strategy** – Utility to group nodes into tasks
- **result_builder** – Utility to build the final result

execute(*fg: FunctionGraph*, *final_vars: List[str]*, *overrides: Dict[str, Any]*, *inputs: Dict[str, Any]*, *run_id: str*) → Dict[str, Any]

Executes a graph, task by task. This blocks until completion.

This does the following: 1. Groups the nodes into tasks 2. Creates an execution state and a results cache 3. Runs it to completion, populating the results cache 4. Returning the results from the results cache

validate(*nodes_to_execute: List[Node]*)

Currently this can run every valid graph

AsyncDriver

Use this driver in an async context. E.g. for use with FastAPI.

class hamilton.async_driver.AsyncDriver(*config*, **modules*, *result_builder: ResultMixin | None = None*, *adapters: List[BasePreDoAnythingHook | BaseDoCheckEdgeTypesMatch | BaseDoValidateInput | BaseValidateNode | BaseValidateGraph | BasePostGraphConstruct | BasePostGraphConstructAsync | BasePreGraphExecute | BasePreGraphExecuteAsync | BasePostTaskGroup | BasePostTaskExpand | BasePreTaskSubmission | BasePostTaskReturn | BasePreTaskExecute | BasePreTaskExecuteAsync | BasePreNodeExecute | BasePreNodeExecuteAsync | BaseDoNodeExecute | BaseDoNodeExecuteAsync | BasePostNodeExecute | BasePostNodeExecuteAsync | BasePostTaskExecute | BasePostTaskExecuteAsync | BasePostGraphExecute | BasePostGraphExecuteAsync | BaseDoBuildResult] = None*, *allow_module_overrides: bool = False*)

Async driver. This is a driver that uses the AsyncGraphAdapter to execute the graph.

```
dr = async_driver.AsyncDriver({}, async_module,
result_builder=base.DictResult())
df = await dr.execute([...], inputs=...)
```

__init__(*config*, **modules*, *result_builder: ResultMixin | None = None*, *adapters: List[BasePreDoAnythingHook | BaseDoCheckEdgeTypesMatch | BaseDoValidateInput | BaseValidateNode | BaseValidateGraph | BasePostGraphConstruct | BasePostGraphConstructAsync | BasePreGraphExecute | BasePreGraphExecuteAsync | BasePostTaskGroup | BasePostTaskExpand | BasePreTaskSubmission | BasePostTaskReturn | BasePreTaskExecute | BasePreTaskExecuteAsync | BasePreNodeExecute | BasePreNodeExecuteAsync | BaseDoNodeExecute | BaseDoNodeExecuteAsync | BasePostNodeExecute | BasePostNodeExecuteAsync | BasePostTaskExecute | BasePostTaskExecuteAsync | BasePostGraphExecute | BasePostGraphExecuteAsync | BaseDoBuildResult] = None*, *allow_module_overrides: bool = False*)

Instantiates an asynchronous driver.

You will also need to call *ainit* to initialize the driver if you have any hooks/adapters.

Note that this is not the desired API – you should be using the `hamilton.async_driver.Builder` class to create the driver.

This will only (currently) work properly with asynchronous lifecycle hooks, and does not support methods or validators. You can still pass in synchronous lifecycle hooks, but they may behave strangely.

Parameters:

- **config** – Config to build the graph
- **modules** – Modules to crawl for fns/graph nodes
- **result_builder** – Results mixin to compile the graph's final results. TBD whether this should be included in the long run.
- **adapters** – Adapters to use for lifecycle methods.
- **allow_module_overrides** – Optional. Same named functions get overridden by later modules. The order of listing the modules is important, since later ones will overwrite the previous ones. This is a global call affecting all imported modules. See https://github.com/apache/hamilton/tree/main/examples/module_overrides for more info.

async ainit() → AsyncDriver

Initializes the driver when using async. This only exists for backwards compatibility. In Hamilton 2.0, we will be using an asynchronous constructor. See https://dev.to/akarshan/asynchronous-python-magic-how-to-create-awaitable-constructors-with-asyncmixin-18j5.

capture_constructor_telemetry(*error: str | None*, *modules: Tuple[ModuleType]*, *config: Dict[str, Any]*, *adapter: HamiltonGraphAdapter*)

Ensures we capture constructor telemetry the right way in an async context.

This is a simpler wrapper around what's in the driver class.

Parameters:

- **error** – sanitized error string, if any.
- **modules** – tuple of modules to build DAG from.
- **config** – config to create the driver.

- **adapter** – adapter class object.

async execute(*final_vars: List[str]*, *overrides: Dict[str, Any] = None*, *display_graph: bool = False*, *inputs: Dict[str, Any] = None*) → Any

Executes computation.

Parameters:

- **final_vars** – the final list of variables we want to compute.
- **overrides** – values that will override "nodes" in the DAG.
- **display_graph** – DEPRECATED. Whether we want to display the graph being computed.
- **inputs** – Runtime inputs to the DAG.

Returns:

an object consisting of the variables requested, matching the type returned by the GraphAdapter. See constructor for how the GraphAdapter is initialized. The default one right now returns a pandas dataframe.

async raw_execute(*final_vars: List[str]*, *overrides: Dict[str, Any] = None*, *display_graph: bool = False*, *inputs: Dict[str, Any] = None*, *_fn_graph: FunctionGraph = None*) → Dict[str, Any]

Executes the graph, returning a dictionary of strings (node keys) to final results.

Parameters:

- **final_vars** – Variables to execute (+ upstream)
- **overrides** – Overrides for nodes
- **display_graph** – whether or not to display graph – this is not supported.
- **inputs** – Inputs for DAG runtime calculation
- **_fn_graph** – Function graph for compatibility with superclass – unused

Returns:

A dict of key -> result

Async Builder

Builds a driver in an async context – use `await builder....build()`.

class hamilton.async_driver.Builder

Builder for the async driver. This is equivalent to the standard builder, but has a more limited API. Note this does not support dynamic execution or materializers (for now).

Here is an example of how you might use it to get the tracker working:

```
from hamilton_sdk import tracker

tracker_async = adapters.AsyncHamiltonTracker(
    project_id=1,
    username="elijah",
    dag_name="async_tracker",
)
dr = (
    await async_driver.Builder()
    .with_modules(async_module)
    .with_adapters(tracking_async)
    .build()
)
```

__init__()

Constructs a driver builder. No parameters as you call methods to set fields.

async build()

Builds the async driver. This also initializes it, hence the async definition. If you don't want to use async, you can use *build_without_init* and call *ainit* later, but we recommend using this in an asynchronous lifespan management function (E.G. in fastAPI), or something similar.

Returns:

The fully

build_without_init() → AsyncDriver

Allows you to build the async driver without initialization. Use this at your own risk – we highly recommend calling *.ainit* on the final result.

Returns:

enable_dynamic_execution(*, *allow_experimental_mode: bool = False*) → Builder

Enables the Parallelizable[] type, which in turn enables: 1. Grouped execution into tasks 2. Parallel execution :return: self

with_adapter(*adapter: HamiltonGraphAdapter*) → Builder

Sets the adapter to use.

Parameters:

adapter – Adapter to use.

Returns:

self

with_materializers(**materializers: ExtractorFactory | MaterializerFactory*) → Builder

Add materializer nodes to the *Driver* The generated nodes can be referenced by name in *.execute()*

Parameters:

materializers – materializers to add to the dataflow

Returns:

self

Custom Driver

If you have a use case for a custom Driver, tell us on Slack or via a GitHub issues. Knowing about your use case and talking through help ensures we aren't duplicating effort, and that it'll be using part of the API we don't intend to change.

Caching

Reference

Caching logic

Caching Behavior

class hamilton.caching.adapter.CachingBehavior(*value*)

Behavior applied by the caching adapter

DEFAULT:

Try to retrieve result from cache instead of executing the node. If the node is executed, store the result. Compute the result data version and store it too.

RECOMPUTE:

Don't try to retrieve result from cache and always execute the node. Otherwise, behaves as default. Useful when nodes are stochastic (e.g., model training) or interact with external components (e.g., read from database).

DISABLE:

Node is executed as if the caching feature wasn't enabled. It never tries to retrieve results. Results are never stored nor versioned. Behaves like IGNORE, but the node remains a dependency for downstream nodes. This means downstream cache lookup will likely fail systematically (i.e., if the cache is empty).

IGNORE:

Node is executed as if the caching feature wasn't enable. It never tries to retrieve results. Results are never stored nor versioned. IGNORE means downstream nodes will ignore this node as a dependency for lookup. Ignoring clients and connections can be useful since they shouldn't directly impact the downstream results.

classmethod from_string(*string: str*) → CachingBehavior

Create a caching behavior from a string of the enum value. This is leveraged by the `hamilton.lifecycle.caching.SmartCacheAdapter` and the `hamilton.function_modifiers.metadata.cache` decorator.

```
CachingBehavior.from_string("recompute")
```

`@cache` decorator

class hamilton.function_modifiers.metadata.cache(*, *behavior: Literal['default', 'recompute', 'ignore', 'disable'] | None = None*, *format: Literal['json', 'file', 'pickle', 'parquet', 'csv', 'feather', 'orc', 'excel'] | str | None = None*, *target_: str | Collection[str] | None | EllipsisType = Ellipsis*)

BEHAVIOR_KEY = 'cache.behavior'

FORMAT_KEY = 'cache.format'

__init__(*, *behavior: Literal['default', 'recompute', 'ignore', 'disable'] | None = None*, *format: Literal['json', 'file', 'pickle', 'parquet', 'csv', 'feather', 'orc', 'excel'] | str | None = None*, *target_: str | Collection[str] | None | EllipsisType = Ellipsis*)

The `@cache` decorator can define the behavior and format of a specific node.

This feature is implemented via tags, but that could change. Thus you should not rely on these tags for other purposes.

```
@cache(behavior="recompute", format="parquet")
def raw_data() -> pd.DataFrame: ...
```

If the function uses other function modifiers and define multiple nodes, you can set `target_` to specify which nodes to cache. The following only caches the `performance` node.

```
@cache(format="json", target_="performance")
@extract_fields(trained_model=LinearRegression,
performance: dict)
def model_training() -> dict:
    # ...
    performance = {"rmse": 0.1, "mae": 0.2}
    return {"trained_model": trained_model, "performance":
performance}
```

Parameters:

- **behavior** – The behavior of the cache. This can be one of the following: * **default**: caching is enabled * **recompute**: always compute the node instead of retrieving * **ignore**: the data version won't be part of downstream keys * **disable**: act as if caching wasn't enabled.

- **format** – The format of the cache. This can be one of the following: * **json**: JSON format * **file**: file format * **pickle**: pickle format * **parquet**: parquet format * **csv**: csv format * **feather**: feather format * **orc**: orc format * **excel**: excel format

- **target_** – Target nodes to decorate. This can be one of the following: * **None**: tag all nodes outputted by this that are "final" (E.g. do not have a node outputted by this that depend on them) * **Ellipsis (...)**: tag *all* nodes outputted by this * **Collection[str]**: tag *only* the nodes with the specified names * **str**: tag *only* the node with the specified name

decorate_node(*node_: Node*) → Node

Decorates the nodes with the cache tags.

Parameters:

node – Node to decorate

Returns:

Copy of the node, with tags assigned

Logging

class hamilton.caching.adapter.CachingEvent(*run_id: str, actor: ~typing.Literal['adapter', 'metadata_store', 'result_store'], event_type: ~hamilton.caching.adapter.CachingEventType, node_name: str, task_id: str | None = None, msg: str | None = None, value: ~typing.Any | None = None, timestamp: float = <factory>*)

Event logged by the caching adapter

__init__(*run_id: str, actor: ~typing.Literal['adapter', 'metadata_store', 'result_store'], event_type: ~hamilton.caching.adapter.CachingEventType, node_name: str, task_id: str | None = None, msg: str | None = None, value: ~typing.Any | None = None, timestamp: float = <factory>*) → None

class hamilton.caching.adapter.CachingEventType(*value*)

Event types logged by the caching adapter

Adapter

class hamilton.caching.adapter.HamiltonCacheAdapter(*path: str | Path = '.hamilton_cache'*, *metadata_store: MetadataStore | None = None*, *result_store: ResultStore | None = None*, *default: Literal[True] | Collection[str] | None = None*, *recompute: Literal[True] | Collection[str] | None = None*, *ignore: Literal[True] | Collection[str] | None = None*, *disable: Literal[True] | Collection[str] | None = None*, *default_behavior: Literal['default', 'recompute', 'disable', 'ignore'] | None = None*, *default_loader_behavior: Literal['default', 'recompute', 'disable', 'ignore'] | None = None*, *default_saver_behavior: Literal['default', 'recompute', 'disable', 'ignore'] | None = None*, *log_to_file: bool = False*, ***kwargs*)

Adapter enabling Hamilton's caching feature through `Builder.with_cache()`

```
from hamilton import driver
import my_dataflow

dr = (
    driver.Builder()
    .with_modules(my_dataflow)
    .with_cache()
    .build()
)

# then, you can access the adapter via
dr.cache
```

__init__(*path: str | Path = '.hamilton_cache'*, *metadata_store: MetadataStore | None = None*, *result_store: ResultStore | None = None*, *default: Literal[True] | Collection[str] | None = None*, *recompute: Literal[True] | Collection[str] | None = None*, *ignore: Literal[True] | Collection[str] | None = None*, *disable: Literal[True] | Collection[str] | None = None*, *default_behavior: Literal['default', 'recompute', 'disable', 'ignore'] | None = None*, *default_loader_behavior: Literal['default', 'recompute', 'disable', 'ignore'] | None = None*, *default_saver_behavior: Literal['default', 'recompute', 'disable', 'ignore'] | None = None*, *log_to_file: bool = False*, ***kwargs*)

Initialize the cache adapter.

Parameters:

- **path** – path where the cache metadata and results will be stored
- **metadata_store** – BaseStore handling metadata for the cache adapter
- **result_store** – BaseStore caching dataflow execution results

- **default** – Set caching behavior to DEFAULT for specified node names. If True, apply to all nodes.
- **recompute** – Set caching behavior to RECOMPUTE for specified node names. If True, apply to all nodes.
- **ignore** – Set caching behavior to IGNORE for specified node names. If True, apply to all nodes.
- **disable** – Set caching behavior to DISABLE for specified node names. If True, apply to all nodes.
- **default_behavior** – Set the default caching behavior.
- **default_loader_behavior** – Set the default caching behavior *DataLoader* nodes.
- **default_saver_behavior** – Set the default caching behavior *DataSaver* nodes.
- **log_to_file** – If True, append cache event logs as they happen in JSONL format.

do_node_execute(*, *run_id: str*, *node_: Node*, *kwargs: Dict[str, Any]*, *task_id: str | None = None*, ***future_kwargs*)

Try to retrieve stored result from previous executions or execute the node.

Use the previously created cache_key to retrieve the data_version from memory or the metadata_store. If data_version is retrieved try to retrieve the result. If it fails, execute the node. Else, execute the node.

get_cache_key(*run_id: str*, *node_name: str*, *task_id: str | None = None*) → str | S

Get the `cache_key` stored in-memory for a specific `run_id`, `node_name`, and `task_id`.

This method is public-facing and can be used directly to inspect the cache.

Parameters:

- **run_id** – Id of the Hamilton execution run.
- **node_name** – Name of the node associated with the cache key. `node_name` is a unique identifier if task-based execution is not used.

- **task_id** – Id of the task when task-based execution is used. Then, the tuple `(node_name, task_id)` is a unique identifier.

Returns:

The cache key if it exists, otherwise return a sentinel value.

```
from hamilton import driver
import my_dataflow

dr =
driver.Builder().with_modules(my_dataflow).with_cache().build()
dr.execute(...)

dr.cache.get_cache_key(run_id=dr.last_run_id,
node_name="my_node", task_id=None)
```

get_data_version(*run_id: str*, *node_name: str*, *cache_key: str | None = None*, *task_id: str | None = None*) → str | S

Get the `data_version` for a specific `run_id`, `node_name`, and `task_id`.

This method is public-facing and can be used directly to inspect the cache. This will check data versions stored both in-memory and in the metadata store.

Parameters:

- **run_id** – Id of the Hamilton execution run.
- **node_name** – Name of the node associated with the data version. `node_name` is a unique identifier if task-based execution is not used.
- **task_id** – Id of the task when task-based execution is used. Then, the tuple `(node_name, task_id)` is a unique identifier.

Returns:

The data version if it exists, otherwise return a sentinel value.

..code-block:: python

from hamilton import driver import my_dataflow

dr = driver.Builder().with_modules(my_dataflow).with_cache().build() dr.execute(...)

dr.cache.get_data_version(run_id=dr.last_run_id, node_name="my_node", task_id=None)

property last_run_id

Run id of the last started run. Not necessarily the last to complete.

logs(*run_id: str | None = None*, *level: Literal['debug', 'info'] = 'info'*) → dict

Execution logs of the cache adapter.

Parameters:

- **run_id** – If `None`, return all logged runs. If provided a `run_id`, group logs by node.
- **level** – If `"debug"` log all events. If `"info"` only log if result is retrieved or executed.

Returns:

a mapping between node/task and a list of logged events

```
from hamilton import driver
import my_dataflow

dr =
driver.Builder().with_modules(my_dataflow).with_cache().build()
dr.execute(...)
dr.execute(...)

all_logs = dr.cache.logs()
# all_logs is a dictionary with run_ids as keys and lists of
CachingEvent as values.
# {
#    run_id_1: [CachingEvent(...), CachingEvent(...)],
#    run_id_2: [CachingEvent(...), CachingEvent(...)],
# }


run_logs = dr.cache.logs(run_id=dr.last_run_id)
# run_logs are keyed by ``node_name``
# {node_name: [CachingEvent(...), CachingEvent(...)], ...}
```

```
    # or ``(node_name, task_id)`` if task-based execution is
    used.
    # {(node_name_1, task_id_1): [CachingEvent(...),
    CachingEvent(...)], ...}
```

post_node_execute(*, *run_id: str*, *node_: Node*, *result: str | None*, *success: bool = True*, *error: Exception | None = None*, *task_id: str | None = None*, ***future_kwargs*)

Get the cache_key and data_version stored in memory (respectively from pre_node_execute and do_node_execute) and store the result in result_store if it doesn't exist.

pre_graph_execute(*, *run_id: str*, *graph: FunctionGraph*, *final_vars: List[str]*, *inputs: Dict[str, Any]*, *overrides: Dict[str, Any]*)

Set up the state of the adapter for a new execution.

Most attributes need to be keyed by run_id to prevent potential conflicts because the same adapter instance is shared between across all `Driver.execute()` calls.

pre_node_execute(*, *run_id: str*, *node_: Node*, *kwargs: Dict[str, Any]*, *task_id: str | None = None*, ***future_kwargs*)

Before node execution or retrieval, create the cache_key and set it in memory. The cache_key is created based on the node's code version and its dependencies' data versions.

Collecting `data_version` for upstream dependencies requires handling special cases when task-based execution is used: - If the current node is `COLLECT` , the dependency annotated with `Collect[]` needs to be versioned item by item instead of versioning the full container. This is because the collect order is inconsistent. - If the current node is `INSIDE` and the dependency is `EXPAND`, this means the `kwargs` dictionary contains a single item. We need to version this individual item because it will not be available from "inside" the branch for some executors (multiprocessing, multithreading) because they lose access to the data_versions of `OUTSIDE` nodes stored in `self.data_versions`.

resolve_behaviors(*run_id: str*) → Dict[str, CachingBehavior]

Resolve the caching behavior for each node based on the `@cache` decorator and the `Builder.with_cache()` parameters for a specific `run_id`.

This is a user-facing method.

Behavior specified via `Builder.with_cache()` have precedence. If no parameters are specified, the `CachingBehavior.DEFAULT` is used. If a node is `Parallelizable` (i.e., `@expand`), the `CachingBehavior` is set to `CachingBehavior.RECOMPUTE` to ensure the yielded items are versioned individually. Internally, this uses the `FunctionGraph` stored for each `run_id` and logs the resolved caching behavior for each node.

Parameters:

run_id – Id of the Hamilton execution run.

Returns:

A dictionary of `{node name: caching behavior}`.

resolve_code_versions(*run_id: str*, *final_vars: List[str] | None = None*, *inputs: Dict[str, Any] | None = None*, *overrides: Dict[str, Any] | None = None*) → Dict[str, str]

Resolve the code version for each node for a specific `run_id`.

This is a user-facing method.

If `final_vars` is None, all nodes will be versioned. If `final_vars` is provided, the `inputs` and `overrides` are used to determine the execution path and only version the code for these nodes.

Parameters:

- **run_id** – Id of the Hamilton execution run.
- **final_vars** – Nodes requested for execution.
- **inputs** – Input node values.
- **overrides** – Override node values.

Returns:

A dictionary of `{node name: code version}`.

version_code(*node_name: str*, *run_id: str | None = None*) → str

Create a unique code version for the source code defining the node

version_data(*result: Any*, *run_id: str = None*) → str

Create a unique data version for the result

This is a user-facing method.

view_run(*run_id: str | None = None*, *output_file_path: str | None = None*)

View the dataflow execution, including cache hits/misses.

Parameters:

- **run_id** – If `None`, view the last run. If provided a `run_id`, view that run.
- **output_file_path** – If provided a path, save the visualization to a file.

```
from hamilton import driver
import my_dataflow

dr =
driver.Builder().with_modules(my_dataflow).with_cache().build()

# execute 3 times
dr.execute(...)
dr.execute(...)
dr.execute(...)

# view the last run
dr.cache.view_run()
# this is equivalent to
dr.cache.view_run(run_id=dr.last_run_id)

# get a specific run id
run_id = dr.cache.run_ids[1]
dr.cache.view_run(run_id=run_id)
```

Quirks and limitations

Caching is a large and complex feature. This section is an attempt to list quirks and limitations, known and theoretical, to help debugging and guide feature development

- The standard library includes a lot of types which are not primitives. Thus, Apache Hamilton might not be supporting them explicitly. It should be simple to add, so ping us if you need it.
- The `ResultStore` could be architectured better to support custom formats. Right now, we use a `DataSaver` to produce the `.parquet` file and we pickle the `DataLoader` for later retrieval. Then, the metadata and result stores are completely unaware of the `.parquet` file making it difficult to handle cache eviction.
- When a function with default parameter values passes through lifecycle hooks, the default values are not part of the `node_kwargs`. They need to be retrieved manually from the `node.Node` object.

- supporting the Apache Hamilton `AsyncDriver` would require making the adapter async, but also the stores. A potential challenge is ensuring that you can use the same cache (i.e., same SQLite db and filesystem) for both sync and async drivers.
- If the `@cache` allows to specify the `format` (e.g., `json`, `parquet`), we probably want `.with_cache()` to support the same feature.
- Apache Hamilton allows a single `do_node_execute()` hook. Since the caching feature uses it, it is currently incompatible with other adapters leveraging it (`PDBDebugger`, `CacheAdapter` (deprecated), `GracefulErrorAdapter` (somewhat redundant with caching), `DiskCacheAdapter` (deprecated), `NarwhalsAdapter` (could be refactored))
- the presence of MD5 hashing can be seen as a security risk and prevent adoption. read more in DVC issues
- when hitting the base case of `fingerprinting.hash_value()` we return the constant `UNHASHABLE_VALUE`. If the adapter receives this value, it will append a random UUID to it. This is to prevent collision between unhashable types. This `data_version` is no longer deterministic, but the value can still be retrieved or be part of another node's `cache_key`.
- having `@functools.singledispatch(object)` allows to override the base case of `hash_value()` because it will catch all types.

Data versioning

This module contains hashing functions for Python objects. It uses functools.singledispatch to allow specialized implementations based on type. Singledispatch automatically applies the most specific implementation

This module houses implementations for the Python standard library. Supporting all types is considerable endeavor, so we'll add support as types are requested by users.

Otherwise, 3rd party types can be supported via the *h_databackends* module. This registers abstract types that can be checked without having to import the 3rd party library. For instance, there are implementations for pandas.DataFrame and polars.DataFrame despite these libraries not being imported here.

IMPORTANT all container types that make a recursive call to *hash_value* or a specific implementation should pass the *depth* parameter to prevent *RecursionError*.

hamilton.caching.fingerprinting.hash_bytes(*obj*, **args*, ***kwargs*) → str

Convert the primitive to a string and hash it

Primitive type returns a hash and doesn't have to handle depth.

hamilton.caching.fingerprinting.hash_mapping(*obj*, *, *ignore_order: bool = True*, *depth: int = 0*, ***kwargs*) → str

Hash each key then its value.

The mapping is always sorted first because order shouldn't matter in a mapping.

NOTE Since Python 3.7, dictionary store insertion order. However, this function assumes that they key order doesn't matter to uniquely identify the dictionary.

```
foo = {"key": 3, "key2": 13}
bar = {"key2": 13, "key": 3}

hash_mapping(foo) == hash_mapping(bar)
```

hamilton.caching.fingerprinting.hash_none(*obj*, **args*, ***kwargs*) → str

Hash for None is <none>

Primitive type returns a hash and doesn't have to handle depth.

hamilton.caching.fingerprinting.hash_numpy_array(*obj*, *args*, *depth: int = 0*, ***kwargs*) → str

Get the bytes representation of the array raw data and hash it.

