# Hamilton: a modular open source declarative paradigm for high level modeling of dataflows

Stefan Krawczyk
skrawczyk@stitchfix.com
stefank@cs.stanford.edu
Stitch Fix
San Francisco, California, USA

Elijah ben Izzy
elijah.benizzy@stitchfix.com
Stitch Fix
San Francisco, California, USA

## ABSTRACT

As the role of data in industry has grown, the need for specific data management tooling has followed. While a hello world example for a typical machine learning workflow might look trivial, once one layers in industry concerns such as data & computational lineage, data quality/observability, scalability, unit testing, code base maintenance and documentation, this melange of specific tooling often results in a poor end to end user experience with high engineering effort. At Stitch Fix, Hamilton was created initially to solve a subset of these concerns for Data Scientists, focusing on simplifying the user experience. However, we have since discovered that the paradigm Hamilton prescribes is conducive to providing a unified interface for a user to describe end to end dataflows, in a way that facilitates modularity of data management system tooling by forcing a clear decoupling of concerns. It does this by requiring a programming paradigm change on part of the user that enables easy specification and execution of dataflow graphs. Hamilton therefore represents a novel high level approach to modeling dataflows, and presents an industry pragmatic avenue for building a simpler user experience that can easily integrate with existing data management tooling in a modular fashion. Hamilton is available as open source code.

## 1 INTRODUCTION

An industry trend that we have lived through at Stitch Fix is the shift to "Full Stack Data Science"[1], where data scientists are expected to not only do data science, but also engineer and manage data pipelines for their production machine learning models. This approach places additional burdens on data scientists, who no longer hand off their ideas off to a software engineering team for implementation and maintenance. Previously, hand-offs allowed data scientists to focus on a specific domain and set of tooling to accomplish their work. They did not have to worry about such production concerns as, lineage, scalability, or data quality. All they had to do was build a model and prescribe the recipe for an engineering team to implement. In a "full stack" model, however, the data scientist has to pick up the engineering work and understand the complexities of implementing a production pipeline. This has made it all the more important to build streamlined experiences that reduce the complexity of their engineering work, while still enabling them to move quickly and adjust their pipelines as the business requires.



At Stitch Fix, the Hamilton framework[5] was conceived to mitigate a data science team's development pain points, with a specific focus on simplifying the user experience and increasing a team's collective ability to collaborate through modular code, while enabling a platform team to change underlying infrastructure with ease. Specifically, Hamilton enables a simple paradigm to create, maintain, and execute code for data transformations, especially in the case of highly complex data transformation dependency chains. Hamilton does this by deriving a directed acyclic graph (DAG) of dependencies from specially defined declarative Python functions that describe the user's intended dataflow. Altogether, Hamilton makes incremental development, code reuse, unit testing, lineage tracking, data quality checks, and code documentation natural and straightforward. Furthermore, its modularity provides avenues to quickly and easily scale computation onto various distributed frameworks, e.g. Ray[4]/Spark[12]/Dask[7], as well as extend the platform to integrate with other data management tools, e.g. lineage/governance and data quality. Hamilton has enabled data science teams at Stitch Fix to scale modeling dataflows to support 4000+ data transformations without impacting team and user productivity.

We will first ground ourselves with a basic extract, transform, load (ETL) approach to machine learning, then explain the requirements that guided Hamilton, and finally spend the rest of this paper diving into Hamilton's programming paradigm. We will show the benefits this paradigm brings, briefly discuss evaluation, propose future extensions, and finish with a summary.

## 2 CURRENT ETL APPROACHES

Bringing a machine learning model to production at Stitch Fix requires building an ETL workflow. One has to extract data (SQL or Python), transform it for input into a model (SQL or Python), transform it into a model (Python), transform data with the help of the model (Python), and finally load the results somewhere to connect it back with the business (SQL or Python). Furthermore, this has to be run on a cadence. If modeled as discrete steps then data/artifacts have to be materialized between them. An orchestration system, e.g. [6, 11], is responsible for scheduling and executing these steps.

Should a data scientist need their ETL to fulfill operational concerns such as tracking data lineage, capturing metadata for governance, or running data quality checks, they would generally have to integrate these concerns as additional steps in a workflow, or directly add them into each step. In general this means the experience of building an ETL is highly coupled with the data management tooling used.

At Stitch Fix, we saw the following problems emerge firsthand:

- Within each step, the code was likely to be poorly documented and have very low unit test coverage. This meant that maintenance and understanding of ETLs over time became harder which negatively impacted team productivity.
- The transformation logic in each step was coupled to materialization. This limited the reuse of this logic and the flexibility to change what a step in an ETL represents. This impacts a team's velocity to iterate and create new ETLs.
- It was hard for a platform team to change underlying infrastructure without requiring a large effort on part on the data science teams to refactor/rewrite their code.

## 3 HAMILTON REQUIREMENTS

The aforementioned problems at Stitch Fix were particularly acute for one data science team that largely worked on time-series forecasting model ETLs. They were tasked with managing thousands of data transforms that were used to generate inputs for many time-series models, that then forecasted the business. With this scale in mind, the following requirements drove the design of Hamilton:

- Unit testing should be easy to implement and not an afterthought.
- It should be natural to document ones work, and keep that documentation up to date.
- Creation & modification of dataflows should be painless.
- A change in underlying data management infrastructure should not overburden a data science team to adopt/migrate to it.

## 4 HAMILTON FRAMEWORK

The Hamilton framework achieves these requirements through three distinct concepts:

- *Hamilton functions*: the low-level unit of work users use to encode dataflow components.
- *Function DAG*: The representation of the dataflow's dependency structure, built by combining function definitions.
- *Driver code*: the code used to generate an output by specifying the functions used to build the DAG, the inputs to execution, and the parts of the DAG to run.

### 4.1 Hamilton Functions

Hamilton functions force a novel programming paradigm on the user. Like regular Python functions, they encapsulate computational logic. However, the user is not responsible for invoking functions and assigning the results to a variable. Instead, this is encoded in the structure of the function itself in a declarative manner. The *function name* serves to specify, or declare, the intended output variable, and the *function input parameters* (as well as their type-annotations) map to expected input variables, i.e. declared dependencies. In the context of creating a dataframe, the function name serves as the *intended output column name*, and the function input parameters serve as the *expected input columns/values*. Type annotations on the function and the variables are required by the Hamilton Framework to enable simple type checking and to aid in code readability.

Note (1), Hamilton can be used to model any python object creation. E.g. extracting data from a database, creating numpy matrices, fitting scikit-learn models, building custom Python objects, etc. More typically, one models a complex data transformation process (i.e. feature engineering) for a machine learning workflow, along with, optionally, the entire machine learning workflow itself. Note (2), Hamilton assumes user python dependencies are homogeneous, though this not a strict requirement.

```
# rather than
df['acquisition_cost'] = df['spend'] / df['signups']

# a user would instead write
def acquisition_cost(
    signups: pd.Series, spend: pd.Series) -> pd.Series:
    """Example showing a simple Hamilton function"""
    return spend / signups
```

**Listing 1: the core Hamilton programming paradigm, using Pandas dataframe column creation as an example.**

Listing 1 shows what the paradigm replaces. How Hamilton processes the defined function is decomposed in Table 1. By defining functions in this manner, the user specifies their intended dataflow. This method of writing Python functions has a variety of implications:

*4.1.1 Tight encapsulation of transform logic.* Hamilton functions push a user to write logic that is decoupled from *how* data is retrieved or saved. This results in a single code block for defining transform logic, and helps ensure that transform logic is re-useable in various contexts.

*4.1.2 Verbosity.* This approach increases the lines of code required for expressing simple operations. However, the benefits outweigh the cost. Inputs are clearly specified, and logic is encapsulated in named functions.

*4.1.3 Unit Testing.* As Hamilton functions contain well encapsulated logic and clearly specify inputs, *all data transform code* is unit testable!

*4.1.4 Code readability and documentation.*

(1) Encapsulating feature logic in functions implies a natural location for documentation (namely the Python docstring).
(2) Coupling the name of the function with a reusable downstream artifact forces more meaningful naming. It is trivial to determine the definition of a feature and locate its usage. One need simply to search the codebase for a function with that name or which has that as an input argument.

*4.1.5 Vector friendly computation.* For creating dataframes, the Hamilton programming paradigm pushes a user to write a function to create a single column, with inputs as columns as well. This then naturally leads the user to write logic that can utilize vector computation, which often speeds up execution.

*4.1.6 Functions as the core interface.* Python functions have well defined boundaries; inputs go in, and one output comes out. They can be serialized, inspected, and executed. Therefore, functions are used as a universal interface and building block for both the user experience and the framework. A user does not need to implement nor understand a special interface to use the core Hamilton features. Similarly, the framework, without knowing the exact shape of the function beforehand, has a clear object to work with, where it

**Table 1: How functions become nodes in a the DAG using the function defined in Listing 1 as an example.**

| | | |
|---|---|---|
| Function Name | `acquisition_cost` | Node name |
| Type-hints | `pd.Series` | Node input & output types |
| Parameter Names | `signups, spend` | Upstream dependencies |
| Documentation | `Example showing a simple Hamilton function` | Node Documentation |
| Function Body | `return spend / signups` | Node Definition |

can wrap the user's functions to inject operational concerns via decorators (see 4.2), or at run time (see 4.3.3).