Might not be ideal because different higher-level numpy objects could have the same underlying array representation (e.g., masked arrays). Unsure, but it's an area to investigate.

hamilton.caching.fingerprinting.hash_pandas_obj(*obj*, *args*, *depth: int = 0*, ***kwargs*) → str

Convert a pandas dataframe, series, or index to a dictionary of {index: row_hash} then hash it.

Given the hashing for mappings, the physical ordering or rows doesn't matter. For example, if the index is a date, the hash will represent the {date: row_hash}, and won't preserve how dates were ordered in the DataFrame.

hamilton.caching.fingerprinting.hash_polars_column(*obj*, *args*, *depth: int = 0*, ***kwargs*) → str

Promote the single Series to a dataframe and hash it

hamilton.caching.fingerprinting.hash_polars_dataframe(*obj*, *args*, *depth: int = 0*, ***kwargs*) → str

Convert a polars dataframe, series, or index to a list of hashes then hash it.

hamilton.caching.fingerprinting.hash_primitive(*obj*, *args*, ***kwargs*) → str

Convert the primitive to a string and hash it

Primitive type returns a hash and doesn't have to handle depth.

hamilton.caching.fingerprinting.hash_repr(*obj*, *args*, ***kwargs*) → str

Use the built-in repr() to get a string representation of the object and hash it.

While .__repr__() might not be implemented for all classes, the function *repr()* will handle it, along with exceptions, to always return a value.

Primitive type returns a hash and doesn't have to handle depth.

hamilton.caching.fingerprinting.hash_sequence(*obj*, **args*, *depth: int = 0*, ***kwargs*) → str

Hash each object of the sequence.

Orders matters for the hash since orders matters in a sequence.

hamilton.caching.fingerprinting.hash_set(*obj*, **args*, *depth: int = 0*, ***kwargs*) → str

Hash each element of the set, then sort hashes, and create a hash of hashes.

For the same objects in the set, the hashes will be the same.

hamilton.caching.fingerprinting.hash_unordered_mapping(*obj*, **args*, *depth: int = 0*, ***kwargs*) → str

When hashing an unordered mapping, the two following dict have the same hash.

```
foo = {"key": 3, "key2": 13}
bar = {"key2": 13, "key": 3}

hash_mapping(foo) == hash_mapping(bar)
```

hamilton.caching.fingerprinting.hash_value(*obj*, **args*, *depth=0*, ***kwargs*) → str
hamilton.caching.fingerprinting.hash_value(*obj: None*, **args*, ***kwargs*) → str
hamilton.caching.fingerprinting.hash_value(*obj: bool*, **args*, ***kwargs*) → str
hamilton.caching.fingerprinting.hash_value(*obj: float*, **args*, ***kwargs*) → str
hamilton.caching.fingerprinting.hash_value(*obj: int*, **args*, ***kwargs*) → str
hamilton.caching.fingerprinting.hash_value(*obj: str*, **args*, ***kwargs*) → str
hamilton.caching.fingerprinting.hash_value(*obj: bytes*, **args*, ***kwargs*) → str
hamilton.caching.fingerprinting.hash_value(*obj: Sequence*, **args*, *depth: int = 0*, ***kwargs*) → str
hamilton.caching.fingerprinting.hash_value(*obj: Mapping*, ***, *ignore_order: bool = True*, *depth: int = 0*, ***kwargs*) → str
hamilton.caching.fingerprinting.hash_value(*obj: Set*, **args*, *depth: int = 0*, ***kwargs*) → str
hamilton.caching.fingerprinting.hash_value(*obj: AbstractPandasColumn*, **args*, *depth: int = 0*, ***kwargs*) → str
hamilton.caching.fingerprinting.hash_value(*obj: AbstractPandasDataFrame*, **args*, *depth: int = 0*, ***kwargs*) → str
hamilton.caching.fingerprinting.hash_value(*obj: AbstractPolarsDataFrame*, **args*, *depth: int = 0*, ***kwargs*) → str
hamilton.caching.fingerprinting.hash_value(*obj: AbstractPolarsColumn*, **args*, *depth: int = 0*, ***kwargs*) → str
hamilton.caching.fingerprinting.hash_value(*obj: AbstractNumpyArray*, **args*, *depth: int = 0*, ***kwargs*) → str

Fingerprinting strategy that computes a hash of the full Python object.

The default case hashes the *__dict__* attribute of the object (recursive).

hamilton.caching.fingerprinting.set_max_depth(*depth: int*) → None

Set the maximum recursion depth for fingerprinting non-supported types.

Parameters:

depth – The maximum depth for fingerprinting.

Stores

stores.base

class hamilton.caching.stores.base.MetadataStore

abstractmethod delete(*cache_key: str*) → None

Delete `data_version` keyed by `cache_key`.

abstractmethod delete_all() → None

Delete all stored metadata.

abstractmethod exists(*cache_key: str*) → bool

boolean check if a `data_version` is found for `cache_key` If True, `.get()` should successfully retrieve the `data_version`.

abstractmethod get(*cache_key: str*, ***kwargs*) → str | None

Try to retrieve `data_version` keyed by `cache_key`. If retrieval misses return `None`.

get_last_run() → Any

Return the metadata from the last started run.

abstractmethod get_run(*run_id: str*) → Sequence[dict]

Return a list of node metadata associated with a run.

For each node, the metadata should include `cache_key` (created or used) and `data_version`. These values allow to manually query the MetadataStore or ResultStore.

Decoding the `cache_key` gives the `node_name`, `code_version`, and `dependencies_data_versions`. Individual implementations may add more information or decode the `cache_key` before returning metadata.

abstractmethod get_run_ids() → Sequence[str]

Return a list of run ids, sorted from oldest to newest start time. A `run_id` is registered when the metadata_store `.initialize()` is called.

abstractmethod initialize(*run_id: str*) → None

Setup the metadata store and log the start of the run

property last_run_id: str

Return

abstractmethod set(*cache_key: str*, *data_version: str*, ***kwargs*) → Any | None

Store the mapping `cache_key -> data_version`. Can include other metadata (e.g., node name, run id, code version) depending on the implementation.

property size: int

Number of unique entries (i.e., cache_keys) in the metadata_store

exception hamilton.caching.stores.base.ResultRetrievalError

Raised by the SmartCacheAdapter when ResultStore.get() fails.

class hamilton.caching.stores.base.ResultStore

abstractmethod delete(*data_version: str*) → None

Delete `result` keyed by `data_version`.

abstractmethod delete_all() → None

Delete all stored results.

abstractmethod exists(*data_version: str*) → bool

boolean check if a `result` is found for `data_version` If True, `.get()` should successfully retrieve the `result`.

abstractmethod get(*data_version: str*, ***kwargs*) → Any | None

Try to retrieve `result` keyed by `data_version`. If retrieval misses, return `None`.

abstractmethod set(*data_version: str*, *result: Any*, ***kwargs*) → None

Store `result` keyed by `data_version`.

hamilton.caching.stores.base.search_data_adapter_registry(*name: str*, *type_: type*) → Tuple[Type[DataSaver], Type[DataLoader]]

Find pair of DataSaver and DataLoader registered with *name* and supporting *type_*

stores.file

class hamilton.caching.stores.file.FileResultStore(*path: str*, *create_dir: bool = True*)

delete(*data_version: str*) → None

Delete `result` keyed by `data_version`.

delete_all() → None
Delete all stored results.

exists(*data_version: str*) → bool
boolean check if a `result` is found for `data_version` If True, `.get()` should successfully retrieve the `result`.

get(*data_version: str*) → Any | None
Try to retrieve `result` keyed by `data_version`. If retrieval misses, return `None`.

set(*data_version: str*, *result: Any*, *saver_cls: DataSaver | None = None*, *loader_cls: DataLoader | None = None*) → None
Store `result` keyed by `data_version`.

stores.sqlite

class hamilton.caching.stores.sqlite.SQLiteMetadataStore(*path: str*, *connection_kwargs: dict | None = None*)
property connection: Connection
Connection to the SQLite database.

delete(*cache_key: str*) → None
Delete metadata associated with `cache_key`.

delete_all() → None
Delete all existing tables from the database

exists(*cache_key: str*) → bool
boolean check if a `data_version` is found for `cache_key` If True, `.get()` should successfully retrieve the `data_version`.

get(*cache_key: str*) → str | None
Try to retrieve `data_version` keyed by `cache_key`. If retrieval misses return `None`.

get_run(*run_id: str*) → List[dict]
Return a list of node metadata associated with a run.

Parameters:

run_id – ID of the run to retrieve

Returns:

List of node metadata which includes `cache_key`, `data_version`, `node_name`, and `code_version`. The list can be empty if a run was initialized but no nodes were executed.

Raises:

IndexError – if the `run_id` is not found in metadata store.

get_run_ids() → List[str]
Return a list of run ids, sorted from oldest to newest start time.

initialize(*run_id*) → None
Call initialize when starting a run. This will create database tables if necessary.

set(*, *cache_key: str*, *data_version: str*, *run_id: str*, *node_name: str = None*, *code_version: str = None*, ***kwargs*) → None
Store the mapping `cache_key -> data_version`. Can include other metadata (e.g., node name, run id, code version) depending on the implementation.

stores.memory

class hamilton.caching.stores.memory.InMemoryMetadataStore
delete(*cache_key: str*) → None
Delete the `data_version` for `cache_key`.

delete_all() → None
Delete all stored metadata.

exists(*cache_key: str*) → bool
Indicate if `cache_key` exists and it can retrieve a `data_version`.

get(*cache_key: str*) → str | None
Retrieve the `data_version` for `cache_key`.

get_run(*run_id: str*) → List[Dict[str, str]]
Return a list of node metadata associated with a run.

get_run_ids() → List[str]
Return a list of all `run_id` values stored.

initialize(*run_id: str*) → None
Set up and log the beginning of the run.

classmethod load_from(*metadata_store: MetadataStore*) → InMemoryMetadataStore

Load in-memory metadata from another MetadataStore instance.

Parameters:

metadata_store – MetadataStore instance to load from.

Returns:

InMemoryMetadataStore copy of the `metadata_store`.

```
from hamilton import driver
from hamilton.caching.stores.sqlite import
SQLiteMetadataStore
from hamilton.caching.stores.memory import
InMemoryMetadataStore
import my_dataflow

sqlite_metadata_store =
SQLiteMetadataStore(path="./.hamilton_cache")
in_memory_metadata_store =
InMemoryMetadataStore.load_from(sqlite_metadata_store)

# create the Driver with the in-memory metadata store
dr = (
  driver.Builder()
  .with_modules(my_dataflow)
  .with_cache(metadata_store=in_memory_metadata_store)
  .build()
)
```

persist_to(*metadata_store: MetadataStore | None = None*) → None

Persist in-memory metadata using another MetadataStore implementation.

Parameters:

metadata_store – MetadataStore implementation to use for persistence. If None, a SQLiteMetadataStore is created with the default path "./.hamilton_cache".

```
from hamilton import driver
from hamilton.caching.stores.sqlite import
SQLiteMetadataStore
from hamilton.caching.stores.memory import
InMemoryMetadataStore
```

```
import my_dataflow

dr = (
  driver.Builder()
  .with_modules(my_dataflow)
  .with_cache(metadata_store=InMemoryMetadataStore())
  .build()
)

# execute the Driver several time. This will populate the
in-memory metadata store
dr.execute(...)

# persist to disk in-memory metadata
dr.cache.metadata_store.persist_to(SQLiteMetadataStore(path="./.hamilto
```

set(*cache_key: str*, *data_version: str*, *run_id: str*, ***kwargs*) → Any | None

Set the `data_version` for `cache_key` and associate it with the `run_id`.

class hamilton.caching.stores.memory.InMemoryResultStore(*persist_on_exit: bool = False*)

delete(*data_version: str*) → None

Delete `result` keyed by `data_version`.

delete_all() → None

Delete all stored results.

exists(*data_version: str*) → bool

boolean check if a `result` is found for `data_version` If True, `.get()` should successfully retrieve the `result`.

get(*data_version: str*) → Any | None

Try to retrieve `result` keyed by `data_version`. If retrieval misses, return `None`.

classmethod load_from(*result_store: ResultStore*, *metadata_store: MetadataStore | None = None*, *data_versions: Sequence[str] | None = None*) → InMemoryResultStore

Load in-memory results from another ResultStore instance.

Since result stores do not store an index of their keys, you must provide a `MetadataStore` instance or a list of `data_version` for which results should be loaded in memory.

Parameters:

- **result_store** – `ResultStore` instance to load results from.
- **metadata_store** – `MetadataStore` instance from which all `data_version` are retrieved.

Returns:

InMemoryResultStore copy of the `result_store`.

```
from hamilton import driver
from hamilton.caching.stores.sqlite import
SQLiteMetadataStore
from hamilton.caching.stores.memory import
InMemoryMetadataStore
import my_dataflow

sqlite_metadata_store =
SQLiteMetadataStore(path="./.hamilton_cache")
in_memory_metadata_store =
InMemoryMetadataStore.load_from(sqlite_metadata_store)

# create the Driver with the in-memory metadata store
dr = (
  driver.Builder()
  .with_modules(my_dataflow)
  .with_cache(metadata_store=in_memory_metadata_store)
  .build()
)
```

persist_to(*result_store: ResultStore | None = None*) → None

Persist in-memory results using another `ResultStore` implementation.

Parameters:

result_store – ResultStore implementation to use for persistence. If None, a FileResultStore is created with the default path "./.hamilton_cache".

set(*data_version: str*, *result: Any*, ***kwargs*) → None

Store `result` keyed by `data_version`.

GraphAdapters

This section helps determine ways to execute Apache Hamilton. Note that these are special cases of the Lifecycle Adapters meant to help with execution. They implement multiple lifecycle customizations in a single place.

Reference

SimplePythonDataFrameGraphAdapter

class hamilton.base.SimplePythonDataFrameGraphAdapter

This is the original Hamilton graph adapter. It uses plain python and builds a dataframe result.

This executes the Hamilton dataflow locally on a machine in a single threaded, single process fashion. It assumes a pandas dataframe as a result.

Use this when you want to execute on a single machine, without parallelization, and you want a pandas dataframe as output.

static check_input_type(*node_type: Type*, *input_value: Any*) → bool

Used to check whether the user inputs match what the execution strategy & functions can handle.

Static purely for legacy reasons.

Parameters:

- **node_type** – The type of the node.
- **input_value** – An actual value that we want to inspect matches our expectation.

Returns:

True if the input is valid, False otherwise.

static check_node_type_equivalence(*node_type: Type*, *input_type: Type*) → bool

Used to check whether two types are equivalent.

Static, purely for legacy reasons.

This is used when the function graph is being created and we're statically type checking the annotations for compatibility.

Parameters:

- **node_type** – The type of the node.
- **input_type** – The type of the input that would flow into the node.

Returns:

True if the types are equivalent, False otherwise.

execute_node(*node: Node*, *kwargs: Dict[str, Any]*) → Any

Given a node that represents a hamilton function, execute it. Note, in some adapters this might just return some type of "future".

Parameters:

- **node** – the Hamilton Node
- **kwargs** – the kwargs required to exercise the node function.

Returns:

the result of exercising the node.

SimplePythonGraphAdapter

class hamilton.base.SimplePythonGraphAdapter(*result_builder: ResultMixin = None*)

This class allows you to swap out the build_result very easily.

This executes the Hamilton dataflow locally on a machine in a single threaded, single process fashion. It allows you to specify a ResultBuilder to control the return type of what `execute()` returns.

Currently this extends SimplePythonDataFrameGraphAdapter, although that's largely for legacy reasons (and can probably be changed).

TODO – change this to extend the right class.

__init__(*result_builder: ResultMixin = None*)

Allows you to swap out the build_result very easily.

Parameters:

result_builder – A ResultMixin object that will be used to build the result.

static build_dataframe_with_dataframes(*outputs: Dict[str, Any]*) → DataFrame

Builds a dataframe from the outputs in an "outer join" manner based on index.

The behavior of pd.Dataframe(outputs) is that it will do an outer join based on indexes of the Series passed in. To handle dataframes, we unpack the dataframe into a dict of series, check to ensure that no columns are redefined in a rolling fashion going in order of the outputs requested. This then results in an "enlarged" outputs dict that is then passed to pd.Dataframe(outputs) to get the final dataframe.

Parameters:

outputs – The outputs to build the dataframe from.

Returns:

A dataframe with the outputs.

build_result(***outputs: Dict[str, Any]*) → Any

Delegates to the result builder function supplied.

static check_input_type(*node_type: Type*, *input_value: Any*) → bool

Used to check whether the user inputs match what the execution strategy & functions can handle.

Static purely for legacy reasons.

Parameters:

- **node_type** – The type of the node.
- **input_value** – An actual value that we want to inspect matches our expectation.

Returns:

True if the input is valid, False otherwise.

static check_node_type_equivalence(*node_type: Type*, *input_type: Type*) → bool

Used to check whether two types are equivalent.

Static, purely for legacy reasons.

This is used when the function graph is being created and we're statically type checking the annotations for compatibility.

Parameters:

- **node_type** – The type of the node.
- **input_type** – The type of the input that would flow into the node.

Returns:

True if the types are equivalent, False otherwise.

static check_pandas_index_types_match(*all_index_types: Dict[str, List[str]]*, *time_indexes: Dict[str, List[str]]*, *no_indexes: Dict[str, List[str]]*) → bool

Checks that pandas index types match.

This only logs warning errors, and if debug is enabled, a debug statement to list index types.

do_build_result(*outputs: Dict[str, Any]*) → Any

Implements the do_build_result method from the BaseDoBuildResult class. This is kept from the user as the public-facing API is build_result, allowing us to change the API/implementation of the internal set of hooks

do_check_edge_types_match(*type_from: type*, *type_to: type*) → bool

Method that checks whether two types are equivalent. This is used when the function graph is being created.

Parameters:

- **type_from** – The type of the node that is the source of the edge.
- **type_to** – The type of the node that is the destination of the edge.

Return bool:

Whether or not they are equivalent

do_node_execute(*run_id: str*, *node_: Node*, *kwargs: Dict[str, Any]*, *task_id: str | None = None*) → Any

Method that is called to implement node execution. This can replace the execution of a node with something all together, augment it, or delegate it.

Parameters:

- **run_id** – ID of the run, unique in scope of the driver.
- **node** – Node that is being executed
- **kwargs** – Keyword arguments that are being passed into the node
- **task_id** – ID of the task, defaults to None if not in a task setting

do_validate_input(*node_type: type*, *input_value: Any*) → bool

Method that an input value maches an expected type.

Parameters:

- **node_type** – The type of the node.
- **input_value** – The value that we want to validate.

Returns:

Whether or not the input value matches the expected type.

execute_node(*node: Node*, *kwargs: Dict[str, Any]*) → Any

Given a node that represents a hamilton function, execute it. Note, in some adapters this might just return some type of "future".

Parameters:

- **node** – the Hamilton Node
- **kwargs** – the kwargs required to exercise the node function.

Returns:

the result of exercising the node.

input_types() → List[Type[Type]]

Currently this just shoves anything into a dataframe. We should probably tighten this up.

output_type() → Type

Returns the output type of this result builder :return: the type that this creates

static pandas_index_types(*outputs: Dict[str, Any]*) → Tuple[Dict[str, List[str]], Dict[str, List[str]], Dict[str, List[str]]]

This function creates three dictionaries according to whether there is an index type or not.

The three dicts we create are: 1. Dict of index type to list of outputs that match it. 2. Dict of time series / categorical index types to list of outputs that match it. 3. Dict of *no-index* key to list of outputs with no index type.

Parameters:

outputs – the dict we're trying to create a result from.

Returns:

dict of all index types, dict of time series/categorical index types, dict if there is no index

HamiltonGraphAdapter

Graph adapters control how functions are executed as the graph is walked.

class hamilton.base.HamiltonGraphAdapter

Legacy graph adapter – see lifecycle methods for more information.

h_async.AsyncGraphAdapter

class hamilton.async_driver.AsyncGraphAdapter(*result_builder: ResultMixin = None*, *async_lifecycle_adapters: LifecycleAdapterSet | None = None*)

Graph adapter for use with the `AsyncDriver` class.

__init__(*result_builder: ResultMixin = None*, *async_lifecycle_adapters: LifecycleAdapterSet | None = None*)

Creates an AsyncGraphAdapter class. Note this will *only* work with the AsyncDriver class.

Some things to note:

1. This executes everything at the end (recursively). E.G. the final DAG nodes are awaited
2. This does *not* work with decorators when the async function is being decorated. That is because that function is called directly within the decorator, so we cannot await it.

build_result(***outputs: Any*) → Any

Given a set of outputs, build the result.

Parameters:

outputs – the outputs from the execution of the graph.

Returns:

the result of the execution of the graph.

do_build_result(*outputs: Dict[str, Any]*) → Any

Implements the do_build_result method from the BaseDoBuildResult class. This is kept from the user as the public-facing API is build_result, allowing us to change the API/implementation of the internal set of hooks

do_node_execute(***, *run_id: str*, *node_: Node*, *kwargs: Dict[str, Any]*, *task_id: str | None = None*) → Any

Executes a node. Note this doesn't actually execute it – rather, it returns a task. This does *not* use async def, as we want it to be awaited on later – this await is done in

processing parameters of downstream functions/final results. We can ensure that as we also run the driver that this corresponds to.

Note that this assumes that everything is awaitable, even if it isn't. In that case, it just wraps it in one.

Parameters:

- **task_id**
- **node**
- **run_id**
- **node** – Node to wrap
- **kwargs** – Keyword arguments (either coroutines or raw values) to call it with

Returns:

A task

input_types() → List[Type[Type]]

Gives the applicable types to this result builder. This is optional for backwards compatibility, but is recommended.

Returns:

A list of types that this can apply to.

output_type() → Type

Returns the output type of this result builder :return: the type that this creates

h_threadpool.FutureAdapter

This is an adapter to delegate execution of the individual nodes in a Apache Hamilton graph to a threadpool. This is useful when you have a graph with many nodes that can be executed in parallel.

class hamilton.plugins.h_threadpool.FutureAdapter(*max_workers: int = None*, *thread_name_prefix: str = ''*, *result_builder: ResultBuilder = None*)

Adapter that lazily submits each function for execution to a ThreadpoolExecutor.

This adapter has similar behavior to the async Hamilton driver which allows for parallel execution of functions.

This adapter works because we don't have to worry about object serialization.

Caveats: - DAGs with lots of CPU intense functions will limit usefulness of this adapter, unless they release the GIL. - DAGs with lots of I/O bound work will benefit from this adapter, e.g. making API calls. - The max parallelism is limited by the number of threads in the ThreadPoolExecutor.

Unsupported behavior: - The FutureAdapter does not support DAGs with Parallelizable & Collect functions. This is due to laziness rather than anything inherently technical. If you'd like this feature, please open an issue on the Hamilton repository.

__init__(*max_workers: int = None*, *thread_name_prefix: str = ''*, *result_builder: ResultBuilder = None*)

Constructor. :param max_workers: The maximum number of threads that can be used to execute the given calls. :param thread_name_prefix: An optional name prefix to give our threads. :param result_builder: Optional. Result builder to use for building the result.

build_result(***outputs: Any*) → Any

Given a set of outputs, build the result.

This function will block until all futures are resolved.

Parameters:

outputs – the outputs from the execution of the graph.

Returns:

the result of the execution of the graph.

do_build_result(*outputs: Dict[str, Any]*) → Any

Implements the do_build_result method from the BaseDoBuildResult class. This is kept from the user as the public-facing API is build_result, allowing us to change the API/implementation of the internal set of hooks

do_remote_execute(*, *execute_lifecycle_for_node: Callable*, *node: Node*, ***kwargs: Dict[str, Any]*) → Any

Function that submits the passed in function to the ThreadPoolExecutor to be executed after wrapping it with the _new_fn function.

Parameters:

- **node** – Node that is being executed
- **execute_lifecycle_for_node** – Function executing lifecycle_hooks and lifecycle_methods
- **kwargs** – Keyword arguments that are being passed into the function

input_types() → List[Type[Type]]

Gives the applicable types to this result builder. This is optional for backwards compatibility, but is recommended.

Returns:

A list of types that this can apply to.

output_type() → Type

Returns the output type of this result builder :return: the type that this creates

CachingGraphAdapter

This is an experimental GraphAdapter; there is a possibility of their API changing. That said, the code is stable, and you should feel comfortable giving the code for a spin - let us know how it goes, and what the rough edges are if you find any. We'd love feedback if you are using these to know how to improve them or graduate them.

class hamilton.experimental.h_cache.CachingGraphAdapter(*cache_path: str*, **args*, *force_compute: Set[str] | None = None*, *writers: Dict[str, Callable[[Any, str, str], None]] | None = None*, *readers: Dict[str, Callable[[Any, str], Any]] | None = None*, ***kwargs*)

Caching adapter.