## 4.2 Advanced Hamilton Functions

In an effort to encapsulate operational concerns and reduce repetitive function logic, Hamilton comes with a variety of decorators. Decorators primarily fulfill one of the following purposes:

(1) Determining whether a function should exist. *if else* blocks are dropped in favor of readable annotations.
(2) Parameterizing function creation. A single function can create multiple DAG nodes.
(3) Simplifying function logic by promoting reuse. Syntactic sugar can help reduce verbosity and repeated code; similarly, an update to business logic need only happen in one place in the code.
(4) Modeling operational concerns in a modular manner. For example, adding metadata for governance purposes, or specifying run time data expectations. This facilitates having a single source of truth for information about a transform.

Hamilton decorators are extensible, and can also be layered to enable highly expressive functions. This a core enabler of data management system modularity. We direct readers to the Hamilton documentation[5] for more information.

## 4.3 The Function DAG

The function DAG is the framework's representation of the nodes that should be executed and the dependencies between them.

*4.3.1 Node Creation.* Hamilton resolves the mapping of functions (written in the above format) to nodes. In the case of Hamilton functions annotated with a decorator(s), a resolution step occurs to determine how many nodes to create (e.g. to add data quality checks), and what the nodes should be named. Functions beginning with _ are presumed to be helper functions and thus excluded from inclusion in the DAG.

*4.3.2 DAG Construction.* Hamilton compiles the DAG from a list of Python modules containing Hamilton functions and optional configuration. It collects the relevant functions to create nodes, determines node dependencies, and assigns edges between them.

*4.3.3 DAG Walking.* Given desired outputs, a topological sorting of the DAG is performed to determine the execution order. As the DAG is walked, additional operational concerns are injected (via graph adapters), i.e. checking input types, delegating computation, and constructing the object returned from execution. See listing 7 in the Appendix for an example of the code required to make Hamilton execute nodes on Ray.

## 4.4 Driver Code

Driver code steers execution of the Function DAG, providing a convenient abstraction layer. Thus the user never has to interact with the DAG itself, and instead utilizes the driver to run and manage their dataflow. This is also where operational settings and concerns are provided for DAG construction. The driver handles the following:

*4.4.1 DAG Instantiation.* The Driver directs construction of the Function DAG. Creation of the driver is as simple as the following:

```
from hamilton import driver
from funcs import spend_forecast, spend_data_loader

config = {...}
modules = [spend_data_loader, spend_forecast]
dr = driver.Driver(config, *modules, adapter=...)
```

**Listing 2: Code to instatiate a Driver & DAG.**

Note that the call to instantiate the driver accepts a *config* argument. This takes the form of a dictionary with string keys and Python objects as values, that serves two purposes: (1) it helps determine the shape of the DAG when coupled with appropriate decorators (section 4.2); (2) it sets inputs that a user wants to be invariant between DAG execution runs. Meanwhile, the *adapter* argument (optional) controls execution (such as delegating to Dask), and determines the object type returned from DAG execution.

*4.4.2 DAG Execution.* The driver has two primary methods:

(1) `execute(outputs_wanted, inputs, overrides)` executes the DAG, computing only what is required to create the output, and returns a python object, e.g. a Pandas dataframe.
(2) `visualize_execution(outputs_wanted, inputs, ...)` visualizes the subsection of the DAG required for execution.

Note that the user can pass parameters to the DAG through two Python dictionaries: `inputs` and `overrides`. `Inputs` specifies run time inputs to the DAG, providing requisite dependencies that are not satisfied by existing nodes. `Overrides` enables the user to bypass execution of specified nodes, effectively short-circuiting their computation. Hamilton will forego computation of any upstream node depended on solely by overridden nodes. By offering these parameterization capabilities, Hamilton enables precise control over the dataflow's structure and execution.

## 4.5 A Hamilton Example

Since Hamilton represents a new paradigm, let us walk through a contrived dataflow that expresses:

(1) extracting some data.
(2) transforming the data to create input features.

(3) fitting a time-series model.
(4) using that time-series model to forecast the future.
(5) employing a few decorators to simplify the code.