Any node with tag "cache" will be cached (or loaded from cache) in the format defined by the tag's value. There are a handful of formats supported, and other formats' readers and writers can be provided to the constructor.

Values are loaded from cache if the node's file exists, unless one of these is true:

- node is explicitly forced to be computed with a constructor argument,
- any of its (potentially transitive) dependencies that are configured to be cached was nevertheless computed (either forced or missing cached file).

Custom Serializers

One can provide custom readers and writers for any format by passing them to the constructor. These readers and writers will override the default ones. If you don't want to override, but rather extend the default ones, you can do so by registering them with the *register* method on the appropriate function.

Writer functions need to have the following signature: *def write_<format>(data: Any, filepath: str, name: str) -> None:* ... where *data* is the data to be written, *filepath* is the path to the file to be written to, and *name* is the name of the node that is being written.

Reader functions need to have the following signature: *def read_<format>(data: Any, filepath: str) -> Any:* ... where *data* is an EMPTY OBJECT of the type you wish to instantiate, and *filepath* is the path to the file to be read from.

For example, if you want to extend JSON reader/writer to work with your custom type *T*, you can do the following:

```
@write_json.register(T)
def write_json_pd1(data: T, filepath: str, name: str) ->
None: ...


@read_json.register(T)
def read_json_dict(data: T, filepath: str) -> T: ...
```

Usage

This is a simple example of the usage of *CachingGraphAdapter*.

First, let's define some nodes in *nodes.py*:

```
import pandas as pd
from hamilton.function_modifiers import tag


def data_a() -> pd.DataFrame: ...


@tag(cache="parquet")
def data_b() -> pd.DataFrame: ...


def transformed(data_a: pd.DataFrame, data_b: pd.DataFrame) ->
pd.DataFrame: ...
```

Notice that *data_b* is configured to be cached in a parquet file.

We then simply initialize the driver with a caching adapter:

```
from hamilton import base
from hamilton.driver import Driver
from hamilton.experimental import h_cache

import nodes

adapter = h_cache.CachingGraphAdapter(cache_path,
base.PandasDataFrameResult())
dr = Driver(config, nodes, adapter=adapter)
result = dr.execute(["transformed"])

# Because `data_b` has been cached now, only `data_a` and
`transformed` nodes
# will actually run.
result = dr.execute(["transformed"])
```

__init__(*cache_path: str*, **args*, *force_compute: Set[str] | None = None*, *writers: Dict[str, Callable[[Any, str, str], None]] | None = None*, *readers: Dict[str, Callable[[Any, str], Any]] | None = None*, ***kwargs*)

Constructs the adapter.

Parameters:

- **cache_path** – Path to the directory where cached files are stored.
- **force_compute** – Set of nodes that should be forced to compute even if cache exists.
- **writers** – A dictionary of writers for custom formats.
- **readers** – A dictionary of readers for custom formats.

static build_dataframe_with_dataframes(*outputs: Dict[str, Any]*) → DataFrame

Builds a dataframe from the outputs in an “outer join” manner based on index.

The behavior of pd.Dataframe(outputs) is that it will do an outer join based on indexes of the Series passed in. To handle dataframes, we unpack the dataframe into a dict of series, check to ensure that no columns are redefined in a rolling fashion going in order of the outputs requested. This then results in an “enlarged” outputs dict that is then passed to pd.Dataframe(outputs) to get the final dataframe.

Parameters:

outputs – The outputs to build the dataframe from.

Returns:

A dataframe with the outputs.

build_result(***outputs: Dict[str, Any]*) → Any

Clears the computed nodes information and delegates to the super class.

static check_input_type(*node_type: Type*, *input_value: Any*) → bool

Used to check whether the user inputs match what the execution strategy & functions can handle.

Static purely for legacy reasons.

Parameters:

- **node_type** – The type of the node.
- **input_value** – An actual value that we want to inspect matches our expectation.

Returns:

True if the input is valid, False otherwise.

static check_node_type_equivalence(*node_type: Type*, *input_type: Type*) → bool

Used to check whether two types are equivalent.

Static, purely for legacy reasons.

This is used when the function graph is being created and we're statically type checking the annotations for compatibility.

Parameters:

- **node_type** – The type of the node.
- **input_type** – The type of the input that would flow into the node.

Returns:

True if the types are equivalent, False otherwise.

static check_pandas_index_types_match(*all_index_types: Dict[str, List[str]]*, *time_indexes: Dict[str, List[str]]*, *no_indexes: Dict[str, List[str]]*) → bool

Checks that pandas index types match.

This only logs warning errors, and if debug is enabled, a debug statement to list index types.

do_build_result(*outputs: Dict[str, Any]*) → Any

Implements the do_build_result method from the BaseDoBuildResult class. This is kept from the user as the public-facing API is build_result, allowing us to change the API/implementation of the internal set of hooks

do_check_edge_types_match(*type_from: type*, *type_to: type*) → bool

Method that checks whether two types are equivalent. This is used when the function graph is being created.

Parameters:

- **type_from** – The type of the node that is the source of the edge.
- **type_to** – The type of the node that is the destination of the edge.

Return bool:

Whether or not they are equivalent

do_node_execute(*run_id: str*, *node_: Node*, *kwargs: Dict[str, Any]*, *task_id: str | None = None*) → Any

Method that is called to implement node execution. This can replace the execution of a node with something all together, augment it, or delegate it.

Parameters:

- **run_id** – ID of the run, unique in scope of the driver.
- **node** – Node that is being executed
- **kwargs** – Keyword arguments that are being passed into the node
- **task_id** – ID of the task, defaults to None if not in a task setting

do_validate_input(*node_type: type*, *input_value: Any*) → bool

Method that an input value maches an expected type.

Parameters:

- **node_type** – The type of the node.
- **input_value** – The value that we want to validate.

Returns:

Whether or not the input value matches the expected type.

execute_node(*node: Node*, *kwargs: Dict[str, Any]*) → Any

Executes nodes conditionally according to caching rules.

This node is executed if at least one of these is true:

- no cache is present,
- it is explicitly forced by passing it to the adapter in `force_compute`,
- at least one of its upstream nodes that had a @cache annotation was computed, either due to lack of cache or being explicitly forced.

input_types() → List[Type[Type]]

Currently this just shoves anything into a dataframe. We should probably tighten this up.

output_type() → Type

Returns the output type of this result builder :return: the type that this creates

static pandas_index_types(*outputs: Dict[str, Any]*) → Tuple[Dict[str, List[str]], Dict[str, List[str]], Dict[str, List[str]]]

This function creates three dictionaries according to whether there is an index type or not.

The three dicts we create are: 1. Dict of index type to list of outputs that match it. 2. Dict of time series / categorical index types to list of outputs that match it. 3. Dict of *no-index* key to list of outputs with no index type.

Parameters:

outputs – the dict we're trying to create a result from.

Returns:

dict of all index types, dict of time series/categorical index types, dict if there is no index

h_dask.DaskGraphAdapter

Runs the entire Hamilton DAG on dask.

class hamilton.plugins.h_dask.DaskGraphAdapter(*dask_client: Client*, *result_builder: ResultMixin = None*, *visualize_kwargs: dict = None*, *use_delayed: bool = True*, *compute_at_end: bool = True*)

Class representing what's required to make Hamilton run on Dask.

This walks the graph and translates it to run onto Dask.

Use *pip install sf-hamilton[dask]* to get the dependencies required to run this.

Try this adapter when:

1. Dask is a good choice to scale computation when you really can't do things in memory anymore with pandas. For most simple pandas operations, you should not have to do anything to scale! You just need to load in data via dask rather than pandas.
2. Dask can help scale to larger data sets if running on a cluster – you'll just have to switch to natively using their object types if that's the case (set use_delayed=False, and compute_at_end=False).
3. Use this adapter if you want to utilize multiple cores on a single machine, or you want to scale to large data set sizes with a Dask cluster that you can connect to.
4. The ONLY CAVEAT really is whether you use *delayed* or *dask datatypes* (or both).

Please read the following notes about its limitations.

Notes on scaling:

- Multi-core on single machine ✅
- Distributed computation on a Dask cluster ✅
- Scales to any size of data supported by Dask ✅ assuming you load it appropriately via Dask loaders.

- Works best with Pandas 2.0+ and pyarrow backend.

Function return object types supported:

- Works for any python object that can be serialized by the Dask framework. ✅

Pandas?

Dask implements a good subset of the Pandas API:

- You might be able to get away with scaling without having to change your code at all!
- See https://docs.dask.org/en/latest/dataframe-api.html for Pandas supported APIs.
- If it is not supported by their API, you have to then read up and think about how to structure you hamilton function computation – https://docs.dask.org/en/latest/dataframe.html
- if paired with DaskDataFrameResult & use_delayed=False & compute_at_end=False, it will help you produce a dask dataframe as a result that you can then convert back to pandas if you want.

Loading Data:

- see https://docs.dask.org/en/latest/best-practices.html#load-data-with-dask.
- we recommend creating a python module specifically encapsulating functions that help you load data.

CAVEATS with use_delayed=True:

- If using *use_delayed=True* serialization costs can outweigh the benefits of parallelism, so you should benchmark your code to see if it's worth it.
- With this adapter & use_delayed=True, it can naively wrap all your functions with *delayed*, which will mean they will be executed and scheduled across the dask workers. This is a good choice if your computation is slow, or Hamilton graph is highly parallelizable.

DISCLAIMER – this class is experimental, so signature changes are a possibility! But we'll aim to be backwards compatible where possible.

__init__(*dask_client: Client*, *result_builder: ResultMixin = None*, *visualize_kwargs: dict = None*, *use_delayed: bool = True*, *compute_at_end: bool = True*)

Constructor

You have the ability to pass in a ResultMixin object to the constructor to control the return type that gets produced by running on Dask.

Parameters:

- **dask_client** – the dask client – we don't do anything with it, but thought that it would be useful to wire through here.
- **result_builder** – The function that will build the result. Optional, defaults to pandas dataframe.
- **visualize_kwargs** – Arguments to visualize the graph using dask's internals. **None**, means no visualization. **Dict**, means visualize – see https://docs.dask.org/en/latest/api.html?highlight=visualize#dask.visualize for what to pass in.
- **use_delayed** – Default is True for backwards compatibility. Whether to use dask.delayed to wrap every function. Note: it is probably not necessary to mix this with using dask objects, e.g. dataframes/series. They are by nature lazily computed and operate over the dask data types, so you don't need to wrap them with delayed. Use delayed if you want to farm out computation.
- **compute_at_end** – Default is True for backwards compatibility. Whether to compute() at the end. That is, should *.compute()* be called in the result builder to quick off computation.

build_result(***outputs: Dict[str, Any]*) → Any

Builds the result and brings it back to this running process.

Parameters:

outputs – the dictionary of key -> Union[delayed object reference | value]

Returns:

The type of object returned by self.result_builder. Note the following behaviors: - if you use_delayed=True, then the result will be a delayed object. - if you use_delayed=True & computed_at_end=True, then the result will be the return type of self.result_builder. - if you use_delayed=False & computed_at_end=True, this will only work if the self.result_builder returns a dask type, as we will try to compute it. - if you use_delayed=False & computed_at_end=False, this will return the result of self.result_builder.

static check_input_type(*node_type: Type*, *input_value: Any*) → bool

Used to check whether the user inputs match what the execution strategy & functions can handle.

Static purely for legacy reasons.

Parameters:

- **node_type** – The type of the node.
- **input_value** – An actual value that we want to inspect matches our expectation.

Returns:

True if the input is valid, False otherwise.

static check_node_type_equivalence(*node_type: Type*, *input_type: Type*) → bool

Used to check whether two types are equivalent.

Static, purely for legacy reasons.

This is used when the function graph is being created and we're statically type checking the annotations for compatibility.

Parameters:

- **node_type** – The type of the node.
- **input_type** – The type of the input that would flow into the node.

Returns:

True if the types are equivalent, False otherwise.

do_build_result(*outputs: Dict[str, Any]*) → Any

Implements the do_build_result method from the BaseDoBuildResult class. This is kept from the user as the public-facing API is build_result, allowing us to change the API/implementation of the internal set of hooks

do_check_edge_types_match(*type_from: type*, *type_to: type*) → bool

Method that checks whether two types are equivalent. This is used when the function graph is being created.

Parameters:

- **type_from** – The type of the node that is the source of the edge.
- **type_to** – The type of the node that is the destination of the edge.

Return bool:

Whether or not they are equivalent

do_node_execute(*run_id: str*, *node_: Node*, *kwargs: Dict[str, Any]*, *task_id: str | None = None*) → Any

Method that is called to implement node execution. This can replace the execution of a node with something all together, augment it, or delegate it.

Parameters:

- **run_id** – ID of the run, unique in scope of the driver.
- **node** – Node that is being executed
- **kwargs** – Keyword arguments that are being passed into the node
- **task_id** – ID of the task, defaults to None if not in a task setting

do_validate_input(*node_type: type*, *input_value: Any*) → bool

Method that an input value maches an expected type.

Parameters:

- **node_type** – The type of the node.
- **input_value** – The value that we want to validate.

Returns:

Whether or not the input value matches the expected type.

execute_node(*node: Node*, *kwargs: Dict[str, Any]*) → Any

Function that is called as we walk the graph to determine how to execute a hamilton function.

Parameters:

- **node** – the node from the graph.
- **kwargs** – the arguments that should be passed to it.

Returns:

returns a dask delayed object.

input_types() → List[Type[Type]]

Gives the applicable types to this result builder. This is optional for backwards compatibility, but is recommended.

Returns:

A list of types that this can apply to.

output_type() → Type

Returns the output type of this result builder :return: the type that this creates

h_spark.PySparkUDFGraphAdapter

This is an experimental GraphAdapter; there is a possibility of their API changing. That said, the code is stable, and you should feel comfortable giving the code for a spin - let us know how it goes, and what the rough edges are if you find any. We'd love feedback if you are using these to know how to improve them or graduate them.

class hamilton.plugins.h_spark.PySparkUDFGraphAdapter

UDF graph adapter for PySpark.

This graph adapter enables one to write Hamilton functions that can be executed as UDFs in PySpark.

Core to this is the mapping of function arguments to Spark columns available in the passed in dataframe.

This adapter currently supports:

- regular UDFs, these are executed in a row based fashion.
- and a single variant of Pandas UDFs: func(series+) -> series
- can also run regular Hamilton functions, which will execute spark driver side.

DISCLAIMER – this class is experimental, so signature changes are a possibility!

__init__()

build_result(***outputs: Dict[str, Any]*) → DataFrame

Builds the result and brings it back to this running process.

Parameters:

outputs – the dictionary of key -> Union[ray object reference | value]

Returns:

The type of object returned by self.result_builder.

static check_input_type(*node_type: Type*, *input_value: Any*) → bool

If the input is a pyspark dataframe, skip, else delegate the check.

static check_node_type_equivalence(*node_type: Type*, *input_type: Type*) → bool

Checks for the htype.column annotation and deals with it.

execute_node(*node: Node*, *kwargs: Dict[str, Any]*) → Any

Given a node to execute, process it and apply a UDF if applicable.

Parameters:

- **node** – the node we're processing.
- **kwargs** – the inputs to the function.

Returns:

the result of the function.

h_ray.RayGraphAdapter

The graph adapter to delegate execution of the individual nodes in a Apache Hamilton graph to Ray.

class hamilton.plugins.h_ray.RayGraphAdapter(*result_builder: ResultMixin***, *ray_init_config: Dict[str, Any] = None*, *shutdown_ray_on_completion: bool = False*)**

Class representing what's required to make Hamilton run on Ray.

This walks the graph and translates it to run onto Ray.

Use *pip install sf-hamilton[ray]* to get the dependencies required to run this.

Use this if:

- you want to utilize multiple cores on a single machine, or you want to scale to larger data set sizes with a Ray cluster that you can connect to. Note (1): you are still constrained by machine memory size with Ray; you can't just scale to any dataset size. Note (2): serialization costs can outweigh the benefits of parallelism so you should benchmark your code to see if it's worth it.

Notes on scaling:

- Multi-core on single machine ✅
- Distributed computation on a Ray cluster ✅
- Scales to any size of data ⛔ you are LIMITED by the memory on the instance/computer 💻.

Function return object types supported:

- Works for any python object that can be serialized by the Ray framework. ✅

Pandas?

- ⛔ Ray DOES NOT do anything special about Pandas.

CAVEATS

- Serialization costs can outweigh the benefits of parallelism, so you should benchmark your code to see if it's worth it.

DISCLAIMER – this class is experimental, so signature changes are a possibility!

__init__(*result_builder: ResultMixin*, *ray_init_config: Dict[str, Any] = None*, *shutdown_ray_on_completion: bool = False*)

Constructor

You have the ability to pass in a ResultMixin object to the constructor to control the return type that gets produce by running on Ray.

Parameters:

- **result_builder** – Required. An implementation of base.ResultMixin.
- **ray_init_config** – allows to connect to an existing cluster or start a new one with custom configuration (https://docs.ray.io/en/latest/ray-core/api/doc/ray.init.html)
- **shutdown_ray_on_completion** – by default we leave the cluster open, but we can also shut it down

do_build_result(*outputs: Dict[str, Any]*) → Any

Builds the result and brings it back to this running process.

Parameters:

outputs – the dictionary of key -> Union[ray object reference | value]

Returns:

The type of object returned by self.result_builder.

static do_check_edge_types_match(*type_from: Type*, *type_to: Type*) → bool

Method that checks whether two types are equivalent. This is used when the function graph is being created.

Parameters:

- **type_from** – The type of the node that is the source of the edge.
- **type_to** – The type of the node that is the destination of the edge.

Return bool:

Whether or not they are equivalent

do_remote_execute(*, *execute_lifecycle_for_node: Callable*, *node: Node*, ***kwargs: Dict[str, Any]*) → Any

Function that is called as we walk the graph to determine how to execute a hamilton function.

Parameters:

- **execute_lifecycle_for_node** – wrapper function that executes lifecycle hooks and methods
- **kwargs** – the arguments that should be passed to it.

Returns:

returns a ray object reference.

static do_validate_input(*node_type: Type*, *input_value: Any*) → bool

Method that an input value maches an expected type.

Parameters:

- **node_type** – The type of the node.
- **input_value** – The value that we want to validate.

Returns:

Whether or not the input value matches the expected type.

post_graph_execute(**args*, ***kwargs*)

We have the option to close the cluster down after execution.

h_spark.SparkKoalasGraphAdapter

This is an experimental GraphAdapter; there is a possibility of their API changing. That said, the code is stable, and you should feel comfortable giving the code for a spin - let us know how it goes, and what the rough edges are if you find any. We'd love feedback if you are using these to know how to improve them or graduate them.

class hamilton.plugins.h_spark.SparkKoalasGraphAdapter(*spark_session*, *result_builder: ResultMixin*, *spine_column: str*)

Class representing what's required to make Hamilton run on Spark with Koalas, i.e. Pandas on Spark.

This walks the graph and translates it to run onto Apache Spark using the Pandas API on Spark

Use *pip install sf-hamilton[spark]* to get the dependencies required to run this.

Currently, this class assumes you're running SPARK 3.2+. You'd generally use this if you have an existing spark cluster running in your workplace, and you want to scale to very large data set sizes.

Some tips on koalas (before it was merged into spark 3.2):

- https://databricks.com/blog/2020/03/31/10-minutes-from-pandas-to-koalas-on-apache-spark.html
- https://spark.apache.org/docs/latest/api/python/user_guide/pandas_on_spark/index.html

Spark is a more heavyweight choice to scale computation for Hamilton graphs creating a Pandas Dataframe.

Notes on scaling:

- Multi-core on single machine ✅ (if you setup Spark locally to do so)
- Distributed computation on a Spark cluster ✅
- Scales to any size of data as permitted by Spark ✅

Function return object types supported:

- ⛔ Not generic. This does not work for every Hamilton graph.
- ✅ Currently we're targeting this at Pandas/Koalas types [dataframes, series].

Pandas?

- ✅ Koalas on Spark 3.2+ implements a good subset of the pandas API. Keep it simple and you should be good to go!

CAVEATS

- Serialization costs can outweigh the benefits of parallelism, so you should benchmark your code to see if it's worth it.

DISCLAIMER – this class is experimental, so signature changes are a possibility!

__init__(*spark_session*, *result_builder: ResultMixin*, *spine_column: str*)

Constructor

You only have the ability to return either a Pandas on Spark Dataframe or a Pandas Dataframe. To do that you either use the stock base.PandasDataFrameResult class, or you use h_spark.KoalasDataframeResult.

Parameters:

- **spark_session** – the spark session to use.
- **result_builder** – the function to build the result – currently on Pandas and Koalas are "supported".
- **spine_column** – the column we should use first as the spine and then subsequently join against.

build_result(***outputs: Dict[str, Any]*) → DataFrame | DataFrame | dict

Given a set of outputs, build the result.

Parameters:

outputs – the outputs from the execution of the graph.

Returns:

the result of the execution of the graph.

static check_input_type(*node_type: Type*, *input_value: Any*) → bool
Function to equate an input value, with expected node type.

We need this to equate pandas and koalas objects/types.

Parameters:

- **node_type** – the declared node type
- **input_value** – the actual input value

Returns:

whether this is okay, or not.

static check_node_type_equivalence(*node_type: Type*, *input_type: Type*) → bool
Function to help equate pandas with koalas types.

Parameters:

- **node_type** – the declared node type.
- **input_type** – the type of what we want to pass into it.

Returns:

whether this is okay, or not.

do_build_result(*outputs: Dict[str, Any]*) → Any
Implements the do_build_result method from the BaseDoBuildResult class. This is kept from the user as the public-facing API is build_result, allowing us to change the API/implementation of the internal set of hooks

do_check_edge_types_match(*type_from: type*, *type_to: type*) → bool
Method that checks whether two types are equivalent. This is used when the function graph is being created.

Parameters:

- **type_from** – The type of the node that is the source of the edge.

- **type_to** – The type of the node that is the destination of the edge.

Return bool:

Whether or not they are equivalent

do_node_execute(*run_id: str*, *node_: Node*, *kwargs: Dict[str, Any]*, *task_id: str | None = None*) → Any

Method that is called to implement node execution. This can replace the execution of a node with something all together, augment it, or delegate it.

Parameters:

- **run_id** – ID of the run, unique in scope of the driver.
- **node** – Node that is being executed
- **kwargs** – Keyword arguments that are being passed into the node
- **task_id** – ID of the task, defaults to None if not in a task setting

do_validate_input(*node_type: type*, *input_value: Any*) → bool

Method that an input value maches an expected type.

Parameters:

- **node_type** – The type of the node.
- **input_value** – The value that we want to validate.

Returns:

Whether or not the input value matches the expected type.

execute_node(*node: Node*, *kwargs: Dict[str, Any]*) → Any

Function that is called as we walk the graph to determine how to execute a hamilton function.

Parameters:

- **node** – the node from the graph.

· **kwargs** – the arguments that should be passed to it.

Returns:

returns a koalas column

input_types() → List[Type[Type]]

Gives the applicable types to this result builder. This is optional for backwards compatibility, but is recommended.

Returns:

A list of types that this can apply to.

output_type() → Type

Returns the output type of this result builder :return: the type that this creates

Lifecycle Adapters

Currently a few of the API extensions are still experimental. Note this doesn't mean they're not well-tested or thought out – rather that we're actively looking for feedback. More docs upcoming, but for now fish around the experimental package, and give the extensions a try!

The other extensions live within plugins. These are fully supported and will be backwards compatible across major versions.

Customization

The subsequent documents contain public-facing APIs for customizing Apache Hamilton's execution. Note that the public-facing APIs are still a work in progress – we will be improving the documentation. We plan for the APIs, however, to be stable looking forward.

lifecycle.ResultBuilder

class hamilton.lifecycle.api.ResultBuilder

Abstract class for building results. All result builders should inherit from this class and implement the build_result function. Note that applicable_input_type and output_type are optional, but recommended, for backwards compatibility. They let us type-check this. They will default to Any, which means that they'll connect to anything.

abstractmethod build_result(***outputs: Any*) → Any

Given a set of outputs, build the result.

Parameters:

outputs – the outputs from the execution of the graph.

Returns:

the result of the execution of the graph.

final do_build_result(*outputs: Dict[str, Any]*) → Any

Implements the do_build_result method from the BaseDoBuildResult class. This is kept from the user as the public-facing API is build_result, allowing us to change the API/implementation of the internal set of hooks

input_types() → List[Type[Type]]

Gives the applicable types to this result builder. This is optional for backwards compatibility, but is recommended.

Returns:

A list of types that this can apply to.

output_type() → Type

Returns the output type of this result builder :return: the type that this creates

lifecycle.LegacyResultMixin

class hamilton.lifecycle.api.LegacyResultMixin

Backwards compatible legacy result builder. This utilizes a static method as we used to do that, although often times they got confused. If you want a result builder, use ResultBuilder above instead.

static build_result(**outputs: Any*) → Any

Given a set of outputs, build the result.