We will then show three ways of executing the dataflow to demonstrate modularity:

(1) as a single step to be orchestrated.
(2) as two steps to be orchestrated.
(3) as a single step that delegates to the Ray framework for distributed computation.

```
# in a module, e.g. my_dataflow_functions.py
import datetime
import pandas as pd
from time_series_library import model
from sklearn.metrics import mean_squared_error
from hamilton.function_modifiers import config,
     extract_columns, tag, check_output
import loader

# extract and data transformation functions
@extract_columns('year', 'week', 'spend', 'signups', 'A',
      'B', 'C')
@tag(source='db.table', pii='False', currency='USD',
     owner='team:data_engineering')
def actual_loader(dates: 'some_date_object') -> pd.
     DataFrame:
    """Runs SQL/pulls from a datastore/database."""
    return loader.load_actuals(dates)

@check_output(allow_nans=False)
def weights() -> pd.Series:
    return loader.get_weights()

@config.when(region='UK')
def holidays__uk(year: pd.Series, week: pd.Series) -> pd.
     Series:
    return is_uk_holiday(year, week)

@config.when(region='US')
def holidays__us(year: pd.Series, week: pd.Series) -> pd.
     Series:
    return is_holiday(year, week)

def avg_3wk_spend(spend: pd.Series) -> pd.Series:
    return spend.rolling(3).mean()

def acquisition_cost(spend: pd.Series, signups: pd.Series
     ) -> pd.Series:
    return spend / signups

def spend_shift_3weeks(spend: pd.Series) -> pd.Series:
    return spend.shift(3)

def spend_b(acquisition_cost: pd.Series, B: pd.Series) ->
      pd.Series:
    return acquisition_cost * B

# curates specific dataset for time-series needs
def data_set(spend_b: pd.Series, spend_shift_3weeks: pd.
     Series, ...) -> pd.DataFrame:
    """Abbreviated. Data set for input to the model."""
    return pd.DataFrame([spend_b, spend_shift_3weeks,
     ...]).T

# model training and forecasting functions
def train_X(cut_off: datetime.datetime, features: List[
     str], data_set: pd.DataFrame) -> pd.DataFrame:
    return data_set[data_set.index < cut_off][features]

def train_y(cut_off: datetime.datetime, target_name: str,
      data_set: pd.DataFrame) -> pd.Series:
    return data_set[data_set.index <cut_off][target_name]

def ts_model(train_X: pd.DataFrame, train_y: pd.Series)
     -> model.TSModel:
    return model.fit(train_X, train_y)

def predict_X(cut_off: datetime.datetime, features: List[
     str], data_set: pd.DataFrame) -> pd.DataFrame:
    return data_set[data_set.index >= cut_off][features]

@check_output(allow_nans=False, range=(0.0, 10000.0))
def forecast(ts_model: model.TSModel, predict_X: pd.
     DataFrame) -> pd.Series:
    return ts_model.predict(predict_X)
```

**Listing 3: Shows how Hamilton could model a time-series modeling dataflow. Note: modularity of the dataflow can be achieved by extracting functions into separate Python modules, rather than in a single one as we have here.**

```
from hamilton import base, driver
import my_dataflow_functions
config = {"region": "US", ...}
adapter =base.SimplePythonGraphAdapter(base.DictResult())
dr = driver.Driver(config, my_dataflow_functions,
     adapter=adapter) # augment driver with adapter
results = dr.execute(['ts_model', 'forecast'])
save_blob(results["ts_model"],
          config["model-artifact-location"])
save_df(results["forecast"], config["forecast-location"])
```

**Listing 4: Driver code: shows how to execute the dataflow in a single step.**

```
## Step (1) in your orchestration system
from hamilton import driver
import my_dataflow_functions
config = {"region": "US", ...}
dr = driver.Driver(config, my_dataflow_functions)
df = dr.execute(['year', ..., 'acquisition_cost', ...])
save_df(df, config["train-data-loc"]) # saves data

## Step (2) in your orchestration system. They are
## linked together by the materialized data set
from hamilton import base, driver
import my_dataflow_functions
import loader
config = {"region": "US", ...}
data_set = loader.load_data(config["train-data-loc"])
adapter =base.SimplePythonGraphAdapter(base.DictResult())
dr = driver.Driver(config, my_dataflow_functions,
                   adapter=adapter)
results = dr.execute(['ts_model', 'forecast'],
                     overrides={"data_set": data_set})
save_blob(results["ts_model"],
          config["model-artifact-location"])
save_df(results["forecast"], config["forecast-location"])
```

**Listing 5: Driver code: shows how one would execute the dataflow in two discrete steps, requiring the materialization of output from the first, and passing it back into the second.**

```
from hamilton import base, driver
from hamilton.experimental import h_ray
import my_dataflow_functions
```

```python
import ray

ray.init() # connect/start cluster
config = {"region": "US", ...}
rga = h_ray.RayGraphAdapter( # adapts DAG traversal
    result_builder=base.DictResult())
dr = driver.Driver(config, my_dataflow_functions,
     adapter=rga) # augment driver with adapter
results = dr.execute(['ts_model', 'forecast'])
save_blob(results["ts_model"],
          config["model-artifact-location"])
save_df(results["forecast"], config["forecast-location"])
ray.shutdown() # shutdown connection/ray
```

**Listing 6: Driver code: shows scaling the dataflow onto a system such as Ray by augmenting the driver.**

As can be seen in Listing 3, each data transformation is neatly encapsulated within a function. Some functions have decorators, and it should be straightforward to understand what they are doing. One can attach metadata and data expectations as desired, without cluttering the dataflow definition. See Figure 1 in the Appendix for an example execution rendering that Hamilton created. To add/update the dataflow is straightforward. For example, if one wanted to evaluate the fit time-series model, then one only needs to define a new function, name it, and request the right input arguments.

To execute the dataflow, one just needs to specify what results to compute, and provide the driver the right inputs. For example, to go from a single step (Listing 4), to a two step (Listing 5) workflow, one just needs to change specification of the desired output, and provide the requisite input. This gives the user a lot of flexibility to compose their workflow as desired; no transform logic needs to change.

Similarly, with Listing 6, a user does not have to change their transformation logic to take advantage of scaling out onto Ray; they only need to change a few lines of driver code. The code to add support for a framework like Ray is minimal, see Listing 7 in the Appendix.

## 4.6 Benefits of Hamilton

We have found the following benefits when using Hamilton.

*4.6.1 Incremental Development.* As dataflows are composed of discrete, unit-testable components, modifications to produce new data can be started locally by conducting test-driven development on the function itself. As node execution only requires running upstream dependencies, integrating with the full dataflow is straightforward. The user need only request computation of the new node via the Hamilton driver to integration test the new addition.

*4.6.2 Debugging.* One can isolate bugs methodically by determining the erroneous output, finding the same-name function definition, debugging that logic, and if no error is found, repeat, tracing through each upstream dependency. For example, to debug *spend_b* from our contrived example (listing 3), it is easy to visualize its execution path, e.g. Figure 1 in the Appendix, and thus determine what needs to be debugged.

*4.6.3 Central Definition Store.* To leverage feature engineering work, most industry solutions target materialized data, e.g. [2], rather than the code itself. With Hamilton, module organization is incentivized, and thus curating modules into a single repository makes it straightforward for a team to refer to and reuse work.

*4.6.4 Transparent Scaling.* Most distributed computation frameworks follow a lazy execution model e.g. Dask, Ray, and Spark. They build a DAG of the computation required prior to distributing execution. As Hamilton's function DAG is structured using the same approach, it can provide a layer of indirection between dataflow definition and method of execution. In practice, this means that most Hamilton functions do not need modification to run on these distributed computation systems, unless the data type they operate over is not supported by that system. For example, both Spark and Dask implement the Pandas dataframe API, so a user would not have to change their Pandas code to scale to a Dask or Spark cluster, other than changing how they load data for execution.

*4.6.5 Lineage.* Hamilton unlocks the ability to provide fine grained lineage of computation for *any* workflow, and together with source code version control (e.g. git), a lightweight means to store it. This is important, especially with the growth of privacy concerns and data regulation, as organizations need to know what data comes in, where it goes, and how it is used. To facilitate that, Hamilton functions can be marked with privacy & regulation concerns, e.g. Personally Identifiable Information (PII), so one can surface answers to questions of data usage and data impact by examining the structure of the produced DAG. To store lineage for a materialized artifact, one only needs to snapshot (e.g. commit via git) the source code, driver script, and configuration that created the DAG. To recover the lineage, one need only go back to that snapshot and re-instantiate the DAG to ask questions of it.

*4.6.6 Modular Components.* The function-based nature of Hamilton (see 4.1.6) enables clear decoupling of operational components from user DAG specification (see 4.2, listing 3). It is straightforward to add, replace, and improve such components (data quality or delegating execution for example), while keeping the users' business logic static, because the underlying assumption is that everything is a function. This greatly reduces the friction for platform teams and data scientists to operate independently of one another.