Parameters:

outputs – the outputs from the execution of the graph.

Returns:

the result of the execution of the graph.

do_build_result(*outputs: Dict[str, Any]*) → Any

Implements the do_build_result method from the BaseDoBuildResult class. This is kept from the user as the public-facing API is build_result, allowing us to change the API/implementation of the internal set of hooks

input_types() → List[Type[Type]]

Gives the applicable types to this result builder. This is optional for backwards compatibility, but is recommended.

Returns:

A list of types that this can apply to.

output_type() → Type

Returns the output type of this result builder :return: the type that this creates

lifecycle.api.GraphAdapter

class hamilton.lifecycle.api.GraphAdapter

This is an implementation of HamiltonGraphAdapter, which has now been implemented with lifecycle methods/hooks.

static build_result(***outputs: Any*) → Any

Given a set of outputs, build the result.

Parameters:

outputs – the outputs from the execution of the graph.

Returns:

the result of the execution of the graph.

abstractmethod static check_input_type(*node_type: Type*, *input_value: Any*) → bool

Used to check whether the user inputs match what the execution strategy & functions can handle.

Static purely for legacy reasons.

Parameters:

- **node_type** – The type of the node.
- **input_value** – An actual value that we want to inspect matches our expectation.

Returns:

True if the input is valid, False otherwise.

abstractmethod static check_node_type_equivalence(*node_type: Type*, *input_type: Type*) → bool

Used to check whether two types are equivalent.

Static, purely for legacy reasons.

This is used when the function graph is being created and we're statically type checking the annotations for compatibility.

Parameters:

- **node_type** – The type of the node.
- **input_type** – The type of the input that would flow into the node.

Returns:

True if the types are equivalent, False otherwise.

do_build_result(*outputs: Dict[str, Any]*) → Any

Implements the do_build_result method from the BaseDoBuildResult class. This is kept from the user as the public-facing API is build_result, allowing us to change the API/implementation of the internal set of hooks

final do_check_edge_types_match(*type_from: type*, *type_to: type*) → bool

Method that checks whether two types are equivalent. This is used when the function graph is being created.

Parameters:

- **type_from** – The type of the node that is the source of the edge.
- **type_to** – The type of the node that is the destination of the edge.

Return bool:

Whether or not they are equivalent

final do_node_execute(*run_id: str*, *node_: Node*, *kwargs: Dict[str, Any]*, *task_id: str | None = None*) → Any

Method that is called to implement node execution. This can replace the execution of a node with something all together, augment it, or delegate it.

Parameters:

- **run_id** – ID of the run, unique in scope of the driver.
- **node** – Node that is being executed
- **kwargs** – Keyword arguments that are being passed into the node

- **task_id** – ID of the task, defaults to None if not in a task setting

final do_validate_input(*node_type: type*, *input_value: Any*) → bool

Method that an input value maches an expected type.

Parameters:

- **node_type** – The type of the node.
- **input_value** – The value that we want to validate.

Returns:

Whether or not the input value matches the expected type.

abstractmethod execute_node(*node: Node*, *kwargs: Dict[str, Any]*) → Any

Given a node that represents a hamilton function, execute it. Note, in some adapters this might just return some type of "future".

Parameters:

- **node** – the Hamilton Node
- **kwargs** – the kwargs required to exercise the node function.

Returns:

the result of exercising the node.

input_types() → List[Type[Type]]

Gives the applicable types to this result builder. This is optional for backwards compatibility, but is recommended.

Returns:

A list of types that this can apply to.

output_type() → Type

Returns the output type of this result builder :return: the type that this creates

lifecycle.NodeExecutionHook

class hamilton.lifecycle.api.NodeExecutionHook

Implement this to hook into the node execution lifecycle. You can call anything before and after the driver

final post_node_execute(*, *run_id: str*, *node_: Node*, *kwargs: Dict[str, Any]*, *success: bool*, *error: Exception | None*, *result: Any | None*, *task_id: str | None = None*)

Wraps the after_execution method, providing a bridge to an external-facing API. Do not override this!

final pre_node_execute(*, *run_id: str*, *node_: Node*, *kwargs: Dict[str, Any]*, *task_id: str | None = None*)

Wraps the before_execution method, providing a bridge to an external-facing API. Do not override this!

abstractmethod run_after_node_execution(*, *node_name: str*, *node_tags: Dict[str, Any]*, *node_kwargs: Dict[str, Any]*, *node_return_type: type*, *result: Any*, *error: Exception | None*, *success: bool*, *task_id: str | None*, *run_id: str*, ***future_kwargs: Any*)

Hook that is executed post node execution.

Parameters:

- **node_name** – Name of the node in question
- **node_tags** – Tags of the node
- **node_kwargs** – Keyword arguments passed to the node
- **node_return_type** – Return type of the node
- **result** – Output of the node, None if an error occurred
- **error** – Error that occurred, None if no error occurred
- **success** – Whether the node executed successfully
- **task_id** – The ID of the task, none if not in a task-based environment
- **run_id** – Run ID (unique in process scope) of the current run. Use this to track state.
- **future_kwargs** – Additional keyword arguments – this is kept for backwards compatibility

abstractmethod run_before_node_execution(*, *node_name: str*, *node_tags: Dict[str, Any]*, *node_kwargs: Dict[str, Any]*, *node_return_type: type*, *task_id: str | None*, *run_id: str*, *node_input_types: Dict[str, Any]*, ***future_kwargs: Any*)

Hook that is executed prior to node execution.

Parameters:

- **node_name** – Name of the node.
- **node_tags** – Tags of the node
- **node_kwargs** – Keyword arguments to pass to the node
- **node_return_type** – Return type of the node
- **task_id** – The ID of the task, none if not in a task-based environment
- **run_id** – Run ID (unique in process scope) of the current run. Use this to track state.
- **node_input_types** – the input types to the node and what it is expecting
- **future_kwargs** – Additional keyword arguments – this is kept for backwards compatibility

lifecycle.api.GraphExecutionHook

class hamilton.lifecycle.api.GraphExecutionHook

Implement this to execute code before and after graph execution. This is useful for logging, etc...

final post_graph_execute(*, *run_id: str*, *graph: FunctionGraph*, *success: bool*, *error: Exception | None*, *results: Dict[str, Any] | None*)

Just delegates to the interface method, passing in the right data.

final pre_graph_execute(*, *run_id: str*, *graph: FunctionGraph*, *final_vars: List[str]*, *inputs: Dict[str, Any]*, *overrides: Dict[str, Any]*)

Implementation of the pre_graph_execute hook. This just converts the inputs to the format the user-facing hook is expecting – performing a walk of the DAG to pass in the set of nodes to execute. Delegates to the interface method.

abstractmethod run_after_graph_execution(*, *graph: HamiltonGraph*, *success: bool*, *error: Exception | None*, *results: Dict[str, Any] | None*, *run_id: str*, ***future_kwargs: Any*)

This is run after graph execution. This allows you to do anything you want after the graph executes, knowing the results of the execution/any errors.

Parameters:

- **graph** – Graph that is being executed
- **results** – Results of the graph execution
- **error** – Error that occurred, None if no error occurred
- **success** – Whether the graph executed successfully
- **run_id** – Run ID (unique in process scope) of the current run. Use this to track state.
- **future_kwargs** – Additional keyword arguments – this is kept for backwards compatibility

abstractmethod run_before_graph_execution(*, *graph: HamiltonGraph*, *final_vars: List[str]*, *inputs: Dict[str, Any]*, *overrides: Dict[str, Any]*, *execution_path: Collection[str]*, *run_id: str*, ***future_kwargs: Any*)

This is run prior to graph execution. This allows you to do anything you want before the graph executes, knowing the basic information that was passed in.

Parameters:

- **graph** – Graph that is being executed
- **final_vars** – Output variables of the graph
- **inputs** – Input variables passed to the graph
- **overrides** – Overrides passed to the graph
- **execution_path** – Collection of nodes that will be executed – these are just the nodes (not input nodes) that will be run during the course of execution.
- **run_id** – Run ID (unique in process scope) of the current run. Use this to track state.
- **future_kwargs** – Additional keyword arguments – this is kept for backwards compatibility

lifecycle.api.EdgeConnectionHook

class hamilton.lifecycle.api.EdgeConnectionHook

Implement this to customize edges that are allowed in the graph. You can do customizations around typing here.

abstractmethod check_edge_types_match(*type_from: type*, *type_to: type*, ***kwargs: Any*) → bool

This is run to check if edge types match. Note that this is an OR functionality – this is run after we do some default checks, so this can only be permissive. Reach out if you want to be more restrictive than the default checks.

Parameters:

- **type_from** – The type of the node that is the source of the edge.
- **type_to** – The type of the node that is the destination of the edge.
- **kwargs** – This is kept for future backwards compatibility.

Returns:

Whether or not the two node types form a valid edge.

final do_check_edge_types_match(***, *type_from: type*, *type_to: type*) → bool

Wraps the check_edge_types_match method, providing a bridge to an external-facing API. Do not override this!

final do_validate_input(***, *node_type: type*, *input_value: Any*) → bool

Wraps the validate_input method, providing a bridge to an external-facing API. Do not override this!

abstractmethod validate_input(*node_type: type*, *input_value: Any*, ***kwargs: Any*) → bool

This is run to check if the input is valid for the node type. Note that this is an OR functionality – this is run after we do some default checks, so this can only be permissive. Reach out if you want to be more restrictive than the default checks.

Parameters:

- **node_type** – Type of the node that is accepting the input.

- **input_value** – Value of the input
- **kwargs** – Keyword arguments – this is kept for future backwards compatibility.

Returns:

Whether the input is valid for the node type.

lifecycle.api.NodeExecutionMethod

class hamilton.lifecycle.api.NodeExecutionMethod

API for executing a node. This takes in tags, callable, node name, and kwargs, and is responsible for executing the node and returning the result. Note this is not (currently) able to be layered together, although we may add that soon.

final do_node_execute(*, *run_id: str*, *node_: Node*, *kwargs: Dict[str, Any]*, *task_id: str | None = None*) → Any

Method that is called to implement node execution. This can replace the execution of a node with something all together, augment it, or delegate it.

Parameters:

- **run_id** – ID of the run, unique in scope of the driver.
- **node** – Node that is being executed
- **kwargs** – Keyword arguments that are being passed into the node
- **task_id** – ID of the task, defaults to None if not in a task setting

abstractmethod run_to_execute_node(*, *node_name: str*, *node_tags: Dict[str, Any]*, *node_callable: Any*, *node_kwargs: Dict[str, Any]*, *task_id: str | None*, *is_expand: bool*, *is_collect: bool*, ***future_kwargs: Any*) → Any

This method is responsible for executing the node and returning the result.

Parameters:

- **node_name** – Name of the node.
- **node_tags** – Tags of the node.

- **node_callable** – Callable of the node.
- **node_kwargs** – Keyword arguments to pass to the node.
- **task_id** – The ID of the task, none if not in a task-based environment
- **is_expand** – Whether the node is parallelizable.
- **is_collect** – Whether the node is a collect node.
- **future_kwargs** – Additional keyword arguments – this is kept for backwards compatibility

Returns:

The result of the node execution – up to you to return this.

lifecycle.api.StaticValidator

class hamilton.lifecycle.api.StaticValidator

Performs static validation of the DAG. Note that this has the option to perform default validation for each method – this means that if you don't implement one of these it is OK.

```
class MyTagValidator(api.StaticValidator):
    '''Validates tags on a node'''

    def run_to_validate_node(
            self, *, node: HamiltonNode, **future_kwargs
    ) -> tuple[bool, Optional[str]]:
        if node.tags.get("node_type", "") == "output":
            table_name = node.tags.get("table_name")
            if not table_name:  # None or empty
                error_msg = (f"Node {node.tags['module']}.
{node.name} "

"is an output node, but does not have a table_name tag.")
                return False, error_msg
        return True, None
```

run_to_validate_graph(*graph: HamiltonGraph*, ***future_kwargs*) → Tuple[bool, str | None]

Override this to build custom DAG validations! Default to just returning that the graph is valid, so you don't have to implement it if you want to just implement a single method. Runs post graph construction to validate a graph. You have access to a bunch of metadata about the graph, stored in the graph argument.

Parameters:

- **graph** – Graph to validate.
- **future_kwargs** – Additional keyword arguments – this is kept for backwards compatibility

Returns:

A tuple of whether the graph is valid and an error message in the case of failure. Return [True, None] for a valid graph. Otherwise, return a detailed error message – this should have all context/debugging information.

run_to_validate_node(*, *node: HamiltonNode*, ***future_kwargs*) → Tuple[bool, str | None]

Override this to build custom node validations! Defaults to just returning that a node is valid so you don't have to implement it if you want to just implement a single method. Runs post node construction to validate a node. You have access to a bunch of metadata about the node, stored in the hamilton_node argument

Parameters:

- **node** – Node to validate
- **future_kwargs** – Additional keyword arguments – this is kept for backwards compatibility

Returns:

A tuple of whether the node is valid and an error message in the case of failure. Return [True, None] for a valid node.Otherwise, return a detailed error message – this should have all context/debugging information, but does not need to mention the node name (it will be aggregated with others).

final validate_graph(*, *graph: FunctionGraph*, *modules: List[ModuleType]*, *config: Dict[str, Any]*) → Tuple[bool, Exception | None]

Validates the graph. This will raise an InvalidNodeException

Parameters:

- **graph** – Graph that has been constructed.
- **modules** – Modules passed into the graph
- **config** – Config passed into the graph

Returns:

A (is_valid, error_message) tuple

final validate_node(*, *created_node: Node*) → Tuple[bool, Exception | None]

Validates a node. This will raise an InvalidNodeException if the node is invalid.

Parameters:

created_node – Node that was created.

Raises:

InvalidNodeException – If the node is invalid.

lifecycle.api.GraphConstructionHook

class hamilton.lifecycle.api.GraphConstructionHook

Hook that is run after graph construction. This allows you to register/capture info on the graph. Note that, in the case of materialization, this may be called multiple times (once when we create the graph, once when we materialize). Currently information into that is not exposed to the user, but we will be adding that in future iterations.

post_graph_construct(*, *graph: FunctionGraph*, *modules: List[ModuleType]*, *config: Dict[str, Any]*)

Hooks that is called after the graph is constructed.

Parameters:

- **graph** – Graph that has been constructed.
- **modules** – Modules passed into the graph
- **config** – Config passed into the graph

abstractmethod run_after_graph_construction(*, *graph: HamiltonGraph*, *config: Dict[str, Any]*, ***future_kwargs: Any*)

Hook that is run post graph construction. This allows you to register/capture info on the graph. A common pattern is to store something in your object's state here so that you can use it later (E.G. compute a hash on the graph)

Parameters:

- **graph** – Graph that was constructed
- **config** – Configuration used to construct the graph
- **future_kwargs** – Reserved for backwards compatibility.

lifecycle.api.TaskSubmissionHook

class hamilton.lifecycle.api.TaskSubmissionHook

Implement this to hook into the task submission process. Tasks are submitted to an executor, which then controls how and where the nodes associated with the task are run.

pre_task_submission(*, *run_id: str*, *task_id: str*, *nodes: List[Node]*, *inputs: Dict[str, Any]*, *overrides: Dict[str, Any]*, *spawning_task_id: str | None*, *purpose: None*)

Hook that is called immediately prior to task submission to an executor as a task future. Note that this is only useful in dynamic execution, although we reserve the right to add this back into the standard hamilton execution pattern.

Parameters:

- **run_id** – ID of the run, unique in scope of the driver.
- **task_id** – ID of the task.
- **nodes** – Nodes that are being executed.
- **inputs** – Inputs to the task.
- **overrides** – Overrides to task execution.
- **spawning_task_id** – ID of the task that spawned this task.
- **purpose** – Purpose of the current task group.

abstractmethod run_before_task_submission(*, *run_id: str*, *task_id: str*, *nodes: List[Node]*, *inputs: Dict[str, Any]*, *overrides: Dict[str, Any]*, *spawning_task_id: str | None*, *purpose: None*, ***future_kwargs*)

Runs prior to a task being submitted to an executor. By definition this is run *outside* of the task executor, on the process that executed the driver.

Parameters:

- **run_id** – ID of the run this is under.
- **task_id** – ID of the task we're launching.
- **nodes** – Nodes that are part of this task
- **inputs** – Inputs to the task
- **overrides** – Overrides passed to the task
- **spawning_task_id** – ID of the task that spawned this task
- **purpose** – Purpose of the current task group
- **future_kwargs** – Reserved for backwards compatibility.

lifecycle.api.TaskReturnHook

class hamilton.lifecycle.api.TaskReturnHook

Implement this to hook into the task return process. Tasks are submitted to an executor, which executes the task and returns the results (or raises an error).

post_task_return(*, *run_id: str*, *task_id: str*, *nodes: List[Node]*, *result: Any*, *success: bool*, *error: Exception | None*, *spawning_task_id: str | None*, *purpose: None*)

Hook called immediately after a task returns from an executor. Note that this is only useful in dynamic execution, although we reserve the right to add this back into the standard hamilton execution pattern.

Parameters:

- **run_id** – ID of the run, unique in scope of the driver.
- **task_id** – ID of the task

- **result** – Return value of the task.
- **success** – Whether or not the task executed successfully
- **error** – The error that was raised, if any
- **spawning_task_id** – ID of the task that spawned this task
- **purpose** – Purpose of the current task group

abstractmethod run_after_task_return(*, *run_id: str*, *task_id: str*, *nodes: List[Node]*, *result: Any*, *success: bool*, *error: Exception | None*, *spawning_task_id: str | None*, *purpose: None*, ***future_kwargs*)

Runs after a task has been returned from a executor. By definition this is run *outside* of the task executor, on the process that executed the driver.

Parameters:

- **run_id** – ID of the run this is under.
- **task_id** – ID of the task that was just executed.
- **nodes** – Nodes that were part of this task
- **result** – Result of the task
- **success** – Whether the task was successful
- **error** – The error the task threw, if any
- **spawning_task_id** – ID of the task that spawned this task
- **purpose** – Purpose of the current task group
- **future_kwargs** – Reserved for backwards compatibility.

lifecycle.api.TaskExecutionHook

class hamilton.lifecycle.api.TaskExecutionHook

Implement this to hook into the task execution process. Tasks consist of a group of one or more nodes that are run on a task executor.

post_task_execute(*, *run_id: str*, *task_id: str*, *nodes: List[Node]*, *results: Dict[str, Any] | None*, *success: bool*, *error: Exception*, *spawning_task_id: str | None*, *purpose: None*)

Hook called immediately after task execution. Note that this is only useful in dynamic execution, although we reserve the right to add this back into the standard hamilton execution pattern.

Parameters:

- **run_id** – ID of the run, unique in scope of the driver.
- **task_id** – ID of the task
- **nodes** – Nodes that were executed
- **results** – Results of the task
- **success** – Whether or not the task executed successfully
- **error** – The error that was raised, if any
- **spawning_task_id** – ID of the task that spawned this task
- **purpose** – Purpose of the current task group

pre_task_execute(*, *run_id: str*, *task_id: str*, *nodes: List[Node]*, *inputs: Dict[str, Any]*, *overrides: Dict[str, Any]*, *spawning_task_id: str | None*, *purpose: None*)

Hook that is called immediately prior to task execution. Note that this is only useful in dynamic execution, although we reserve the right to add this back into the standard hamilton execution pattern.

Parameters:

- **run_id** – ID of the run, unique in scope of the driver.
- **task_id** – ID of the task, unique in scope of the driver.
- **nodes** – Nodes that are being executed
- **inputs** – Inputs to the task
- **overrides** – Overrides to task execution
- **spawning_task_id** – ID of the task that spawned this task

- **purpose** – Purpose of the current task group

abstractmethod run_after_task_execution(*, *task_id: str*, *run_id: str*, *nodes: List[HamiltonNode]*, *results: Dict[str, Any] | None*, *success: bool*, *error: Exception*, *spawning_task_id: str | None*, *purpose: None*, ***future_kwargs*)

Runs after all of the nodes associated with a task have been executed. By definition this is run *inside* of the executor and therefore may be run on separate or distributed processes.

Parameters:

- **task_id** – ID of the task that was just executed
- **run_id** – ID of the run this was under.
- **nodes** – Nodes that were part of this task
- **results** – Results of the task, per-node
- **success** – Whether the task was successful
- **error** – The error the task threw, if any
- **future_kwargs** – Reserved for backwards compatibility.
- **spawning_task_id** – ID of the task that spawned this task
- **purpose** – Purpose of the current task group

abstractmethod run_before_task_execution(*, *task_id: str*, *run_id: str*, *nodes: List[HamiltonNode]*, *inputs: Dict[str, Any]*, *overrides: Dict[str, Any]*, *spawning_task_id: str | None*, *purpose: None*, ***future_kwargs*)

Runs prior to any of the nodes associated with a task. By definition this is run *inside* of the executor and therefore may be run on separate or distributed processes.

Parameters:

- **task_id** – ID of the task we're launching.
- **run_id** – ID of the run this is under.
- **nodes** – Nodes that are part of this task
- **inputs** – Inputs to the task
- **overrides** – Overrides passed to the task

- **future_kwargs** – Reserved for backwards compatibility.
- **spawning_task_id** – ID of the task that spawned this task
- **purpose** – Purpose of the current task group

lifecycle.api.TaskGroupingHook

class hamilton.lifecycle.api.TaskGroupingHook

Implement this to run something after task grouping or task expansion. This will allow you to capture information about the tasks during *Parallelize/Collect* blocks in dynamic DAG execution.

final post_task_expand(*, *run_id: str*, *task_id: str*, *parameters: Dict[str, Any]*)

Hook that is called immediately after a task is expanded into parallelizable tasks. Note that this is only useful in dynamic execution.

Parameters:

- **run_id** – ID of the run, unique in scope of the driver.
- **task_id** – ID of the task.
- **parameters** – Parameters that are being passed to each of the expanded tasks.

final post_task_group(*, *run_id: str*, *task_ids: List[str]*)

Hook that is called immediately after a task group is created. Note that this is only useful in dynamic execution, although we reserve the right to add this back into the standard hamilton execution pattern.

Parameters:

- **run_id** – ID of the run, unique in scope of the driver.
- **task_ids** – IDs of tasks that are in the group.

abstractmethod run_after_task_expansion(*, *run_id: str*, *task_id: str*, *parameters: Dict[str, Any]*, ***future_kwargs*)

Runs after task expansion in Parallelize/Collect blocks. This allows you to capture information about the task that was expanded.

Parameters:

- **run_id** – ID of the run, unique in scope of the driver.
- **task_id** – ID of the task that was expanded.
- **parameters** – Parameters that were passed to the task.
- **future_kwargs** – Additional keyword arguments – this is kept for backwards compatibility.

abstractmethod run_after_task_grouping(*, *run_id: str*, *task_ids: List[str]*, ***future_kwargs*)

Runs after task grouping. This allows you to capture information about which tasks were created for a given run.

Parameters:

- **run_id** – ID of the run, unique in scope of the driver.
- **task_ids** – List of tasks that were grouped together.
- **future_kwargs** – Additional keyword arguments – this is kept for backwards compatibility.

Available Adapters

In addition to the base classes for lifecycle adapters, we have a few adapters implemented and available for use. Note that some of these are plugins, meaning they require installing additional (external) libraries.

Recall to add lifecycle adapters, you just need to call the `with_adapters` method of the driver:

```
dr = (
    driver
    .Builder()
    .with_modules(...)
    .with_adapters(
        Adapter1(...),
        Adapter2(...),
        *more_adapters)
    ...build()
)
```

lifecycle.PDBDebugger

class hamilton.lifecycle.default.PDBDebugger(*node_filter: Callable[[str, Dict[str, Any]], bool] | List[str] | str | None*, *before: bool = False*, *during: bool = False*, *after: bool = False*)

Class to inject a PDB debugger into a node execution. This is still somewhat experimental as it is a debugging utility. We reserve the right to change the API and the implementation of this class in the future.

__init__(*node_filter: Callable[[str, Dict[str, Any]], bool] | List[str] | str | None*, *before: bool = False*, *during: bool = False*, *after: bool = False*)

Creates a PDB debugger. This has three possible modes:

1. Before – places you in a function with (a) node information, and (b) inputs
2. **During – runs the node with pdb.run. Note this may not always work or give what you expect as**

   node functions are often wrapped in multiple levels of input modifications/whatnot. That said, it should give you something. Also note that this is not (currently) compatible with graph adapters.
3. After – places you in a function with (a) node information, (b) inputs, and (c) results

Parameters:

- **node_filter** – A function that takes a node name and a node tags dict and returns a boolean. If the boolean is True, the node will be printed out.
- **before** – Whether to place you in a PDB debugger before a node executes
- **during** – Whether to place you in a PDB debugger during a node's execution
- **after** – Whether to place you in a PDB debugger after a node executes

do_node_execute(*, *run_id: str*, *node_: Node*, *kwargs: Dict[str, Any]*, *task_id: str | None = None*) → Any

Method that is called to implement node execution. This can replace the execution of a node with something all together, augment it, or delegate it.