## 4.7 Evaluation Discussion

*4.7.1 Adoption.* To enjoy the benefits of Hamilton, one must use the paradigm. For existing systems, migration has been the largest friction point to adopting Hamilton. Internally, teams with active feature development for time-series forecasting have been the best adopters. Externally (since October 2021), teams using Pandas and wanting to improve software engineering hygiene have been Hamilton's best adopters.

*4.7.2 Quantitative measurement.* Hamilton's focus is on improving the user experience, and not improving execution of a dataflow. So a quantitative assessment of Hamilton's benefits to a team is challenging. One would have to construct a tightly controlled user experiment, however in an industry environment, it's hard to secure resourcing for such an endeavor. That said, anecdotally, for one data science team, a monthly feature engineering task to add and adjust features for model fitting used to take a whole day for a team

member to complete prior to Hamilton. After Hamilton, this task takes no more than two hours, which represents a 4x improvement!

*4.7.3 Qualitative measurement.* The success criteria for the Hamilton project were all qualitative measures. Namely, that a core data science team adopted the tooling, enjoyed using it, and were able to deliver on their business objectives. On all accounts, Hamilton delivered successfully, without any detractors. Since then, two and a half years in production have passed and the same qualitative measures still hold.

## 5 FUTURE EXTENSIONS/AVENUES

Here are some select future directions.

### 5.1 Data Governance Integrations

With Hamilton one can encode a rich repository of metadata (see 4.6.5), however currently, the exposure of this information requires querying the DAG directly. With the ability to provide modular operational functionality, at DAG definition or execution time, Hamilton can integrate with existing governance tooling. For example, run time integrations with tools like open lineage[9] or datahub[10] are possible. Similarly, Hamilton could be used as a conduit to enforce data access policies, since dataflow definition and execution are decoupled.

### 5.2 Compiling to an Orchestration Framework

A common problem with ETL tooling is choosing an orchestration system. This is a big decision, because companies rarely change this infrastructure. Because Hamilton functions do not define or set materialization concerns, it cannot be used in place of an orchestration framework such as Airflow[6], where computation is split into discrete steps and materialized to a data store in between steps. If one were to provide node groupings and a materialization function, then it would be straightforward to compile the Hamilton function DAG onto any existing framework. Programmatically defining orchestration would also unlock the possibility for low cost infrastructure migrations, while also avoiding vendor lock in.

### 5.3 Logically Modeling your Data Warehouse

Common industry data tools (e.g.[8]) and orchestration frameworks leak materialization concerns into the user experience. For example, using SQL, the user has to think in tables. This thinking naturally cascades to how data is materialized and transferred between workflow steps. What if, instead, you could model the dependencies of all your data transforms independently of how and where the data is stored? The declarative nature of Hamilton unlocks this possibility. Having such a fine grained model of dataflows in a data warehouse opens the door to exploring global optimizations for workflow execution and data materialization.

### 5.4 High Performance Computing

With the rise of large machine learning models, GPUs and super computers are getting renewed interest. However, writing code for GPUs or super computers requires specialized knowledge to capitalize on such powerful hardware. Python efforts to make this easier on the user generally revolve around frameworks that make use of decorators[3, 13] to wrap Python code and "compile" it for their target. Because Hamilton naturally forces all logic to be written as functions, injecting these frameworks could be done at DAG creation/walk time. This would enable a quick way to get up and running with such frameworks, or better yet, swap them out trivially without having to change much (if any) transform logic.

## 6 SUMMARY

Hamilton is a novel open source tool that prescribes a declarative, high level paradigm for defining dataflows in python. It enables decoupling transform definitions from execution and materialization which yields many beneficial properties for creating, managing, and executing data or machine learning transformations. It started at Stitch Fix to address a data science team's developer experience, and has since been extended to cover wider concerns with lineage, scalability, portability, and data quality. It creates an opinionated user experience for those writing transformation logic, while providing a platform team enough abstractions to modularize underlying data management systems, resulting in little to no effort from the transformation logic owner to adopt/migrate when changes occur.

The horizon is bright for Hamilton, as the project looks to further expand integration types as well as integration implementations with a wider set of data management tooling. Come join us.

## REFERENCES


[1] Eric Colson. 2019. *Beware the data science pin factory: The power of the full-stack data science generalist and the perils of division of labor through function.* https://multithreaded.stitchfix.com/blog/2019/03/11/FullStackDS-Generalists/

[2] Theofilos Kakantousis, Antonios Kouzoupis, Fabio Buso, Gautier Berthou, Jim Dowling, and Seif Haridi. 2019. Horizontally scalable ml pipelines with a feature store. In *Proc. 2nd SysML Conf., Palo Alto, USA.*

[3] Siu Kwan Lam, Antoine Pitrou, and Stanley Seibert. 2015. Numba: A LLVM-Based Python JIT Compiler. In *Proceedings of the Second Workshop on the LLVM Compiler Infrastructure in HPC* (Austin, Texas) *(LLVM '15).* Association for Computing Machinery, New York, NY, USA, Article 7, 6 pages. https://doi.org/10.1145/2833157.2833162

[4] Philipp Moritz. 2019. *Ray: A Distributed Execution Engine for the Machine Learning Ecosystem.* Ph.D. Dissertation. EECS Department, University of California, Berkeley. http://www2.eecs.berkeley.edu/Pubs/TechRpts/2019/EECS-2019-124.html

[5] Stefan Krawczyk, Elijah ben Izzy, Danielle Quinn. 2021. *A scalable general purpose micro-framework for defining dataflows.* https://github.com/stitchfix/hamilton

[6] Various. 2015. *Apache Airflow.* https://github.com/apache/airflow

[7] Various. 2016. *Dask: Library for dynamic task scheduling.* https://dask.org

[8] Various. 2016. *DBT.* https://github.com/dbt-labs/dbt-core

[9] Various. 2017. *Open Lineage.* https://openlineage.io/

[10] Various. 2020. *Datahub.* https://github.com/datahub-project/datahub

[11] Various. 2020. *Metaflow.* https://github.com/Netflix/metaflow

[12] Matei Zaharia, Reynold S. Xin, Patrick Wendell, Tathagata Das, Michael Armbrust, Ankur Dave, Xiangrui Meng, Josh Rosen, Shivaram Venkataraman, Michael J. Franklin, Ali Ghodsi, Joseph Gonzalez, Scott Shenker, and Ion Stoica. 2016. Apache Spark: a unified engine for big data processing. *Commun. ACM* 59, 11 (2016), 56–65. https://doi.org/10.1145/2934664

[13] Alexandros Nikolaos Ziogas, Timo Schneider, Tal Ben-Nun, Alexandru Calotoiu, Tiziano De Matteis, Johannes de Fine Licht, Luca Lavarini, and Torsten Hoefler. 2021. Productivity, Portability, Performance: Data-Centric Python. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis* (St. Louis, Missouri) *(SC '21).* Association for Computing Machinery, New York, NY, USA, Article 95, 13 pages. https://doi.org/10.1145/3458817.3476176