Parameters:

- **run_id** – ID of the run, unique in scope of the driver.

- **node** – Node that is being executed
- **kwargs** – Keyword arguments that are being passed into the node
- **task_id** – ID of the task, defaults to None if not in a task setting

post_node_execute(*, *run_id: str*, *node_: Node*, *kwargs: Dict[str, Any]*, *success: bool*, *error: Exception | None*, *result: Any | None*, *task_id: str | None = None*)

Wraps the after_execution method, providing a bridge to an external-facing API. Do not override this!

pre_node_execute(*, *run_id: str*, *node_: Node*, *kwargs: Dict[str, Any]*, *task_id: str | None = None*)

Wraps the before_execution method, providing a bridge to an external-facing API. Do not override this!

run_after_node_execution(*, *node_name: str*, *node_tags: Dict[str, Any]*, *node_kwargs: Dict[str, Any]*, *node_return_type: type*, *result: Any*, *error: Exception | None*, *success: bool*, *task_id: str | None*, ***future_kwargs: Any*)

Executes after a node, whether or not it was successful. Does nothing, just runs pdb.set_trace().

Parameters:

- **node_name** – Name of the node
- **node_tags** – Tags of the node
- **node_kwargs** – Keyword arguments passed to the node
- **node_return_type** – Return type of the node
- **result** – Result of the node, None if there was an error
- **error** – Error of the node, None if there was no error
- **success** – Whether the node ran successful or not
- **task_id** – Task ID of the node, if any
- **future_kwargs** – Additional keyword arguments that may be passed to the hook yet are ignored for now

run_before_node_execution(*, *node_name: str*, *node_tags: Dict[str, Any]*, *node_kwargs: Dict[str, Any]*, *node_return_type: type*, *task_id: str | None*, ***future_kwargs: Any*)

Executes before a node executes. Does nothing, just runs pdb.set_trace()

Parameters:

- **node_name** – Name of the node
- **node_tags** – Tags of the node
- **node_kwargs** – Keyword arguments passed to the node
- **node_return_type** – Return type of the node
- **task_id** – ID of the task that the node is in, if any
- **future_kwargs** – Additional keyword arguments that may be passed to the hook yet are ignored for now

Returns:

Result of the node

run_to_execute_node(*, *node_name: str*, *node_tags: Dict[str, Any]*, *node_callable: Any*, *node_kwargs: Dict[str, Any]*, *task_id: str | None*, ***future_kwargs: Any*) → Any

Executes the node with a PDB debugger. This modifies the global PDBDebugger.CONTEXT variable to contain information about the node, so you can access it while debugging.

Parameters:

- **node_name** – Name of the node
- **node_tags** – Tags of the node
- **node_callable** – Callable function of the node
- **node_kwargs** – Keyword arguments passed to the node
- **task_id** – ID of the task that the node is in, if any
- **future_kwargs** – Additional keyword arguments that may be passed to the hook yet are ignored for now

Returns:

Result of the node

lifecycle.PrintLn

Use this hook to print out data before/after a node's execution for debugging

class hamilton.lifecycle.default.PrintLn(*verbosity: int = 1, print_fn: ~typing.Callable[[str], None] = <built-in function print>, node_filter: ~typing.Callable[[str, ~typing.Dict[str, ~typing.Any]], bool] | ~typing.List[str] | str | None = None*)

Basic hook to print out information before/after node execution.

__init__(*verbosity: int = 1, print_fn: ~typing.Callable[[str], None] = <built-in function print>, node_filter: ~typing.Callable[[str, ~typing.Dict[str, ~typing.Any]], bool] | ~typing.List[str] | str | None = None*)

Prints out information before/after node execution.

Parameters:

- **verbosity** – The verbosity level to print out at. *verbosity=1* Print out just the node name and time it took to execute *verbosity=2*. Print out inputs of the node + results on execute
- **print_fn** – A function that takes a string and prints it out – defaults to print. Pass in a logger function, etc... if you so choose.
- **node_filter** – A function that takes a node name and a node tags dict and returns a boolean. If the boolean is True, the node will be printed out. If False, it will not be printed out.

post_node_execute(*, *run_id: str*, *node_: Node*, *kwargs: Dict[str, Any]*, *success: bool*, *error: Exception | None*, *result: Any | None*, *task_id: str | None = None*)

Wraps the after_execution method, providing a bridge to an external-facing API. Do not override this!

pre_node_execute(*, *run_id: str*, *node_: Node*, *kwargs: Dict[str, Any]*, *task_id: str | None = None*)

Wraps the before_execution method, providing a bridge to an external-facing API. Do not override this!

run_after_node_execution(*, *node_name: str*, *node_tags: Dict[str, Any]*, *node_kwargs: Dict[str, Any]*, *result: Any*, *error: Exception | None*, *success: bool*, *task_id: str | None*, ***future_kwargs: Any*)

Runs after a node executes. Prints out the node name and time it took, the output if verbosity is 1.

Parameters:

- **node_name** – Name of the node
- **node_tags** – Tags of the node
- **node_kwargs** – Keyword arguments passed to the node
- **result** – Result of the node
- **error** – Error of the node
- **success** – Whether the node was successful or not
- **task_id** – ID of the task that the node is in, if any
- **future_kwargs** – Additional keyword arguments that may be passed to the hook yet are ignored for now

run_before_node_execution(*, *node_name: str*, *node_tags: Dict[str, Any]*, *node_kwargs: Dict[str, Any]*, *task_id: str | None*, ***future_kwargs: Any*)

Runs before a node executes. Prints out the node name and inputs if verbosity is 2.

Parameters:

- **node_name** – Name of the node
- **node_tags** – Tags of the node
- **node_kwargs** – Keyword arguments of the node
- **task_id** – ID of the task that the node is in, if any
- **future_kwargs** – Additional keyword arguments that may be passed to the hook yet are ignored for now

plugins.h_tqdm.ProgressBar

Provides a progress bar for Apache Hamilton execution. Must have *tqdm* installed to use it:

pip install sf-hamilton[tqdm] (use quotes if using zsh)

class hamilton.plugins.h_tqdm.ProgressBar(*desc: str = 'Graph execution'*, *max_node_name_width: int = 50*, ***kwargs*)

An adapter that uses tqdm to show progress bars for the graph execution.

Note: you need to have tqdm installed for this to work. If you don't have it installed, you can install it with *pip install tqdm* (or *pip install sf-hamilton[tqdm]* – use quotes if you're using zsh).

```
from hamilton.plugins import h_tqdm

dr = (
    driver.Builder()
    .with_config({})
    .with_modules(some_modules)
    .with_adapters(h_tqdm.ProgressBar(desc="DAG-NAME"))
    .build()
)
# and then when you call .execute() or .materialize() you'll get
a progress bar!
```

__init__(*desc: str = 'Graph execution'*, *max_node_name_width: int = 50*, ***kwargs*)

Create a new Progress Bar adapter.

Parameters:

- **desc** – The description to show in the progress bar. E.g. DAG Name is a good choice.
- **kwargs** – Additional kwargs to pass to TQDM. See TQDM docs for more info.
- **node_name_target_width** – the target width for the node name so that the progress bar is consistent. If this is None, it will take the longest, until it hits max_node_name_width.

post_graph_execute(***, *run_id: str*, *graph: FunctionGraph*, *success: bool*, *error: Exception | None*, *results: Dict[str, Any] | None*)

Just delegates to the interface method, passing in the right data.

post_node_execute(*, *run_id: str*, *node_: Node*, *kwargs: Dict[str, Any]*, *success: bool*, *error: Exception | None*, *result: Any | None*, *task_id: str | None = None*)

Wraps the after_execution method, providing a bridge to an external-facing API. Do not override this!

pre_graph_execute(*, *run_id: str*, *graph: FunctionGraph*, *final_vars: List[str]*, *inputs: Dict[str, Any]*, *overrides: Dict[str, Any]*)

Implementation of the pre_graph_execute hook. This just converts the inputs to the format the user-facing hook is expecting – performing a walk of the DAG to pass in the set of nodes to execute. Delegates to the interface method.

pre_node_execute(*, *run_id: str*, *node_: Node*, *kwargs: Dict[str, Any]*, *task_id: str | None = None*)

Wraps the before_execution method, providing a bridge to an external-facing API. Do not override this!

run_after_graph_execution(***future_kwargs*)

This is run after graph execution. This allows you to do anything you want after the graph executes, knowing the results of the execution/any errors.

Parameters:

- **graph** – Graph that is being executed
- **results** – Results of the graph execution
- **error** – Error that occurred, None if no error occurred
- **success** – Whether the graph executed successfully
- **run_id** – Run ID (unique in process scope) of the current run. Use this to track state.
- **future_kwargs** – Additional keyword arguments – this is kept for backwards compatibility

run_after_node_execution(***future_kwargs*)

Hook that is executed post node execution.

Parameters:

- **node_name** – Name of the node in question
- **node_tags** – Tags of the node
- **node_kwargs** – Keyword arguments passed to the node

- **node_return_type** – Return type of the node
- **result** – Output of the node, None if an error occurred
- **error** – Error that occurred, None if no error occurred
- **success** – Whether the node executed successfully
- **task_id** – The ID of the task, none if not in a task-based environment
- **run_id** – Run ID (unique in process scope) of the current run. Use this to track state.
- **future_kwargs** – Additional keyword arguments – this is kept for backwards compatibility

run_before_graph_execution(*, *graph: HamiltonGraph*, *final_vars: List[str]*, *inputs: Dict[str, Any]*, *overrides: Dict[str, Any]*, *execution_path: Collection[str]*, ***future_kwargs: Any*)

This is run prior to graph execution. This allows you to do anything you want before the graph executes, knowing the basic information that was passed in.

Parameters:

- **graph** – Graph that is being executed
- **final_vars** – Output variables of the graph
- **inputs** – Input variables passed to the graph
- **overrides** – Overrides passed to the graph
- **execution_path** – Collection of nodes that will be executed – these are just the nodes (not input nodes) that will be run during the course of execution.
- **run_id** – Run ID (unique in process scope) of the current run. Use this to track state.
- **future_kwargs** – Additional keyword arguments – this is kept for backwards compatibility

run_before_node_execution(*, *node_name: str*, *node_tags: Dict[str, Any]*, *node_kwargs: Dict[str, Any]*, *node_return_type: type*, *task_id: str | None*, ***future_kwargs: Any*)

Hook that is executed prior to node execution.

Parameters:

- **node_name** – Name of the node.
- **node_tags** – Tags of the node
- **node_kwargs** – Keyword arguments to pass to the node
- **node_return_type** – Return type of the node
- **task_id** – The ID of the task, none if not in a task-based environment
- **run_id** – Run ID (unique in process scope) of the current run. Use this to track state.
- **node_input_types** – the input types to the node and what it is expecting
- **future_kwargs** – Additional keyword arguments – this is kept for backwards compatibility

plugins.h_rich.RichProgressBar

Provides a progress bar for Apache Hamilton execution. Must have *rich* installed to use it:

pip install sf-hamilton[rich] (use quotes if using zsh)

class hamilton.plugins.h_rich.RichProgressBar(*run_desc: str = ''*, *collect_desc: str = ''*, *columns: list[str | ProgressColumn] | None = None*, ***kwargs*)

An adapter that uses rich to show simple progress bars for the graph execution.

Note: you need to have rich installed for this to work. If you don't have it installed, you can install it with *pip install rich* (or *pip install sf-hamilton[rich]* – use quotes if you're using zsh).

```
from hamilton import driver
from hamilton.plugins import h_rich

dr = (
    driver.Builder()
    .with_config({})
    .with_modules(some_modules)
    .with_adapters(h_rich.RichProgressBar())
```

```
    .build()
)
```

and then when you call .execute() or .materialize() you'll get a progress bar!

Additionally, this progress bar will also work with task-based execution, showing the progress of overall execution as well as the tasks within a parallelized group.

```
from hamilton import driver
from hamilton.execution import executors
from hamilton.plugins import h_rich

dr = (
    driver.Builder()
    .with_modules(__main__)
    .enable_dynamic_execution(allow_experimental_mode=True)
    .with_adapters(RichProgressBar())
    .with_local_executor(executors.SynchronousLocalTaskExecutor())
    .with_remote_executor(executors.SynchronousLocalTaskExecutor())
    .build()
)
```

__init__(*run_desc: str = ''*, *collect_desc: str = ''*, *columns: list[str | ProgressColumn] | None = None*, ***kwargs*) → None

Create a new Rich Progress Bar adapter.

Parameters:

- **run_desc** – The description to show for the running phase.
- **collect_desc** – The description to show for the collecting phase (if applicable).
- **columns** – Column configuration for the progress bar. See rich docs for more info.
- **kwargs** – Additional kwargs to pass to rich.progress.Progress. See rich docs for more info.

post_graph_execute(***, *run_id: str*, *graph: FunctionGraph*, *success: bool*, *error: Exception | None*, *results: Dict[str, Any] | None*)

Just delegates to the interface method, passing in the right data.

post_node_execute(***, *run_id: str*, *node_: Node*, *kwargs: Dict[str, Any]*, *success: bool*, *error: Exception | None*, *result: Any | None*, *task_id: str | None = None*)

Wraps the after_execution method, providing a bridge to an external-facing API. Do not override this!

post_task_execute(*, *run_id: str*, *task_id: str*, *nodes: List[Node]*, *results: Dict[str, Any] | None*, *success: bool*, *error: Exception*, *spawning_task_id: str | None*, *purpose: None*)

Hook called immediately after task execution. Note that this is only useful in dynamic execution, although we reserve the right to add this back into the standard hamilton execution pattern.

Parameters:

- **run_id** – ID of the run, unique in scope of the driver.
- **task_id** – ID of the task
- **nodes** – Nodes that were executed
- **results** – Results of the task
- **success** – Whether or not the task executed successfully
- **error** – The error that was raised, if any
- **spawning_task_id** – ID of the task that spawned this task
- **purpose** – Purpose of the current task group

post_task_expand(*, *run_id: str*, *task_id: str*, *parameters: Dict[str, Any]*)

Hook that is called immediately after a task is expanded into parallelizable tasks. Note that this is only useful in dynamic execution.

Parameters:

- **run_id** – ID of the run, unique in scope of the driver.
- **task_id** – ID of the task.
- **parameters** – Parameters that are being passed to each of the expanded tasks.

post_task_group(*, *run_id: str*, *task_ids: List[str]*)

Hook that is called immediately after a task group is created. Note that this is only useful in dynamic execution, although we reserve the right to add this back into the standard hamilton execution pattern.

Parameters:

- **run_id** – ID of the run, unique in scope of the driver.
- **task_ids** – IDs of tasks that are in the group.

pre_graph_execute(*, *run_id: str*, *graph: FunctionGraph*, *final_vars: List[str]*, *inputs: Dict[str, Any]*, *overrides: Dict[str, Any]*)

Implementation of the pre_graph_execute hook. This just converts the inputs to the format the user-facing hook is expecting – performing a walk of the DAG to pass in the set of nodes to execute. Delegates to the interface method.

pre_node_execute(*, *run_id: str*, *node_: Node*, *kwargs: Dict[str, Any]*, *task_id: str | None = None*)

Wraps the before_execution method, providing a bridge to an external-facing API. Do not override this!

pre_task_execute(*, *run_id: str*, *task_id: str*, *nodes: List[Node]*, *inputs: Dict[str, Any]*, *overrides: Dict[str, Any]*, *spawning_task_id: str | None*, *purpose: None*)

Hook that is called immediately prior to task execution. Note that this is only useful in dynamic execution, although we reserve the right to add this back into the standard hamilton execution pattern.

Parameters:

- **run_id** – ID of the run, unique in scope of the driver.
- **task_id** – ID of the task, unique in scope of the driver.
- **nodes** – Nodes that are being executed
- **inputs** – Inputs to the task
- **overrides** – Overrides to task execution
- **spawning_task_id** – ID of the task that spawned this task
- **purpose** – Purpose of the current task group

run_after_graph_execution(**kwargs: Any*)

This is run after graph execution. This allows you to do anything you want after the graph executes, knowing the results of the execution/any errors.

Parameters:

- **graph** – Graph that is being executed

- **results** – Results of the graph execution
- **error** – Error that occurred, None if no error occurred
- **success** – Whether the graph executed successfully
- **run_id** – Run ID (unique in process scope) of the current run. Use this to track state.
- **future_kwargs** – Additional keyword arguments – this is kept for backwards compatibility

run_after_node_execution(***kwargs*)

Hook that is executed post node execution.

Parameters:

- **node_name** – Name of the node in question
- **node_tags** – Tags of the node
- **node_kwargs** – Keyword arguments passed to the node
- **node_return_type** – Return type of the node
- **result** – Output of the node, None if an error occurred
- **error** – Error that occurred, None if no error occurred
- **success** – Whether the node executed successfully
- **task_id** – The ID of the task, none if not in a task-based environment
- **run_id** – Run ID (unique in process scope) of the current run. Use this to track state.
- **future_kwargs** – Additional keyword arguments – this is kept for backwards compatibility

run_after_task_execution(*, *purpose: NodeGroupPurpose*, ***kwargs*)

Runs after all of the nodes associated with a task have been executed. By definition this is run *inside* of the executor and therefore may be run on separate or distributed processes.

Parameters:

- **task_id** – ID of the task that was just executed
- **run_id** – ID of the run this was under.
- **nodes** – Nodes that were part of this task
- **results** – Results of the task, per-node
- **success** – Whether the task was successful
- **error** – The error the task threw, if any
- **future_kwargs** – Reserved for backwards compatibility.
- **spawning_task_id** – ID of the task that spawned this task
- **purpose** – Purpose of the current task group

run_after_task_expansion(*, *parameters: dict[str, Any]*, ***kwargs*)

Runs after task expansion in Parallelize/Collect blocks. This allows you to capture information about the task that was expanded.

Parameters:

- **run_id** – ID of the run, unique in scope of the driver.
- **task_id** – ID of the task that was expanded.
- **parameters** – Parameters that were passed to the task.
- **future_kwargs** – Additional keyword arguments – this is kept for backwards compatibility.

run_after_task_grouping(*, *task_ids: List[str]*, ***kwargs*)

Runs after task grouping. This allows you to capture information about which tasks were created for a given run.

Parameters:

- **run_id** – ID of the run, unique in scope of the driver.
- **task_ids** – List of tasks that were grouped together.

- **future_kwargs** – Additional keyword arguments – this is kept for backwards compatibility.

run_before_graph_execution(*, *execution_path: Collection[str]*, ***kwargs: Any*)

This is run prior to graph execution. This allows you to do anything you want before the graph executes, knowing the basic information that was passed in.

Parameters:

- **graph** – Graph that is being executed
- **final_vars** – Output variables of the graph
- **inputs** – Input variables passed to the graph
- **overrides** – Overrides passed to the graph
- **execution_path** – Collection of nodes that will be executed – these are just the nodes (not input nodes) that will be run during the course of execution.
- **run_id** – Run ID (unique in process scope) of the current run. Use this to track state.
- **future_kwargs** – Additional keyword arguments – this is kept for backwards compatibility

run_before_node_execution(***kwargs*)

Hook that is executed prior to node execution.

Parameters:

- **node_name** – Name of the node.
- **node_tags** – Tags of the node
- **node_kwargs** – Keyword arguments to pass to the node
- **node_return_type** – Return type of the node
- **task_id** – The ID of the task, none if not in a task-based environment
- **run_id** – Run ID (unique in process scope) of the current run. Use this to track state.

- **node_input_types** – the input types to the node and what it is expecting
- **future_kwargs** – Additional keyword arguments – this is kept for backwards compatibility

run_before_task_execution(*, *purpose: NodeGroupPurpose*, ***kwargs*)

Runs prior to any of the nodes associated with a task. By definition this is run *inside* of the executor and therefore may be run on separate or distributed processes.

Parameters:

- **task_id** – ID of the task we're launching.
- **run_id** – ID of the run this is under.
- **nodes** – Nodes that are part of this task
- **inputs** – Inputs to the task
- **overrides** – Overrides passed to the task
- **future_kwargs** – Reserved for backwards compatibility.
- **spawning_task_id** – ID of the task that spawned this task
- **purpose** – Purpose of the current task group

plugins.h_ddog.DDOGTracer

class hamilton.plugins.h_ddog.DDOGTracer(*root_name: str*, *include_causal_links: bool = False*, *service: str = None*)

Lifecycle adapter to use datadog to run tracing on node execution. This works with the following execution environments: 1. Vanilla Hamilton – no task-based computation, just nodes 2. Task-based, synchronous 3. Task-based with Multithreading, Ray, and Dask It will likely work with others, although we have not yet tested them. This does not work with async (yet).

Note that this is not a typical use of Datadog if you're not using hamilton for a microservice. It does work quite nicely, however! Monitoring ETLs is not a typical datadog case (you can't see relationships between nodes/tasks or data summaries), but it is easy enough to work with and gives some basic information.

This tracer bypasses context management so we can more accurately track relationships between nodes/tags. Also, we plan to get this working with OpenTelemetry, and use that for datadog integration.

To use this, you'll want to run *pip install sf-hamilton[ddog]* (or *pip install "sf-hamilton[ddog]"* if using zsh)

__init__(*root_name: str*, *include_causal_links: bool = False*, *service: str = None*)

Creates a DDOGTracer. This has the option to specify some parameters.

Parameters:

- **root_name** – Name of the root trace/span. Due to the way datadog inherits, this will inherit an active span.
- **include_causal_links** – Whether or not to include span causal links. Note that there are some edge-cases here, and This is in beta for datadog, and actually broken in the current client, but it has been fixed and will be released shortly: https://github.com/DataDog/dd-trace-py/issues/8049. Furthermore, the query on datadog is slow for displaying causal links. We've disabled this by default, but feel free to test it out – its likely they'll be improving the docum
- **service** – Service name – will pick it up from the environment through DDOG if not available.

post_graph_execute(*, *run_id: str*, *graph: FunctionGraph*, *success: bool*, *error: Exception | None*, *results: Dict[str, Any] | None*)

Just delegates to the interface method, passing in the right data.

post_node_execute(*, *run_id: str*, *node_: Node*, *kwargs: Dict[str, Any]*, *success: bool*, *error: Exception | None*, *result: Any | None*, *task_id: str | None = None*)

Wraps the after_execution method, providing a bridge to an external-facing API. Do not override this!

post_task_execute(*, *run_id: str*, *task_id: str*, *nodes: List[Node]*, *results: Dict[str, Any] | None*, *success: bool*, *error: Exception*, *spawning_task_id: str | None*, *purpose: None*)

Hook called immediately after task execution. Note that this is only useful in dynamic execution, although we reserve the right to add this back into the standard hamilton execution pattern.

Parameters:

- **run_id** – ID of the run, unique in scope of the driver.
- **task_id** – ID of the task
- **nodes** – Nodes that were executed
- **results** – Results of the task
- **success** – Whether or not the task executed successfully
- **error** – The error that was raised, if any
- **spawning_task_id** – ID of the task that spawned this task
- **purpose** – Purpose of the current task group

pre_graph_execute(*, *run_id: str*, *graph: FunctionGraph*, *final_vars: List[str]*, *inputs: Dict[str, Any]*, *overrides: Dict[str, Any]*)

Implementation of the pre_graph_execute hook. This just converts the inputs to the format the user-facing hook is expecting – performing a walk of the DAG to pass in the set of nodes to execute. Delegates to the interface method.

pre_node_execute(*, *run_id: str*, *node_: Node*, *kwargs: Dict[str, Any]*, *task_id: str | None = None*)

Wraps the before_execution method, providing a bridge to an external-facing API. Do not override this!

pre_task_execute(*, *run_id: str*, *task_id: str*, *nodes: List[Node]*, *inputs: Dict[str, Any]*, *overrides: Dict[str, Any]*, *spawning_task_id: str | None*, *purpose: None*)

Hook that is called immediately prior to task execution. Note that this is only useful in dynamic execution, although we reserve the right to add this back into the standard hamilton execution pattern.

Parameters:

- **run_id** – ID of the run, unique in scope of the driver.
- **task_id** – ID of the task, unique in scope of the driver.
- **nodes** – Nodes that are being executed
- **inputs** – Inputs to the task
- **overrides** – Overrides to task execution

- **spawning_task_id** – ID of the task that spawned this task
- **purpose** – Purpose of the current task group

run_after_graph_execution(*, *error: Exception | None*, *run_id: str*, ***future_kwargs: Any*)

Runs after graph execution. Garbage collects + finishes the root span.

Parameters:

- **error** – Error the graph raised when running, if any
- **run_id** – ID of the run
- **future_kwargs** – reserved for future keyword arguments/backwards compatibility.

run_after_node_execution(*, *node_name: str*, *error: Exception | None*, *task_id: str | None*, *run_id: str*, ***future_kwargs: Any*)

Runs after a node's execution – completes the span.

Parameters:

- **node_name** – Name of the node
- **error** – Error that the node raised, if any
- **task_id** – Task ID that spawned the node
- **run_id** – ID of the run.
- **future_kwargs** – reserved for future keyword arguments/backwards compatibility.

run_after_task_execution(*, *task_id: str*, *run_id: str*, *error: Exception*, ***future_kwargs*)

Rusn after task execution. Finishes task-level spans.

Parameters:

- **task_id** – ID of the task, ID of the run.
- **run_id** – ID of the run
- **error** – Error the graph raised when running, if any
- **future_kwargs** – Future keyword arguments for backwards compatibility

run_before_graph_execution(*, *run_id: str*, ***future_kwargs: Any*)

Runs before graph execution – sets the state so future ones can reference it.

Parameters:

- **run_id** – ID of the run
- **future_kwargs** – reserved for future keyword arguments/backwards compatibility.

run_before_node_execution(*, *node_name: str*, *node_kwargs: Dict[str, Any]*, *node_tags: Dict[str, Any]*, *task_id: str | None*, *run_id: str*, ***future_kwargs: Any*)

Runs before a node's execution. Sets up/stores spans.

Parameters:

- **node_name** – Name of the node.
- **node_kwargs** – Keyword arguments of the node.
- **node_tags** – Tags of the node (they'll get stored as datadog tags)
- **task_id** – Task ID that spawned the node
- **run_id** – ID of the run.
- **future_kwargs** – reserved for future keyword arguments/backwards compatibility.

run_before_task_execution(*, *task_id: str*, *run_id: str*, ***future_kwargs*)

Runs before task execution. Sets up the task span.

Parameters:

- **task_id** – ID of the task

- **run_id** – ID of the run,
- **future_kwargs** – reserved for future keyword arguments/backwards compatibility.

class hamilton.plugins.h_ddog.AsyncDDOGTracer(*root_name: str*, *include_causal_links: bool = False*, *service: str | None = None*)

__init__(*root_name: str*, *include_causal_links: bool = False*, *service: str | None = None*)

Creates a AsyncDDOGTracer, the asyncio-friendly version of DDOGTracer.

This has the option to specify some parameters:

Parameters:

- **root_name** – Name of the root trace/span. Due to the way datadog inherits, this will inherit an active span.
- **include_causal_links** – Whether or not to include span causal links. Note that there are some edge-cases here, and This is in beta for datadog, and actually broken in the current client, but it has been fixed and will be released shortly: https://github.com/DataDog/dd-trace-py/issues/8049. Furthermore, the query on datadog is slow for displaying causal links. We've disabled this by default, but feel free to test it out – its likely they'll be improving the docum
- **service** – Service name – will pick it up from the environment through DDOG if not available.

async post_graph_construct(*graph: FunctionGraph*, *modules: List[ModuleType]*, *config: Dict[str, Any]*) → None

Runs after graph construction. This is a no-op for this plugin.

Parameters:

- **graph** – Graph that has been constructed.
- **modules** – Modules passed into the graph
- **config** – Config passed into the graph

async post_graph_execute(*run_id: str*, *graph: FunctionGraph*, *success: bool*, *error: Exception | None*, *results: Dict[str, Any] | None*) → None

Runs after graph execution. Garbage collects + finishes the root span.

Parameters:

- **run_id** – ID of the run, unique in scope of the driver.
- **graph** – Graph that was executed
- **success** – Whether or not the graph executed successfully
- **error** – Error that was raised, if any
- **results** – Results of the graph execution

async post_node_execute(*run_id: str*, *node_: Node*, *success: bool*, *error: Exception | None*, *result: Any*, *task_id: str | None = None*, ***future_kwargs: dict*) → None

Runs after a node's execution – completes the span.

Parameters:

- **run_id** – ID of the run, unique in scope of the driver.
- **node** – Node that is being executed
- **kwargs** – Keyword arguments that are being passed into the node
- **success** – Whether or not the node executed successfully
- **error** – The error that was raised, if any
- **result** – The result of the node execution, if no error was raised
- **task_id** – ID of the task, defaults to None if not in a task-based execution

async pre_graph_execute(*run_id: str*, *graph: FunctionGraph*, *final_vars: List[str]*, *inputs: Dict[str, Any]*, *overrides: Dict[str, Any]*) → None

Runs before graph execution – sets the state so future ones can reference it.

Parameters:

- **run_id** – ID of the run, unique in scope of the driver.
- **graph** – Graph that is being executed

- **final_vars** – Variables we are extracting from the graph
- **inputs** – Inputs to the graph
- **overrides** – Overrides to graph execution

async pre_node_execute(*run_id: str*, *node_: Node*, *kwargs: Dict[str, Any]*, *task_id: str | None = None*) → None

Runs before a node's execution. Sets up/stores spans.

Parameters:

- **run_id** – ID of the run, unique in scope of the driver.
- **node** – Node that is being executed
- **kwargs** – Keyword arguments that are being passed into the node
- **task_id** – ID of the task, defaults to None if not in a task setting

lifecycle.FunctionInputOutputTypeChecker

Use this hook to print out data before/after a node's execution for debugging

class hamilton.lifecycle.default.FunctionInputOutputTypeChecker(*check_input: bool = True*, *check_output: bool = True*)

This lifecycle hook checks the input and output types of a function.

It is a simple, but very strict type check against the declared type with what was actually received. E.g. if you don't want to check the types of a dictionary, don't annotate it with a type.

__init__(*check_input: bool = True*, *check_output: bool = True*)

Constructor.

Parameters:

- **check_input** – check inputs to all functions
- **check_output** – check outputs to all functions

post_node_execute(*, *run_id: str*, *node_: Node*, *kwargs: Dict[str, Any]*, *success: bool*, *error: Exception | None*, *result: Any | None*, *task_id: str | None = None*)

Wraps the after_execution method, providing a bridge to an external-facing API. Do not override this!

pre_node_execute(*, *run_id: str*, *node_: Node*, *kwargs: Dict[str, Any]*, *task_id: str | None = None*)

Wraps the before_execution method, providing a bridge to an external-facing API. Do not override this!

run_after_node_execution(*node_name: str*, *node_tags: Dict[str, Any]*, *node_kwargs: Dict[str, Any]*, *node_return_type: type*, *result: Any*, *error: Exception | None*, *success: bool*, *task_id: str | None*, *run_id: str*, ***future_kwargs: Any*)

Checks that the result type matches the expected node return type.

run_before_node_execution(*node_name: str*, *node_tags: Dict[str, Any]*, *node_kwargs: Dict[str, Any]*, *node_return_type: type*, *task_id: str | None*, *run_id: str*, *node_input_types: Dict[str, Any]*, ***future_kwargs: Any*)

Checks that the result type matches the expected node return type.

plugins.h_slack.SlackNotifier

Provides a Slack notifier for Apache Hamilton execution. Must have *slack_sdk* installed to use it:

pip install sf-hamilton[slack] (use quotes if using zsh)

class hamilton.plugins.h_slack.SlackNotifier(*api_key: str*, *channel: str*, ***kwargs*)

This is a adapter that sends a message to a slack channel when a node is executed & fails.

Note: you need to have slack_sdk installed for this to work. If you don't have it installed, you can install it with *pip install slack_sdk* (or *pip install sf-hamilton[slack]* – use quotes if you're using zsh).

```
from hamilton.plugins import h_slack

dr = (
    driver.Builder()
    .with_config({})
    .with_modules(some_modules)
    .with_adapters(h_slack.SlackNotifier(api_key="YOUR_API_KEY",
channel="YOUR_CHANNEL"))
    .build()
)
```

```
# and then when you call .execute() or .materialize() you'll get
a message in your slack channel!
```

__init__(*api_key: str*, *channel: str*, ***kwargs*)

Constructor.

Parameters:

- **api_key** – API key to use for sending messages.
- **channel** – Channel to send messages to.

post_node_execute(***, *run_id: str*, *node_: Node*, *kwargs: Dict[str, Any]*, *success: bool*, *error: Exception | None*, *result: Any | None*, *task_id: str | None = None*)

Wraps the after_execution method, providing a bridge to an external-facing API. Do not override this!

pre_node_execute(***, *run_id: str*, *node_: Node*, *kwargs: Dict[str, Any]*, *task_id: str | None = None*)

Wraps the before_execution method, providing a bridge to an external-facing API. Do not override this!

run_after_node_execution(*node_name: str*, *node_tags: Dict[str, Any]*, *node_kwargs: Dict[str, Any]*, *node_return_type: type*, *result: Any*, *error: Exception | None*, *success: bool*, *task_id: str | None*, *run_id: str*, ***future_kwargs: Any*)

Sends a message to the slack channel after a node is executed.

run_before_node_execution(*node_name: str*, *node_tags: Dict[str, Any]*, *node_kwargs: Dict[str, Any]*, *node_return_type: type*, ***future_kwargs: Any*)

Placeholder required to subclass *NodeExecutionMethod*

lifecycle.GracefulErrorAdapter

class hamilton.lifecycle.default.GracefulErrorAdapter(*error_to_catch: Type[Exception]*, *sentinel_value: Any = None*, *try_all_parallel: bool = True*, *allow_injection: bool = True*)

Gracefully handles errors in a graph's execution. This allows you to proceed despite failure, dynamically pruning branches. While it still runs every node, it replaces them with no-ops if any upstream required dependencies fail (including optional dependencies).

__init__(*error_to_catch: Type[Exception]*, *sentinel_value: Any = None*, *try_all_parallel: bool = True*, *allow_injection: bool = True*)

Initializes the adapter. Allows you to customize the error to catch (which exception your graph will throw to indicate failure), as well as the sentinel value to use in place of a node's result if it fails (this defaults to `None`).

Note that this is currently only compatible with the dict-based result builder (use at your own risk with pandas series, etc...).

Be careful using `None` as the default – feel free to replace it with a sentinel value of your choice (this could negatively impact your graph's execution if you actually *do* intend to use `None` return values).

You can use this as follows:

```
# my_module.py
# custom exception
class DoNotProceed(Exception):
    pass


def wont_proceed() -> int:
    raise DoNotProceed()


def will_proceed() -> int:
    return 1


def never_reached(wont_proceed: int) -> int:
    return 1  # this should not be reached


dr = (
    driver.Builder()
    .with_modules(my_module)
    .with_adapters(
        default.GracefulErrorAdapter(
            error_to_catch=DoNotProceed,
            sentinel_value=None
        )
    )
    .build()
)
dr.execute(
    ["will_proceed", "never_reached"]
)  # will return {'will_proceed': 1, 'never_reached': None}
```

Note you can customize the error you want it to fail on and the sentinel value to use in place of a node's result if it fails.

For Parallelizable nodes, this adapter will attempt to iterate over the node outputs. If an error occurs, the sentinel value is returned and no more iterations over the node will occur. Meaning if item (3) fails out of 1,2,3,4,5, 4/5 will not run. If you set `try_all_parallel` to be False, it only sends one sentinel value into the parallelize sub-dag.

Here's an example for parallelizable to demonstrate try_all_parallel:

```
# parallel_module.py
# custom exception
class DoNotProceed(Exception):
    pass


def start_point() -> Parallelizable[int]:
    for i in range(5):
        if i == 3:
            raise DoNotProceed()
        yield i


def inner(start_point: int) -> int:
    return start_point


def gather(inner: Collect[int]) -> list[int]:
    return inner


dr = (
    driver.Builder()
    .with_modules(parallel_module)
    .with_adapters(
        default.GracefulErrorAdapter(
            error_to_catch=DoNotProceed,
            sentinel_value=None,
            try_all_parallel=True,
        )
    )
    .build()
)
dr.execute(["gather"])  # will return {'gather':
[0,1,2,None]}

dr = (
    driver.Builder()
    .with_modules(parallel_module)
    .with_adapters(
        default.GracefulErrorAdapter(
            error_to_catch=DoNotProceed,
```

```
            sentinel_value=None,
            try_all_parallel=False,
        )
    )
    .build()
)
dr.execute(["gather"])  # will return {'gather': [None]}
```

Parameters:

- **error_to_catch** – The error to catch
- **sentinel_value** – The sentinel value to use in place of a node's result if it fails
- **try_all_parallel** – Gather parallelizable outputs until a failure, then add a Sentinel.
- **allow_injection** – Flag for considering the `accept_error_sentinels` tag. Defaults to True.

default.accept_error_sentinels()

Tag a function to allow passing in error sentinels.

For use with `GracefulErrorAdapter`. The standard adapter behavior is to skip a node when an error sentinel is one of its inputs. This decorator will cause the node to run, and place the error sentinel into the appropriate input.

Take care to ensure your sentinels are easily distinguishable if you do this - see the note in the GracefulErrorAdapater docstring.

A use case is any data or computation aggregation step that still wants partial results, or considers a failure interesting enough to log or notify.

```
SENTINEL = object()

...

@accept_error_sentinels
def results_gathering(result_1: float, result_2: float) ->
dict[str, Any]:
    answer = {}
    for name, res in zip(["result 1", "result 2"], [result_1,
result_2])
        answer[name] = res
        if res is SENTINEL:
            answer[name] = "Node failure: no result"
```

```
            # You may want side-effects for a failure.
            _send_text_that_your_runs_errored()
    return answer

...
adapter = GracefulErrorAdapter(sentinel_value=SENTINEL)
...
```

plugins.h_spark.SparkInputValidator

class hamilton.plugins.h_spark.SparkInputValidator

This is a graph hook adapter that allows you to get past a <4.0.0 limitation in spark. Spark has the option to choose between spark connect and spark, which largely have the same API. That said, they don't have the proper subclass relationships, which make hamilton fail on the input type checking.

See the following for more information as to why this is necessary: - https://community.databricks.com/t5/data-engineering/pyspark-sql-connect-dataframe-dataframe-vs-pyspark-sql-dataframe/td-p/71055 - https://issues.apache.org/jira/browse/SPARK-47909

You can access an instance of this through the convenience variable *SPARK_INPUT_CHECK*. This allows you to bypass that. This has to be used with the driver builder pattern – this will look as follows:

```
from hamilton import driver
from hamilton.plugins import h_spark

dr =
driver.Builder().with_modules(...).with_adapters(h_spark.SPARK_INPUT_CHECK)
```

Then run it as you would normally. Note that in spark==4.0.0, you will only need the spark session check, not the dataframe check.

do_validate_input(*, *node_type: type*, *input_value: Any*) → bool

Validates the input. Treats connect/classic sessios/dataframe as interchangeable.

plugins.h_narhwals.NarwhalsAdapter

Provides a convenience wrapper for the *Narwhals* library; use the Narwhals decorator underneath. Must have *Narwhals* installed to use it:

pip install "sf-hamilton[narwhals]"

class hamilton.plugins.h_narwhals.NarwhalsAdapter

Adapter to make it simpler to use narwhals with Hamilton.

```
from hamilton import base, driver
from hamilton.plugins import h_narwhals
import example

# pandas
dr = (
    driver.Builder()
    .with_config({"load": "pandas"})
    .with_modules(example)
    .with_adapters(
        h_narwhals.NarwhalsAdapter(),
        h_narwhals.NarwhalsDataFrameResultBuilder(
            base.PandasDataFrameResult()
        ),
    )
    .build()
)
result = dr.execute(
     [example.group_by_mean, example.example1],
     inputs={"col_name": "a"}
)
```

do_node_execute(*, *run_id: str*, *node_: Node*, *kwargs: Dict[str, Any]*, *task_id: str | None = None*) → Any

Method that is called to implement node execution. This can replace the execution of a node with something all together, augment it, or delegate it.

Parameters:

- **run_id** – ID of the run, unique in scope of the driver.
- **node** – Node that is being executed
- **kwargs** – Keyword arguments that are being passed into the node
- **task_id** – ID of the task, defaults to None if not in a task setting

run_to_execute_node(*, *node_name: str*, *node_tags: Dict[str, Any]*, *node_callable: Any*, *node_kwargs: Dict[str, Any]*, *task_id: str | None*, ***future_kwargs: Any*) → Any

This method is responsible for executing the node and returning the result.

It uses *nw_kwargs* from the node tags to know if any special flags should be passed to the narwhals decorator function.

Parameters:

- **node_name** – Name of the node.
- **node_tags** – Tags of the node.
- **node_callable** – Callable of the node.
- **node_kwargs** – Keyword arguments to pass to the node.
- **task_id** – The ID of the task, none if not in a task-based environment
- **future_kwargs** – Additional keyword arguments – this is kept for backwards compatibility

Returns:

The result of the node execution – up to you to return this.

plugins.h_narhwals.NarwhalsDataFrameResultBuilder

Result builder to be used with the NarwhalsAdapter. Must have *Narwhals* installed to use it:

pip install "sf-hamilton[narwhals]"

class hamilton.plugins.h_narwhals.NarwhalsDataFrameResultBuilder(*result_builder: ResultBuilder | LegacyResultMixin*)

Builds the result. It unwraps the narwhals parts of it and delegates to the passed in result builder.

```
from hamilton import base, driver
from hamilton.plugins import h_narwhals, h_polars
import example

# polars
dr = (
    driver.Builder()
    .with_config({"load": "polars"})
    .with_modules(example)
    .with_adapters(
        h_narwhals.NarwhalsAdapter(),
        h_narwhals.NarwhalsDataFrameResultBuilder(
```

```
            h_polars.PolarsDataFrameResult()
        ),
    )
    .build()
)
result = dr.execute(
    ["group_by_mean", "example1"],
    inputs={"col_name": "a"}
)
```

__init__(*result_builder: ResultBuilder | LegacyResultMixin*)

build_result(***outputs: Any*) → Any

Given a set of outputs, build the result.

Parameters:

outputs – the outputs from the execution of the graph.

Returns:

the result of the execution of the graph.

do_build_result(*outputs: Dict[str, Any]*) → Any

Implements the do_build_result method from the BaseDoBuildResult class. This is kept from the user as the public-facing API is build_result, allowing us to change the API/implementation of the internal set of hooks

input_types() → List[Type[Type]]

Gives the applicable types to this result builder. This is optional for backwards compatibility, but is recommended.

Returns:

A list of types that this can apply to.

output_type() → Type

Returns the output type of this result builder :return: the type that this creates

plugins.h_mlflow.MLFlowTracker

class hamilton.plugins.h_mlflow.MLFlowTracker(*tracking_uri: str | None = None*, *registry_uri: str | None = None*, *artifact_location: str | None = None*, *experiment_name: str = 'Hamilton'*, *experiment_tags: dict | None = None*, *experiment_description: str | None = None*, *run_id: str |*

None = None, *run_name: str | None = None*, *run_tags: dict | None = None*, *run_description: str | None = None*, *log_system_metrics: bool = False*)

Driver adapter logging Hamilton execution results to an MLFlow server.

__init__(*tracking_uri: str | None = None*, *registry_uri: str | None = None*, *artifact_location: str | None = None*, *experiment_name: str = 'Hamilton'*, *experiment_tags: dict | None = None*, *experiment_description: str | None = None*, *run_id: str | None = None*, *run_name: str | None = None*, *run_tags: dict | None = None*, *run_description: str | None = None*, *log_system_metrics: bool = False*)

Configure the MLFlow client and experiment for the lifetime of the tracker

Parameters:

- **tracking_uri** – Destination of the logged artifacts and metadata. It can be a filesystem, database, or server. [reference](https://mlflow.org/docs/latest/getting-started/tracking-server-overview/index.html)
- **registry_uri** – Destination of the registered models. By default it's the same as the tracking destination, but they can be different. [reference](https://mlflow.org/docs/latest/getting-started/registering-first-model/index.html)
- **artifact_location** – Root path on tracking server where experiment is stored
- **experiment_name** – MLFlow experiment name used to group runs.
- **experiment_tags** – Tags to query experiments programmatically (not displayed).
- **experiment_description** – Description of the experiment displayed
- **run_id** – Run id to log to an existing run (every execution logs to the same run)
- **run_name** – Run name displayed and used to query runs. You can have multiple runs with the same name but different run ids.
- **run_tags** – Tags to query runs and appears as columns in the UI for filtering and grouping. It

automatically includes serializable inputs and Driver config.

- **run_description** – Description of the run displayed
- **log_system_metrics** – Log system metrics to display (requires additonal dependencies)

post_graph_construct(*, *graph: FunctionGraph*, *modules: List[ModuleType]*, *config: Dict[str, Any]*)

Hooks that is called after the graph is constructed.

Parameters:

- **graph** – Graph that has been constructed.
- **modules** – Modules passed into the graph
- **config** – Config passed into the graph

post_graph_execute(*, *run_id: str*, *graph: FunctionGraph*, *success: bool*, *error: Exception | None*, *results: Dict[str, Any] | None*)

Just delegates to the interface method, passing in the right data.

post_node_execute(*, *run_id: str*, *node_: Node*, *kwargs: Dict[str, Any]*, *success: bool*, *error: Exception | None*, *result: Any | None*, *task_id: str | None = None*)

Wraps the after_execution method, providing a bridge to an external-facing API. Do not override this!

pre_graph_execute(*, *run_id: str*, *graph: FunctionGraph*, *final_vars: List[str]*, *inputs: Dict[str, Any]*, *overrides: Dict[str, Any]*)

Implementation of the pre_graph_execute hook. This just converts the inputs to the format the user-facing hook is expecting – performing a walk of the DAG to pass in the set of nodes to execute. Delegates to the interface method.

pre_node_execute(*, *run_id: str*, *node_: Node*, *kwargs: Dict[str, Any]*, *task_id: str | None = None*)

Wraps the before_execution method, providing a bridge to an external-facing API. Do not override this!

run_after_graph_construction(*, *config: dict[str, Any]*, ***kwargs*)

Store the Driver config before creating the graph

run_after_graph_execution(*success: bool*, **args*, ***kwargs*)

End the MLFlow run

run_after_node_execution(*, *node_name: str*, *node_return_type: Type*, *node_tags: dict*, *node_kwargs: dict*, *result: Any*, ***kwargs*)

Log materializers and final vars as artifacts

run_before_graph_execution(*, *run_id: str*, *final_vars: List[str]*, *inputs: Dict[str, Any]*, *graph: HamiltonGraph*, ***kwargs*)

Create and start MLFlow run. Log graph version, run_id, inputs, overrides

run_before_node_execution(**args*, ***kwargs*)

Placeholder required to subclass NodeExecutionHook

lifecycle.NoEdgeAndInputTypeChecking

Use this hook turn off edge and input type checking during graph construction and execution; the only time you'd really want this is during some really fast and loose development. Otherwise production use of this should be frowned upon.

class hamilton.lifecycle.default.NoEdgeAndInputTypeChecking

Permissive adapter to help you skip edge and input type checking.

Useful for development.

```
from hamilton import driver
from hamilton.lifecycle import NoEdgeAndInputTypeChecking

dr =
driver.Builder().with_adapters(NoEdgeAndInputTypeChecking()).build()

# now driver is built without any type checking
dr.execute([...], ...)
```

check_edge_types_match(*type_from: type*, *type_to: type*, ***kwargs: Any*) → bool

This is run to check if edge types match. Note that this is an OR functionality – this is run after we do some default checks, so this can only be permissive. Return True - always

do_check_edge_types_match(*, *type_from: type*, *type_to: type*) → bool

Wraps the check_edge_types_match method, providing a bridge to an external-facing API. Do not override this!

do_validate_input(*, *node_type: type*, *input_value: Any*) → bool

Wraps the validate_input method, providing a bridge to an external-facing API. Do not override this!

validate_input(*node_type: type*, *input_value: Any*, ***kwargs: Any*) → bool

This is run to check if the input is valid for the node type. Note that this is an OR functionality – this is run after we do some default checks, so this can only be permissive. Returns True - always.

plugins.h_openlineage.OpenLineageAdapter

class hamilton.plugins.h_openlineage.OpenLineageAdapter(*client: OpenLineageClient*, *namespace: str*, *job_name: str*)

This adapter emits OpenLineage events.

```
# create the openlineage client
from openlineage.client import OpenLineageClient

# write to file
from openlineage.client.transport.file import FileConfig,
FileTransport
file_config = FileConfig(
    log_file_path="/path/to/your/file",
    append=False,
)
client = OpenLineageClient(transport=FileTransport(file_config))

# write to HTTP, e.g. marquez
client = OpenLineageClient(url="http://localhost:5000")

# create the adapter
adapter = OpenLineageAdapter(client, "my_namespace",
"my_job_name")

# add to Hamilton
# import your pipeline code
dr =
driver.Builder().with_modules(YOUR_MODULES).with_adapters(adapter).build()
# execute as normal -- and openlineage events will be emitted
dr.execute(...)
```

Note for data lineage to be emitted, you must use the "materializer" abstraction to provide metadata. See https://hamilton.apache.org/concepts/materialization/. This can be done via the *@datasaver()* and *@dataloader()* decorators, or using the *@load_from* or *@save_to* decorators, as well as passing in data savers and data loaders via *.with_materializers()* on the Driver Builder, or via *.materialize()* on the driver object.