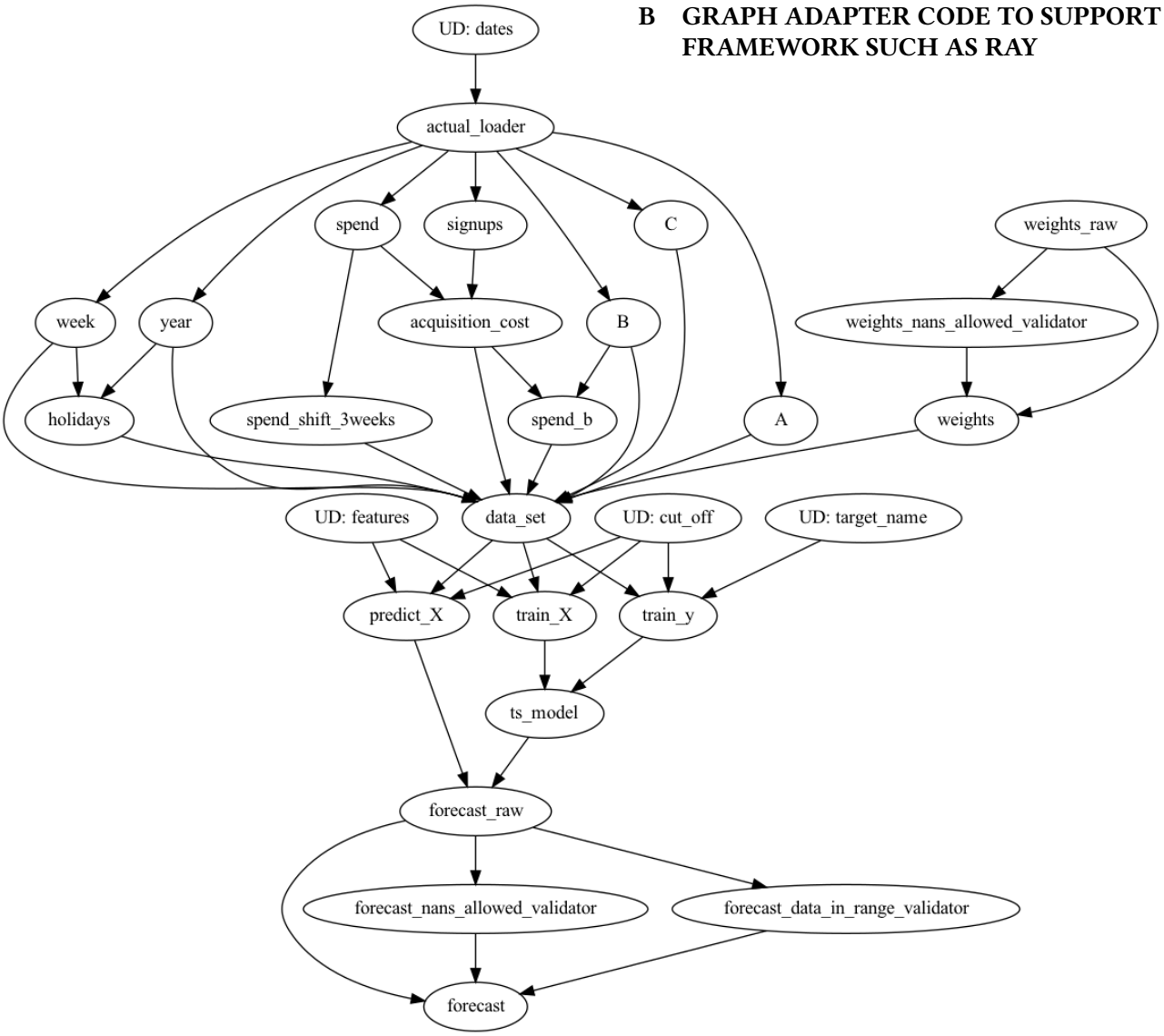


**Figure 1: Example rendering showing what would be executed by running Listing 4, by exercising *visualize_execution()* on the driver. Note: (1) that the check_output decorator produces extra nodes during computation; (2): UD stands for user defined input and would be required to be passed for the DAG to be executed.**

## A EXAMPLE EXECUTION RENDERING

## B GRAPH ADAPTER CODE TO SUPPORT A FRAMEWORK SUCH AS RAY

```python
class RayGraphAdapter(base.HamiltonGraphAdapter, base.ResultMixin):
    """Class representing what's required to make Hamilton run on Ray"""

    def __init__(self, result_builder: base.ResultMixin):
        """Constructor
        :param result_builder: Required. An implementation of base.ResultMixin.
        """
        self.result_builder = result_builder
        if not self.result_builder:
            raise ValueError('Error: ResultMixin object required. Please pass one in for `result_builder`.')

    @staticmethod
    def check_input_type(node_type: typing.Type, input_value: typing.Any) -> bool:
        """Used during DAG execution - user inputs to the DAG are checked against function expected types."""
        # NOTE: the type of a raylet is unknown until they are computed
        if isinstance(input_value, ray._raylet.ObjectRef):
            return True
        return node_type == typing.Any or isinstance(input_value, node_type)

    @staticmethod
    def check_node_type_equivalence(node_type: typing.Type, input_type: typing.Type) -> bool:
        """Used during DAG construction - logic for ensuring types match between functions."""
        return node_type == input_type

    def execute_node(self, node: node.Node, kwargs: typing.Dict[str, typing.Any]) -> typing.Any:
        """Function that is called as we walk the DAG to determine how to execute a Hamilton function.

        :param node: the node from the graph.
        :param kwargs: the arguments that should be passed to it.
        :return: returns a ray object reference.
        """
        return ray.remote(node.callable).remote(**kwargs)

    def build_result(self, **outputs: typing.Dict[str, typing.Any]) -> typing.Any:
        """Builds the result and brings it back to this running process. Analogous to a "reduce" step.

        :param outputs: the dictionary of key -> Union[ray object reference | value]
        :return: The type of object returned by self.result_builder.
        """
        # need to wrap our result builder in a remote call and then pass in what we want to build from.
        remote_combine = ray.remote(self.result_builder.build_result).remote(**outputs)
        result = ray.get(remote_combine)  # this materializes the object locally
        return result
```

**Listing 7: GraphAdapter code: shows what's required to support an integration with the Ray framework. The key functions are execute_node() and build_result(). See function documentation strings for an explanation of each.**