__init__(*client: OpenLineageClient*, *namespace: str*, *job_name: str*)

Constructor. You pass in the OLClient.

Parameters:

- **self**
- **client**
- **namespace**
- **job_name**

Returns:

post_graph_execute(*run_id: str*, *graph: FunctionGraph*, *success: bool*, *error: Exception | None*, *results: Dict[str, Any] | None*)

Emits a Run COMPLETE or FAIL event.

Parameters:

- **run_id**
- **graph**
- **success**
- **error**
- **results**

Returns:

post_node_execute(*run_id: str*, *node_: Node*, *kwargs: Dict[str, Any]*, *success: bool*, *error: Exception | None*, *result: Any | None*, *task_id: str | None = None*)

Run Event: will emit a RUNNING event with updates on input/outputs.

A Job Event will be emitted for graph execution, and additional SQLJob facet if data was loaded from a SQL source.

A Dataset Event will be emitted if a dataloader or datasaver was used:

- input data set if loader
- output data set if saver
- appropriate facets will be added to the dataset where it makes sense.

TODO: attach statistics facets

Parameters:

- **run_id**
- **node**
- **kwargs**
- **success**
- **error**
- **result**
- **task_id**

Returns:

pre_graph_execute(*run_id: str*, *graph: FunctionGraph*, *final_vars: List[str]*, *inputs: Dict[str, Any]*, *overrides: Dict[str, Any]*)

Emits a Run START event. Emits a Job Event with the sourceCode Facet for the entire DAG as the job.

Parameters:

- **run_id**
- **graph**
- **final_vars**
- **inputs**
- **overrides**

Returns:

pre_node_execute(*run_id: str*, *node_: Node*, *kwargs: Dict[str, Any]*, *task_id: str | None = None*)

No event emitted.

ResultBuilders

This section helps determine what comes out of the box for determining how to construct a return type from `execute`.

Reference

Generic

Result builders help you augment what is returned by the driver's *execute()* function. Here are the generic ones.

class hamilton.base.ResultMixin
: Legacy result builder – see lifecycle methods for more information.

class hamilton.base.DictResult
: Simple function that returns the dict of column -> value results.

  It returns the results as a dictionary, where the keys map to outputs requested, and values map to what was computed for those values.

  Use this when you want to:

  1. debug dataflows.
  2. have heterogeneous return types.
  3. Want to manually transform the result into something of your choosing.

```
from hamilton import base, driver

dict_builder = base.DictResult()
adapter = base.SimplePythonGraphAdapter(dict_builder)
dr = driver.Driver(config, *modules, adapter=adapter)
dict_result = dr.execute([...], inputs=...)
```

  Note, if you just want the dict result + the SimplePythonGraphAdapter, you can use the DefaultAdapter

```
adapter = base.DefaultAdapter()
```

static build_result(***outputs: Dict[str, Any]*) → Dict
This function builds a simple dict of output -> computed values.

input_types() → List[Type[Type]] | None
Gives the applicable types to this result builder. This is optional for backwards compatibility, but is recommended.

Returns:

A list of types that this can apply to.

output_type() → Type
Returns the output type of this result builder :return: the type that this creates

Numpy

class hamilton.base.NumpyMatrixResult
Mixin for building a Numpy Matrix from the result of walking the graph.

All inputs to the build_result function are expected to be numpy arrays.

```
from hamilton import base, driver

adapter = base.SimplePythonGraphAdapter(base.NumpyMatrixResult())
dr = driver.Driver(config, *modules, adapter=adapter)
numpy_matrix = dr.execute([...], inputs=...)
```

static build_result(***outputs: Dict[str, Any]*) → matrix
Builds a numpy matrix from the passed in, inputs.

Note: this does not check that the inputs are all numpy arrays/array like things.

Parameters:

outputs – function_name -> np.array.

Returns:

numpy matrix

Pandas

class hamilton.base.PandasDataFrameResult

Mixin for building a pandas dataframe from the result.

It returns the results as a Pandas Dataframe, where the columns map to outputs requested, and values map to what was computed for those values. Note: this only works if the computed values are pandas series, or scalar values.

Use this when you want to create a pandas dataframe.

Example:

```
from hamilton import base, driver
df_builder = base.PandasDataFrameResult()
adapter = base.SimplePythonGraphAdapter(df_builder)
dr =  driver.Driver(config, *modules, adapter=adapter)
df = dr.execute([...], inputs=...)
```

static build_result(**outputs: Dict[str, Any*]) → DataFrame

Builds a Pandas DataFrame from the outputs.

This function will check the index types of the outputs, and log warnings if they don't match. The behavior of pd.Dataframe(outputs) is that it will do an outer join based on indexes of the Series passed in.

Parameters:

outputs – the outputs to build a dataframe from.

class hamilton.base.StrictIndexTypePandasDataFrameResult

A ResultBuilder that produces a dataframe only if the index types match exactly.

Note: If there is no index type on some outputs, e.g. the value is a scalar, as long as there exists a single pandas index type, no error will be thrown, because a dataframe can be easily created.

Use this when you want to create a pandas dataframe from the outputs, but you want to ensure that the index types match exactly.

To use:

```
from hamilton import base, driver
strict_builder = base.StrictIndexTypePandasDataFrameResult()
adapter = base.SimplePythonGraphAdapter(strict_builder)
```

```
dr =  driver.Driver(config, *modules, adapter=adapter)
df = dr.execute([...], inputs=...)  # this will now error if
index types mismatch.
```

static build_result(***outputs: Dict[str, Any]*) → DataFrame

Builds a Pandas DataFrame from the outputs.

This function will check the index types of the outputs, and log warnings if they don't match. The behavior of pd.Dataframe(outputs) is that it will do an outer join based on indexes of the Series passed in.

Parameters:

outputs – the outputs to build a dataframe from.

Polars

class hamilton.plugins.h_polars.PolarsDataFrameResult

A ResultBuilder that produces a polars dataframe.

Use this when you want to create a polars dataframe from the outputs. Caveat: you need to ensure that the length of the outputs is the same, otherwise you will get an error; mixed outputs aren't that well handled.

To use:

```
from hamilton import base, driver
from hamilton.plugins import polars_extensions

polars_builder = polars_extensions.PolarsDataFrameResult()
adapter = base.SimplePythonGraphAdapter(polars_builder)
dr = driver.Driver(config, *modules, adapter=adapter)
df = dr.execute([...], inputs=...)  # returns polars dataframe
```

Note: this is just a first attempt at something for Polars. Think it should handle more? Come chat/open a PR!

build_result(***outputs: Dict[str, Series | DataFrame | Any]*) → DataFrame

This is the method that Hamilton will call to build the final result. It will pass in the results of the requested outputs that you passed in to the execute() method.

Note: this function could do smarter things; looking for contributions here!

Parameters:

outputs – The results of the requested outputs.

Returns:

a polars DataFrame.

Dask

class hamilton.plugins.h_dask.DaskDataFrameResult

static build_result(***outputs: Dict[str, Any]*) → Any

Builds a dask dataframe from the outputs.

This has some assumptions:

1. the order specified in the output will mirror the order of "joins" here.
2. it tries to massage types into dask types where it can
3. otherwise it duplicates any "scalars/objects" using the first valid input with an index as the template. It assumes a single partition.

plugins.h_pyarrow.PyarrowTableResult

class hamilton.plugins.h_pyarrow.PyarrowTableResult

Add this result builder to a materializer's *combine* statement to convert your dataframe object to a pyarrow representation and make it compatible with pyarrow DataSavers.

It implicitly support input_type == Any, but it expects dataframe objects implementing the dataframe interchange protocol: ref: https://arrow.apache.org/docs/python/interchange_protocol.html for example: - pandas - polars - dask - vaex - ibis - duckdb results

build_result(***outputs: Any*) → Any

This function converts objects implementing the *__dataframe__* protocol to a pyarrow table. It doesn't support receiving multiple outputs because it can't handle any joining logic.

ref: https://arrow.apache.org/docs/python/interchange_protocol.html

do_build_result(*outputs: Dict[str, Any]*) → Any

Implements the do_build_result method from the BaseDoBuildResult class. This is kept from the user as the public-facing API is build_result, allowing us to change the API/implementation of the internal set of hooks

input_types() → List[Type[Type]]
: Gives the applicable types to this result builder. This is optional for backwards compatibility, but is recommended.

  Returns:

  A list of types that this can apply to.

output_type() → Type
: Returns the output type of this result builder :return: the type that this creates

Custom ResultBuilder

If you have a use case for a custom ResultBuilder, tell us on Slack or via a GitHub issues. Knowing about your use case and talking through help ensures we aren't duplicating effort, and that it'll be using part of the API we don't intend to change.

What you need to do

You need to implement a class that implements a single function - see GitHub:

```
class ResultBuilder(object):
    """Base class housing the result builder"""
    @abc.abstractmethod
    def build_result(self, **outputs: typing.Dict[str, typing.Any]) -
> typing.Any:
        """This function builds the result given the computed
values."""
        pass
```

For example:

```
import typing
from hamilton import lifecycle
class MyCustomBuilder(lifecycle.ResultBuilder):
    # add a constructor if you need to
    @staticmethod
    def build_result(**outputs: typing.Dict[str, typing.Any]) ->
YOUR_RETURN_TYPE:
        """Custom function you fill in"""
```

```
        # your logic would go here
        return OBJECT_OF_YOUR_CHOOSING
```

How to use it

You would then have the option to pair that with a graph adapter that takes in a ResultMixin object. E.g. `SimplePythonGraphAdapter`. See GraphAdapters for which ones take in a custom ResultMixin object.

You can pass the result builder or a graph adapters to the `driver.Builder(result_builder).with_adapters(...)` function.

I/O

This section contains any information about I/O within Apache Hamilton. If you're using materializers or the *save_to*/*load_from* decorator, you'll need this page to help you find the set of available loading/saving targets.

Reference

Using Data Adapters

This is an index of all the available data adapters, both savers and loaders. Note that some savers and loaders are the same (certain classes can handle both), but some are different. You will want to reference this when calling out to any of the following:

1. Using save_to [or for just exposing metadata datasaver].

2. Using load_from [or for just exposing metadata dataloader].

3. Using materializers.

To read these tables, you want to first look at the key to determine which format you want – these should be human-readable and familiar to you. Then you'll want to look at the *types* field to figure out which is the best for your case (the object you want to load from or save to).

Finally, look up the adapter params to see what parameters you can pass to the data adapters. The optional params come with their default value specified.

If you want more information, click on the *module*, it will send you to the code that implements it to see how the parameters are used.

As an example, say we wanted to save a pandas dataframe to a CSV file. We would first find the key *csv*, which would inform us that we want to call *save_to.csv* (or *to.csv* in the case of *materialize*). Then, we would look at the *types* field, finding that there is a pandas dataframe adapter. Finally, we would look at the *params* field, finding that we can pass *path*, and (optionally) *sep* (which we'd realize defaults to , when looking at the code).

All together, we'd end up with:

```
import pandas as pd
from hamilton.function_modifiers import value, save_to

@save_to.csv(path=value("my_file.csv"))
def my_data(...) -> pd.DataFrame:
    ...
```

For a less "abstracted" approach, where you just expose metadata from saving and loading, you can annotated your saving/loading functions to do so, e.g. analogous to the above you could do:

```
import pandas as pd
from hamilton.function_modifiers import datasaver

 def my_data(...) -> pd.DataFrame:
    # your function
    ...
    return _df  # return some df

@datasaver
def my_data_saver(my_data: pd.DataFrame, path: str) -> dict:
    # code to save my_data
    return {"path": path, "type": "csv", ...} # add other metadata
```

See dataloader for more information on how to load data and expose metadata via this more lighter weight way.

If you want to extend the *@save_to* or *@load_from* decorators, see Using Data Adapters for documentation, and the example in the repository for an example of how to do so.

Note that you will need to call *registry.register_adapters* (or import a module that does that) prior to dynamically referring to these in the code – otherwise we won't know about them, and won't be able to access that key!

Data Loaders

key	loader params	types	module
json	path `str`	`dict` `list`	hamilton.io.default_data_
json			hamilton.plugins.pandas_

key	loader params	types	module
	filepath_or_buffer `Union` chunksize `Optional=None` compression `Union=infer` convert_axes `Optional=None` convert_dates `Union=True` date_unit `Optional=None` dtype `Union=None` dtype_backend `Optional=None` encoding `Optional=None` encoding_errors `Optional=strict` engine `str=ujson` keep_default_dates `bool=True` lines `bool=False` nrows `Optional=None` orient `Optional=None` precise_float `bool=False` storage_options `Optional=None` typ `str=frame`	`DataFrame`	
json		`DataFrame`	hamilton.plugins.polars_

key	loader params	types	module

source `Union` schema

```
 collections.abc.Mapping[str,
typing.Union[ForwardRef('DataTypeClass'),
ForwardRef('DataType'),         type[int],
type[float],    type[bool],     type[str],
type['date'],               type['time'],
type['datetime'],       type['timedelta'],
type[list[typing.Any]],
type[tuple[typing.Any, ...]], type[bytes],
type[object], type['Decimal'], type[None],
NoneType]] | collections.abc.Sequence[str
|                               tuple[str,
typing.Union[ForwardRef('DataTypeClass'),
ForwardRef('DataType'),         type[int],
type[float],    type[bool],     type[str],
type['date'],               type['time'],
type['datetime'],       type['timedelta'],
type[list[typing.Any]],
type[tuple[typing.Any, ...]], type[bytes],
type[object], type['Decimal'], type[None],
NoneType]]=None
```

schema_overrides

```
 collections.abc.Mapping[str,
typing.Union[ForwardRef('DataTypeClass'),
ForwardRef('DataType'),         type[int],
type[float],    type[bool],     type[str],
type['date'],               type['time'],
type['datetime'],       type['timedelta'],
type[list[typing.Any]],
type[tuple[typing.Any, ...]], type[bytes],
type[object], type['Decimal'], type[None],
NoneType]] | collections.abc.Sequence[str
|                               tuple[str,
typing.Union[ForwardRef('DataTypeClass'),
ForwardRef('DataType'),         type[int],
type[float],    type[bool],     type[str],
type['date'],               type['time'],
type['datetime'],       type['timedelta'],
type[list[typing.Any]],
type[tuple[typing.Any, ...]], type[bytes],
type[object], type['Decimal'], type[None],
NoneType]]=None
```

key	loader params	types	module
json	path `Union`	`XGBModel` `Booster`	hamilton.plugins.xgboost
literal	value `Any`	`Any`	hamilton.io.default_data_
file	path `str` encoding `str=utf-8`	`str`	hamilton.io.default_data_
file	path `Union`	`LGBMModel` `Booster` `CVBooster`	hamilton.plugins.lightgbm
pickle	path `str`	`object` `Any`	hamilton.io.default_data_
pickle	filepath_or_buffer `Union=None` path `Union=None` compression `Union=infer` storage_options `Optional=None`	`DataFrame`	hamilton.plugins.pandas_
environment	names `Tuple`	`dict`	hamilton.io.default_data_
yaml	path `Union`	`str` `int` `float` `bool` `dict` `list`	hamilton.plugins.yaml_ex
npy		`ndarray`	hamilton.plugins.numpy_

key	loader params	types	module
	path Union mmap_mode Optional=None allow_pickle Optional=None fix_imports Optional=None encoding Literal=ASCII		
csv	path Union sep Optional=, delimiter Optional=None header Union=infer names Optional=None index_col Union=None usecols Union=None dtype Union=None engine Optional=None converters Optional=None true_values Optional=None false_values Optional=None skipinitialspace Optional=False skiprows Union=None skipfooter int=0 nrows Optional=None na_values Union=None keep_default_na bool=True na_filter bool=True verbose bool=False skip_blank_lines bool=True parse_dates Union=False keep_date_col bool=False date_format Optional=None dayfirst bool=False cache_dates bool=True iterator bool=False chunksize Optional=None compression Union=infer thousands Optional=None decimal str=. lineterminator Optional=None quotechar Optional=None quoting int=0 doublequote bool=True escapechar Optional=None comment Optional=None encoding str=utf-8 encoding_errors Union=strict dialect Union=None on_bad_lines Union=error delim_whitespace bool=False low_memory bool=True memory_map bool=False float_precision Optional=None storage_options Optional=None dtype_backend Literal=numpy_nullable	DataFrame	hamilton.plugins.pandas_
csv		DataFrame	hamilton.plugins.polars_

key	loader params	types	module
	file `Union` has_header `bool=True` include_header `bool=True` columns `Union=None` new_columns `Sequence=None` separator `str=,` comment_char `str=None` quote_char `str="` skip_rows `int=0` dtypes `Union=None` null_values `Union=None` missing_utf8_is_empty_string `bool=False` ignore_errors `bool=False` try_parse_dates `bool=False` n_threads `int=None` infer_schema_length `int=100` batch_size `int=8192` n_rows `int=None` encoding `Union=utf8` low_memory `bool=False` rechunk `bool=True` use_pyarrow `bool=False` storage_options `Dict=None` skip_rows_after_header `int=0` row_count_name `str=None` row_count_offset `int=0` sample_size `int=1024` eol_char `str=` raise_if_empty `bool=True`		
csv	file `Union` has_header `bool=True` columns `Union=None` new_columns `Sequence=None` separator `str=,` comment_char `str=None` quote_char `str="` skip_rows `int=0` dtypes `Union=None` null_values `Union=None` missing_utf8_is_empty_string `bool=False` ignore_errors `bool=False` try_parse_dates `bool=False` n_threads `int=None` infer_schema_length `int=100` batch_size `int=8192` n_rows `int=None` encoding `Union=utf8` low_memory `bool=False` rechunk `bool=True` use_pyarrow `bool=False` storage_options `Dict=None` skip_rows_after_header `int=0` row_count_name `str=None` row_count_offset `int=0` eol_char `str=` raise_if_empty `bool=True`	`LazyFrame`	hamilton.plugins.polars_l

key	loader params	types	module
csv	spark `SparkSession` path `str` header `bool=True` sep `str=,`	`DataFrame`	hamilton.plugins.spark_e
parquet	path `Union` engine `Literal=auto` columns `Optional=None` storage_options `Optional=None` use_nullable_dtypes `bool=False` dtype_backend `Literal=numpy_nullable` filesystem `Optional=None` filters `Union=None`	`DataFrame`	hamilton.plugins.pandas_
parquet	file `Union` columns `Union=None` n_rows `int=None` use_pyarrow `bool=False` memory_map `bool=True` storage_options `Dict=None` parallel `Any=auto` row_count_name `str=None` row_count_offset `int=0` low_memory `bool=False` pyarrow_options `Dict=None` use_statistics `bool=True` rechunk `bool=True`	`DataFrame`	hamilton.plugins.polars_
parquet	file `Union` columns `Union=None` n_rows `int=None` use_pyarrow `bool=False` memory_map `bool=True` storage_options `Dict=None` parallel `Any=auto` row_count_name `str=None` row_count_offset `int=0` low_memory `bool=False` use_statistics `bool=True` rechunk `bool=True`	`LazyFrame`	hamilton.plugins.polars_l
parquet	spark `SparkSession` path `str`	`DataFrame`	hamilton.plugins.spark_e
sql		`DataFrame`	hamilton.plugins.pandas_

key	loader params	types	module
	query_or_table `str` db_connection `Union` chunksize `Optional=None` coerce_float `bool=True` columns `Optional=None` dtype `Union=None` dtype_backend `Optional=None` index_col `Union=None` params `Union=None` parse_dates `Union=None`		
xml	path_or_buffer `Union` xpath `Optional=./*` namespace `Optional=None` elems_only `Optional=False` attrs_only `Optional=False` names `Optional=None` dtype `Optional=None` converters `Optional=None` parse_dates `Union=False` encoding `Optional=utf-8` parser `str=lxml` stylesheet `Union=None` iterparse `Optional=None` compression `Union=infer` storage_options `Optional=None` dtype_backend `str=numpy_nullable`	`DataFrame`	hamilton.plugins.pandas_
html	io `Union` match `Optional=.+` flavor `Union=None` header `Union=None` index_col `Union=None` skiprows `Union=None` attrs `Optional=None` parse_dates `Optional=None` thousands `Optional=,` encoding `Optional=None` decimal `str=.` converters `Optional=None` na_values `Iterable=None` keep_default_na `bool=True` displayed_only `bool=True` extract_links `Optional=None` dtype_backend `Literal=numpy_nullable` storage_options `Optional=None`	`DataFrame`	hamilton.plugins.pandas_
stata		`DataFrame`	hamilton.plugins.pandas_

key	loader params	types	module
	filepath_or_buffer `Union` convert_dates `bool=True` convert_categoricals `bool=True` index_col `Optional=None` convert_missing `bool=False` preserve_dtypes `bool=True` columns `Optional=None` order_categoricals `bool=True` chunksize `Optional=None` iterator `bool=False` compression `Union=infer` storage_options `Optional=None`		
feather	path `Union` columns `Optional=None` use_threads `bool=True` storage_options `Optional=None` dtype_backend `Literal=numpy_nullable`	`DataFrame`	hamilton.plugins.pandas_
feather	source `Union` columns `Union=None` n_rows `Optional=None` use_pyarrow `bool=False` memory_map `bool=True` storage_options `Optional=None` row_count_name `Optional=None` row_count_offset `int=0` rechunk `bool=True`	`DataFrame`	hamilton.plugins.polars_
feather	source `Union` columns `Union=None` n_rows `Optional=None` use_pyarrow `bool=False` memory_map `bool=True` storage_options `Optional=None` row_count_name `Optional=None` row_count_offset `int=0` rechunk `bool=True`	`LazyFrame`	hamilton.plugins.polars_l
orc	path `Union` columns `Optional=None` dtype_backend `Literal=numpy_nullable` filesystem `Union=None`	`DataFrame`	hamilton.plugins.pandas_
excel			hamilton.plugins.pandas_

key	loader params	types	module
	path `Union=None` sheet_name `Union=0` header `Union=0` names `Optional=None` index_col `Union=None` usecols `Union=None` dtype `Union=None` engine `Optional=None` converters `Union=None` true_values `Optional=None` false_values `Optional=None` skiprows `Union=None` nrows `Optional=None` keep_default_na `bool=True` na_filter `bool=True` verbose `bool=False` parse_dates `Union=False` date_format `Union=None` thousands `Optional=None` decimal `str=.` comment `Optional=None` skipfooter `int=0` storage_options `Optional=None` dtype_backend `Literal=numpy_nullable` engine_kwargs `Optional=None`	`DataFrame`	
table		`DataFrame`	hamilton.plugins.pandas_

key	loader params	types	module
	filepath_or_buffer `Union` sep `Optional=None` delimiter `Optional=None` header `Union=infer` names `Optional=None` index_col `Union=None` usecols `Optional=None` dtype `Union=None` engine `Optional=None` converters `Optional=None` true_values `Optional=None` false_values `Optional=None` skipinitialspace `bool=False` skiprows `Union=None` skipfooter `int=0` nrows `Optional=None` na_values `Union=None` keep_default_na `bool=True` na_filter `bool=True` verbose `bool=False` skip_blank_lines `bool=True` parse_dates `Union=False` infer_datetime_format `bool=False` keep_date_col `bool=False` date_parser `Optional=None` date_format `Optional=None` dayfirst `bool=False` cache_dates `bool=True` iterator `bool=False` chunksize `Optional=None` compression `Union=infer` thousands `Optional=None` decimal `str=.` lineterminator `Optional=None` quotechar `Optional="` quoting `int=0` doublequote `bool=True` escapechar `Optional=None` comment `Optional=None` encoding `Optional=None` encoding_errors `Optional=strict` dialect `Optional=None` on_bad_lines `Union=error` delim_whitespace `bool=False` low_memory `bool=True` memory_map `bool=False` float_precision `Optional=None` storage_options `Optional=None` dtype_backend `Literal=numpy_nullable`		
fwf	filepath_or_buffer `Union` colspecs `Union=infer` widths `Optional=None` infer_nrows `int=100` dtype_backend `Literal=numpy_nullable`	`DataFrame`	hamilton.plugins.pandas_

key	loader params	types	module
spss	path `Union` usecols `Union=None` convert_categoricals `bool=True` dtype_backend `Literal=numpy_nullable`	`DataFrame`	hamilton.plugins.pandas_
avro	file `Union` columns `Union=None` n_rows `Optional=None`	`DataFrame`	hamilton.plugins.polars_
ndjson		`DataFrame`	hamilton.plugins.polars_

key	loader params	types	module
	source `Union` schema `collections.abc.Mapping[str, typing.Union[ForwardRef('DataTypeClass'), ForwardRef('DataType'), type[int], type[float], type[bool], type[str], type['date'], type['time'], type['datetime'], type['timedelta'], type[list[typing.Any]], type[tuple[typing.Any, ...]], type[bytes], type[object], type['Decimal'], type[None], NoneType]]	collections.abc.Sequence[str	tuple[str, typing.Union[ForwardRef('DataTypeClass'), ForwardRef('DataType'), type[int], type[float], type[bool], type[str], type['date'], type['time'], type['datetime'], type['timedelta'], type[list[typing.Any]], type[tuple[typing.Any, ...]], type[bytes], type[object], type['Decimal'], type[None], NoneType]]=None` schema_overrides `collections.abc.Mapping[str, typing.Union[ForwardRef('DataTypeClass'), ForwardRef('DataType'), type[int], type[float], type[bool], type[str], type['date'], type['time'], type['datetime'], type['timedelta'], type[list[typing.Any]], type[tuple[typing.Any, ...]], type[bytes], type[object], type['Decimal'], type[None], NoneType]]	collections.abc.Sequence[str	tuple[str, typing.Union[ForwardRef('DataTypeClass'), ForwardRef('DataType'), type[int], type[float], type[bool], type[str], type['date'], type['time'], type['datetime'], type['timedelta'], type[list[typing.Any]], type[tuple[typing.Any, ...]], type[bytes], type[object], type['Decimal'], type[None], NoneType]]=None`		
database		`DataFrame`	hamilton.plugins.polars_

key	loader params	types	module
	query `str` connection `Union` iter_batches `bool=False` batch_size `Optional=None` schema_overrides `Optional=None` infer_schema_length `Optional=None` execute_options `Optional=None`		
spreadsheet	source `Union` sheet_id `Union=None` sheet_name `Union=None` engine `Literal=xlsx2csv` engine_options `Optional=None` read_options `Optional=None` schema_overrides `Optional=None` raise_if_empty `bool=True`	`DataFrame`	hamilton.plugins.polars_
dlt	resource `DltResource`	`DataFrame`	hamilton.plugins.dlt_exte
mlflow	model_uri `Optional=None` mode `Literal=tracking` run_id `Optional=None` path `Union=model` model_name `Optional=None` version `Union=None` version_alias `Optional=None` flavor `Union=None` mlflow_kwargs `Dict=None`	`Any`	hamilton.plugins.mlflow_

Data Savers

key	saver params	types	module
json	path `str`	`dict list`	hamilton.io.default_data_loaders
json		`DataFrame`	hamilton.plugins.pandas_extensio

key	saver params	types	module
	filepath_or_buffer `Union` compression `str=infer` date_format `str=epoch` date_unit `str=ms` default_handler `Optional=None` double_precision `int=10` force_ascii `bool=True` index `Optional=None` indent `int=0` lines `bool=False` mode `str=w` orient `Optional=None` storage_options `Optional=None`		
json	file `Union`	`DataFrame` `LazyFrame`	hamilton.plugins.polars_post_1_0_
json	path `Union`	`XGBModel` `Booster`	hamilton.plugins.xgboost_extensio
file	path `str` encoding `str=utf-8`	`str`	hamilton.io.default_data_loaders
file	path `Union`	`bytes` `BytesIO`	hamilton.io.default_data_loaders
file		`LGBMModel` `Booster` `CVBooster`	hamilton.plugins.lightgbm_extensi

key	saver params	types	module
	path `Union` num_iteration `Optional=None` start_iteration `int=0` importance_type `Literal=split`		
pickle	path `str`	`object`	hamilton.io.default_data_loaders
pickle	path `Union` compression `Union=infer` protocol `int=5` storage_options `Optional=None`	`DataFrame`	hamilton.plugins.pandas_extensio
memory		`Any`	hamilton.io.default_data_loaders
yaml	path `Union`	`str` `int` `float` `bool` `dict` `list`	hamilton.plugins.yaml_extensions
plt		`Figure`	hamilton.plugins.matplotlib_exten

key	saver params	types	module
	path `Union` dpi `Union=None` format `Optional=None` metadata `Optional=None` bbox_inches `Union=None` pad_inches `Union=None` facecolor `Union=None` edgecolor `Union=None` backend `Optional=None` orientation `Optional=None` papertype `Optional=None` transparent `Optional=None` bbox_extra_artists `Optional=None` pil_kwargs `Optional=None`		
npy	path `Union` allow_pickle `Optional=None` fix_imports `Optional=None`	`ndarray`	hamilton.plugins.numpy_extensio
csv		`DataFrame`	hamilton.plugins.pandas_extensio

key	saver params	types	module
	path `Union` sep `Optional=,` na_rep `str=` float_format `Union=None` columns `Optional=None` header `Union=True` index `Optional=False` index_label `Union=None` mode `str=w` encoding `Optional=None` compression `Union=infer` quoting `Optional=None` quotechar `Optional="` lineterminator `Optional=None` chunksize `Optional=None` date_format `Optional=None` doublequote `bool=True` escapechar `Optional=None` decimal `str=.` errors `str=strict` storage_options `Optional=None`		
csv		`DataFrame` `LazyFrame`	hamilton.plugins.polars_post_1_0_

key	saver params	types	module
	file `Union` include_header `bool=True` separator `str=,` line_terminator `str=` quote_char `str="` batch_size `int=1024` datetime_format `str=None` date_format `str=None` time_format `str=None` float_precision `int=None` null_value `str=None` quote_style `Type=None`		
parquet	path `Union` engine `Literal=auto` compression `Optional=snappy` index `Optional=None` partition_cols `Optional=None` storage_options `Optional=None` extra_kwargs `Optional=None`	`DataFrame`	hamilton.plugins.pandas_extensio
parquet		`DataFrame` `LazyFrame`	hamilton.plugins.polars_post_1_0_

key	saver params	types	module
	file `Union` compression `Any=zstd` compression_level `int=None` statistics `bool=False` row_group_size `int=None` use_pyarrow `bool=False` pyarrow_options `Dict=None`		
sql	table_name `str` db_connection `Any` chunksize `Optional=None` dtype `Union=None` if_exists `str=fail` index `bool=True` index_label `Union=None` method `Union=None` schema `Optional=None`	`DataFrame`	hamilton.plugins.pandas_extensio
xml		`DataFrame`	hamilton.plugins.pandas_extensio

key	saver params	types	module
	path_or_buffer `Union` index `bool=True` root_name `str=data` row_name `str=row` na_rep `Optional=None` attr_cols `Optional=None` elems_cols `Optional=None` namespaces `Optional=None` prefix `Optional=None` encoding `str=utf-8` xml_declaration `bool=True` pretty_print `bool=True` parser `str=lxml` stylesheet `Union=None` compression `Union=infer` storage_options `Optional=None`		
html		`DataFrame`	hamilton.plugins.pandas_extensio

key	saver params	types	module
	buf `Union=None` columns `Optional=None` col_space `Union=None` header `Optional=True` index `Optional=True` na_rep `Optional=NaN` formatters `Union=None` float_format `Optional=None` sparsify `Optional=True` index_names `Optional=True` justify `str=None` max_rows `Optional=None` max_cols `Optional=None` show_dimensions `bool=False` decimal `str=.` bold_rows `bool=True` classes `Union=None` escape `Optional=True` notebook `Literal=False` border `int=None` table_id `Optional=None` render_links `bool=False` encoding `Optional=utf-8`		

key	saver params	types	module
stata	path `Union=None` convert_dates `Optional=None` write_index `bool=True` byteorder `Optional=None` time_stamp `Optional=None` data_label `Optional=None` variable_labels `Optional=None` version `Literal=114` convert_strl `Optional=None` compression `Union=infer` storage_options `Optional=None` value_labels `Optional=None`	`DataFrame`	hamilton.plugins.pandas_extensio
feather	path `Union` dest `Optional=None` compression `Literal=None` compression_level `Optional=None` chunksize `Optional=None` version `Optional=2`	`DataFrame`	hamilton.plugins.pandas_extensio

key	saver params	types	module
feather	file `Union=None` compression `Type=uncompressed`	`DataFrame` `LazyFrame`	hamilton.plugins.polars_post_1_0_
orc	path `Union` engine `Literal=pyarrow` index `Optional=None` engine_kwargs `Optional=None`	`DataFrame`	hamilton.plugins.pandas_extensio
excel		`DataFrame`	hamilton.plugins.pandas_extensio

key	saver params	types	module
	path `Union` sheet_name `str=Sheet1` na_rep `str=` float_format `Optional=None` columns `Optional=None` header `Union=True` index `bool=True` index_label `Union=None` startrow `int=0` startcol `int=0` engine `Optional=None` merge_cells `bool=True` inf_rep `str=inf` freeze_panes `Optional=None` storage_options `Optional=None` engine_kwargs `Optional=None` mode `Optional=w` if_sheet_exists `Optional=None` datetime_format `str=None` date_format `str=None`		
avro	file `Union` compression `Any=uncompressed`	`DataFrame` `LazyFrame`	hamilton.plugins.polars_post_1_0_
ndjson	file `Union`	`DataFrame` `LazyFrame`	hamilton.plugins.polars_post_1_0_

key	saver params	types	module
database	table_name `str` connection `Union` if_table_exists `Literal=fail` engine `Literal=sqlalchemy`	`DataFrame` `LazyFrame`	hamilton.plugins.polars_post_1_0_
spreadsheet		`DataFrame` `LazyFrame`	hamilton.plugins.polars_post_1_0_

workbook `Union`
worksheet
`Optional=None`
position `Union=A1`
table_style
`Union=None`
table_name
`Optional=None`
column_formats
`Optional=None`
dtype_formats
`Optional=None`
conditional_formats
`Optional=None`
header_format
`Optional=None`
column_totals
`Union=None`
column_widths
`Union=None`
row_totals
`Union=None`
row_heights
`Union=None`
sparklines
`Optional=None`
formulas
`Optional=None`
float_precision
`int=3`
include_header
`bool=True` autofilter
`bool=True` autofit
`bool=False`
hidden_columns
`Union=None`
hide_gridlines
`bool=None`
sheet_zoom
`Optional=None`
freeze_panes
`Union=None`

key	saver params	types	module
png	path `Union` dpi `float=200` format `str=png` metadata `Optional=None` bbox_inches `str=None` pad_inches `float=0.1` backend `Optional=None` papertype `str=None` transparent `bool=None` bbox_extra_artists `Optional=None` pil_kwargs `Optional=None`	`ConfusionMatrixDisplay` `DetCurveDisplay` `PrecisionRecallDisplay` `PredictionErrorDisplay` `RocCurveDisplay` `DecisionBoundaryDisplay` `LearningCurveDisplay` `PartialDependenceDisplay` `ValidationCurveDisplay` `Figure`	hamilton.plugins.sklearn_plot_ext
dlt	pipeline `Pipeline` table_name `str` primary_key `Optional=None` write_disposition `Optional=None` columns `Optional=None` schema `Optional=None` loader_file_format `Optional=None`	`Iterable` `DataFrame` `Table` `RecordBatch`	hamilton.plugins.dlt_extensions
mlflow		`Any`	hamilton.plugins.mlflow_extensio

key	saver params	types	module
	path `Union=model` register_as `Optional=None` flavor `Union=None` run_id `Optional=None` mlflow_kwargs `Dict=None`		

Data Adapters

Reference for data adapter base classes:

class hamilton.io.data_adapters.DataLoader

Base class for data loaders. Data loaders are used to load data from a data source. Note that they are inherently polymorphic – they declare what type(s) they can load to, and may choose to load differently depending on the type they are loading to.

abstractmethod classmethod applicable_types() → Collection[Type]

Returns the types that this data loader can load to. These will be checked against the desired type to determine whether this is a suitable loader for that type.

Note that a loader can load to multiple types. This is the function to override if you want to add a new type to a data loader.

Note if you have any specific requirements for loading types (generic/whatnot), you can override applies_to as well, but it will make it much harder to document/determine what is happening.

Returns:

classmethod applies_to(*type_: Type[Type]*) → bool

Tells whether or not this data loader can load to a specific type. For instance, a CSV data loader might be able to load to a dataframe, a json, but not an integer.

I.e. is the adapter type a subclass of the passed in type?

This is a classmethod as it will be easier to validate, and we have to construct this, delayed, with a factory.

Parameters:

type – Candidate type

Returns:

True if this data loader can load to the type, False otherwise.

classmethod can_load() → bool

Returns whether this adapter can "load" data. Subclasses are meant to implement this function to tell the framework what to do with them.

Returns:

classmethod can_save() → bool

Returns whether this adapter can "save" data. Subclasses are meant to implement this function to tell the framework what to do with them.

Returns:

classmethod get_optional_arguments() → Dict[str, Type[Type]]

Gives the optional arguments for the class. Note that this just uses the type hints from the dataclass.

Returns:

The optional arguments for the class.

classmethod get_required_arguments() → Dict[str, Type[Type]]

Gives the required arguments for the class. Note that this just uses the type hints from the dataclass.

Returns:

The required arguments for the class.

abstractmethod load_data(*type_: Type[Type]*) → Tuple[Type, Dict[str, Any]]

Loads the data from the data source. Note this uses the constructor parameters to determine how to load the data.

Returns:

The type specified

abstractmethod classmethod name() → str

Returns the name of the data loader. This is used to register the data loader with the load_from decorator.

Returns:

The name of the data loader.

class hamilton.io.data_adapters.DataSaver

Base class for data savers. Data savers are used to save data to a data source. Note that they are inherently polymorphic – they declare what type(s) they can save from, and may choose to save differently depending on the type they are saving from.

abstractmethod classmethod applicable_types() → Collection[Type]

Returns the types that this data loader can load to. These will be checked against the desired type to determine whether this is a suitable loader for that type.

Note that a loader can load to multiple types. This is the function to override if you want to add a new type to a data loader.

Note if you have any specific requirements for loading types (generic/whatnot), you can override applies_to as well, but it will make it much harder to document/determine what is happening.

Returns:

classmethod applies_to(*type_: Type[Type]*) → bool

Tells whether or not this data saver can ingest a specific type to save it.

I.e. is the adapter type a superclass of the passed in type?

This is a classmethod as it will be easier to validate, and we have to construct this, delayed, with a factory.

Parameters:

type – Candidate type

Returns:

True if this data saver can handle to the type, False otherwise.

classmethod can_load() → bool

Returns whether this adapter can “load” data. Subclasses are meant to implement this function to tell the framework what to do with them.

Returns:

classmethod can_save() → bool

Returns whether this adapter can “save” data. Subclasses are meant to implement this function to tell the framework what to do with them.

Returns:

classmethod get_optional_arguments() → Dict[str, Type[Type]]

Gives the optional arguments for the class. Note that this just uses the type hints from the dataclass.

Returns:

The optional arguments for the class.

classmethod get_required_arguments() → Dict[str, Type[Type]]

Gives the required arguments for the class. Note that this just uses the type hints from the dataclass.

Returns:

The required arguments for the class.

abstractmethod classmethod name() → str

Returns the name of the data loader. This is used to register the data loader with the load_from decorator.

Returns:

The name of the data loader.

abstractmethod save_data(*data: Any*) → Dict[str, Any]

Saves the data to the data source.

Note this uses the constructor parameters to determine how to save the data.

Returns:

Any relevant metadata. This is up the the data saver, but will likely include the URI, etc... This is going to be similar to the metadata returned by the data loader in the loading tuple.

class hamilton.io.data_adapters.AdapterCommon

abstractmethod classmethod applicable_types() → Collection[Type]

Returns the types that this data loader can load to. These will be checked against the desired type to determine whether this is a suitable loader for that type.

Note that a loader can load to multiple types. This is the function to override if you want to add a new type to a data loader.

Note if you have any specific requirements for loading types (generic/whatnot), you can override applies_to as well, but it will make it much harder to document/determine what is happening.

Returns:

abstractmethod classmethod applies_to(*type_: Type[Type]*) → bool

Tells whether or not this adapter applies to the given type.

Note: you need to understand the edge direction to properly determine applicability. For loading data, the loader type needs to be a subclass of the type being loaded into. For saving data, the saver type needs to be a superclass of the type being passed in.

This is a classmethod as it will be easier to validate, and we have to construct this, delayed, with a factory.

Parameters:

type – Candidate type

Returns:

True if this adapter can be used with that type, False otherwise.

classmethod can_load() → bool

Returns whether this adapter can "load" data. Subclasses are meant to implement this function to tell the framework what to do with them.

Returns:

classmethod can_save() → bool

Returns whether this adapter can “save” data. Subclasses are meant to implement this function to tell the framework what to do with them.

Returns:

classmethod get_optional_arguments() → Dict[str, Type[Type]]

Gives the optional arguments for the class. Note that this just uses the type hints from the dataclass.

Returns:

The optional arguments for the class.

classmethod get_required_arguments() → Dict[str, Type[Type]]

Gives the required arguments for the class. Note that this just uses the type hints from the dataclass.

Returns:

The required arguments for the class.

abstractmethod classmethod name() → str

Returns the name of the data loader. This is used to register the data loader with the load_from decorator.

Returns:

The name of the data loader.

Dataflows

Here lies reference documentation for *dataflows* module functions that enable you to discover and use community-contributed dataflows. See the ecosystem page for available dataflow resources.

Reference

clear_storage()

hamilton.dataflows.clear_storage()

Clears all the data under DATAFLOW_FOLDER. By default its "~/.hamilton/dataflows".

copy()

hamilton.dataflows.copy(*dataflow: ModuleType*, *destination_path: str*, *overwrite: bool = False*, *renamed_module: str = None*)

Copies a dataflow module to the passed in path.

```
from hamilton import dataflows

# dynamically pull and then copy
NAME_OF_DATAFLOW = dataflow.import_module("NAME_OF_DATAFLOW",
"NAME_OF_USER")
dataflow.copy(NAME_OF_DATAFLOW,
destination_path="PATH_TO_DIRECTORY")
# copy from the installed library
from hamilton.contrib.user.NAME_OF_USER import NAME_OF_DATAFLOW

dataflow.copy(NAME_OF_DATAFLOW,
destination_path="PATH_TO_DIRECTORY")
```

Parameters:

- **dataflow** – the module to copy.

- **destination_path** – the path to a directory to place the module in.
- **overwrite** – whether to overwrite the destination. Default is False and raise an error.
- **renamed_module** – whether to rename the copied module. Default is None and will use the original name.

find()

hamilton.dataflows.find(*query: str*, *version: str = None*, *user: str = None*)

Searches for locally downloaded dataflows based on a query string.

Parameters:

- **query** – key words to search for.
- **version** – the version to inspect. "latest" will resolve to the most recent commit, else pass a commit SHA.
- **user** – the github name of the user.

Returns:

list of tuples of (version, user, dataflow)

import_module()

hamilton.dataflows.import_module(*dataflow: str*, *user: str = None*, *version: str = 'latest'*, *overwrite: bool = False*) → ModuleType

Pulls & imports dataflow code from github and returns a module.

```
from hamilton import dataflows, driver
# downloads into ~/.hamilton/dataflows and loads the module --
WARNING: ensure you know what code you're importing!
# NAME_OF_DATAFLOW = dataflow.import_module("NAME_OF_DATAFLOW") #
if using official dataflow
NAME_OF_DATAFLOW = dataflow.import_module("NAME_OF_DATAFLOW",
"NAME_OF_USER")
```

```
dr = (
  driver.Builder()
  .with_config({})  # replace with configuration as appropriate
  .with_modules(NAME_OF_DATAFLOW)
  .build()
)
# execute the dataflow, specifying what you want back. Will
return a dictionary.
result = dr.execute(
  [NAME_OF_DATAFLOW.FUNCTION_NAME, ...],  # this specifies what
you want back
  inputs={...}  # pass in inputs as appropriate
)
```

Parameters:

- **dataflow** – the name of the dataflow.
- **user** – Optional. If none it assumes official.
- **version** – the version to get. “latest” will resolve to the most recent commit. Otherwise pass a the commit SHA you want to pull.
- **overwrite** – whether to overwrite the local path. Default is False.

Returns:

a Module that you can then pass to Hamilton.

inspect()

Use this to get cursory information about a Apache Hamilton module.

class hamilton.dataflows.InspectResult(*version*, *user*, *dataflow*, *python_dependencies*, *configurations*)

hamilton.dataflows.inspect(*dataflow: str*, *user: str = None*, *version: str = 'latest'*) → InspectResult

Inspects a dataflow for information.

This is a helper function to get information about a dataflow that exists locally. It does not get more information because we don’t want to assume we can import the module.

```
from hamilton import dataflows

info = dataflows.inspect("text_summarization", "zilto")
```

Parameters:

- **dataflow** – the dataflow name.
- **user** – the github name of the user. None for DAGWorks official.
- **version** – the version to inspect. “latest” will resolve to the most recent commit, else pass a commit SHA.

Returns:

hamilton.dataflow.InspectResult object that contains version, user URL, dataflow URL, python dependencies, configurations.

inspect_module()

Use this to get deep information about a Apache Hamilton module.

class hamilton.dataflows.InspectModuleResult(*version*, *user*, *dataflow*, *python_dependencies*, *configurations*, *possible_inputs*, *nodes*, *designated_outputs*)

hamilton.dataflows.inspect_module(*module: ModuleType*) → InspectModuleResult

Inspects the import module for information.

This does more than *inspect* because the module has been loaded and thus we can put it into a Hamilton driver and ask questions of it.

```
from hamilton.contrib.user.zilto import text_summarization
from hamilton import dataflows

info = dataflows.inspect_module(text_summarization)
```

Parameters:

module – the module with Hamilton code to deeply introspect.

Returns:

hamilton.dataflow.InspectModuleResult object.

install_dependencies_string()

hamilton.dataflows.install_dependencies_string(*dataflow: str*, *user: str = None*, *version: str = 'latest'*) → str

Returns a string for the user to install dependencies.

Parameters:

- **dataflow** – the name of the dataflow.
- **user** – the github name of the user.
- **version** – the version to inspect. “latest” will resolve to the most recent commit, else pass a commit SHA.

Returns:

pip install string to use.

latest_commit()

hamilton.dataflows.latest_commit(*dataflow: str*, *user: str = None*) → str

Determines the latest commit for a dataflow.

This is useful to know if you want to pull the latest version of a dataflow.

Parameters:

- **dataflow** – the string name of the dataflow
- **user** – the name of the user. None if official.

Returns:

the commit sha.

list()

hamilton.dataflows.list(*version: str = 'latest'*, *user: str = None*) → list

Lists dataflows locally downloaded based on commit_ish and user.

Parameters:

- **version** – the version to inspect. “latest” will resolve to the most recent commit, else pass a commit SHA.
- **user** – the github name of the user.

Returns:

list of tuples of (version, user, dataflow)

pull_module()

hamilton.dataflows.pull_module(*dataflow: str*, *user: str = None*, *version: str = 'latest'*, *overwrite: bool = False*)

Pulls a dataflow module.

Saves to hamilton.dataflow.USER_PATH. An import should just work right after doing this.

It performs the following:

1. Creates a URL to pull from github.
2. Pulls the code for the dataflow.
3. Save to the local location based on hamilton.dataflow.USER_PATH.

Parameters:

- **dataflow** – the dataflow name.
- **user** – the user’s github handle.

- **version** – the commit version. “latest” will resolve to the most recent commit, else pass a commit SHA.
- **overwrite** – whether to overwrite. Default is False.

Telemetry

If you do not wish to participate in telemetry capture, one can opt-out with one of the following methods:

1. Set it to false programmatically in your code before creating a Hamilton Driver:

```
from hamilton import telemetry
telemetry.disable_telemetry()
```

2. Set the key `telemetry_enabled` to `false` in ~/.hamilton.conf under the `DEFAULT` section:

```
[DEFAULT]
telemetry_enabled = False
```

3. Set HAMILTON_TELEMETRY_ENABLED=false as an environment variable. Either setting it for your shell session:

```
export HAMILTON_TELEMETRY_ENABLED=false
```

or passing it as part of the run command:

```
HAMILTON_TELEMETRY_ENABLED=false python NAME_OF_MY_DRIVER.py
```

ASF

Apache Software Foundation links.

- Foundation
- License
- Events
- Privacy
- Security
- Sponsorship
- Thanks
- Code of Conduct

Mailing Lists

Apache Hamilton uses mailing lists for project discussions, announcements, and community engagement.

Users Mailing List

For general questions, discussions, and user support.

How to Subscribe

Send an empty email to users-subscribe@hamilton.apache.org. Use a subject line like “subscribe” to avoid spam filters. You will receive a confirmation message with instructions to complete the subscription process.

How to Unsubscribe

Send an empty message to users-unsubscribe@hamilton.apache.org from the same email address used to subscribe.

How to Post

Once subscribed, send messages to users@hamilton.apache.org

Archives

View users list archives

Dev Mailing List

For development discussions, design proposals, and contributing to Apache Hamilton.

How to Subscribe

Send an empty email to dev-subscribe@hamilton.apache.org. Use a subject line like “subscribe” to avoid spam filters. You will receive a confirmation message with instructions to complete the subscription process.

How to Unsubscribe

Send an empty message to dev-unsubscribe@hamilton.apache.org from the same email address used to subscribe.

How to Post

Once subscribed, send messages to dev@hamilton.apache.org

Archives

View dev list archives

Created using Sphinx 9.0.4.