Hamilton: enabling software engineering best practices for data transformations via generalized dataflow graphs

Stefan Krawczyk[1,*], Elijah ben Izzy[1] and Danielle Quinn[1]

[1]Stitch Fix, 1 Montgomery Tower, Suite 1500, 94104, San Francisco, California, USA

Abstract

While data science, as a high level consumer of and producer to data ecosystems, has grown in prevalence within organizations, software engineering practices for data science code bases have not. Stereotypical data science code is not known for unit testing coverage, ease of documentation, reuseability, or enabling quick incremental development as it grows. Over time, this lack of software engineering quality impacts the maintainers ability to make progress within a data ecosystem. The data platform team at Stitch Fix created Hamilton to solve these software engineering pain points with respect to data transformations. It does this by requiring a programming paradigm change that enables straightforward specification and execution of dataflow graphs. Hamilton has enabled data science teams at Stitch Fix to scale their code bases to support 4000+ data transformations, by ensuring that transformation code is always unit testable, documentation friendly, easily curated, reuseable, and amenable to fast incremental development. Hamilton also enables transparently scaling computation onto distributed systems such as Dask, Ray, and Spark, without requiring a rewrite of data transform logic. Hamilton therefore represents a novel approach to modeling dataflows that is decoupled from materialization concerns, and presents an industry pragmatic avenue for building a simpler user experience for high level data ecosystem practitioners. Hamilton is available as open source code.

1. Introduction

With the shift to "Full Stack Data Science"[1], data scientists are expected to not only do data science, but also engineer and manage data pipelines for their production models. This additional responsibility places burdens on data scientists, who no longer hand off their ideas off to a software engineering team for implementation and maintenance. This burden becomes especially acute in the domain of time-series forecasting, where data transformation needs involve creating an ever increasing number of features (columns) in a dataframe (table) for use with model fitting/forecasting. To create better time-series forecasts, one is continually seeking to add more features by incorporating new data, updating existing features, and deriving new features from existing ones. The majority of features are the product of a chain of transformations over other features. At Stitch Fix, the Forecasting, Estimation, and Demand (FED) team had curated a code base over the course of several years, to produce a dataframe for fitting time-series models with thousands of such features. Unfortunately, maintaining and adding features to the code base had become burdensome to the point where their delivery of work slowed significantly. Unit-testing was virtually non-existent, documentation was scattered and inconsistent, and determining feature lineage grew in difficulty with the number of transforms.

The Hamilton framework[2] was therefore conceived to mitigate the FED team's software engineering pain points. Specifically, Hamilton enables a simpler paradigm to create, maintain, and execute code for data engineering, especially in the case of highly complex data transformation dependency chains. Hamilton does this by deriving a directed acyclic graph (DAG) of dependencies using specially defined Python functions that describe the user's intended dataflow. Altogether, Hamilton makes incremental development, code reuse, unit testing, determining lineage, and documentation natural and straightforward. Furthermore, it provides avenues to quickly and easily scale computation onto various distributed computation frameworks, e.g. Ray[3]/Spark[4]/Dask[5], without changing much code.

We will first provide some examples of typical software engineering pain points with data transformations at Stitch Fix, then talk about related tooling, and spend the rest of this report diving into Hamilton's programming paradigm. We will show the benefits this paradigm brings, provide a lightweight evaluation of the framework, and finish with a summary and a description of future work.

2. Software engineering pain points with data transformations

Since software engineering pain points are somewhat subjective, we present the following Python script using Pandas[6] to illustrate common software engineering

Proc. of the First International Workshop on Data Ecosystems (DEco'22), September 5, 2022, Sydney, Australia

*Corresponding author

stefank@cs.stanford.edu (S. Krawczyk);
elijah.benizzy@stitchfix.com (E. b. Izzy);
danielle.quinn@stitchfix.com (D. Quinn)

pain points we encountered at Stitch Fix. It demonstrates creating data transforms that represent features to fit a time-series model.

```
1 # create_features.py
2 import pandas as pd
3 from library import loader, is_holiday, is_uk_holiday
4
5 def compute_bespoke_feature(df: pd.DataFrame) -> pd.
      Series:
6   """Some documentation explaining what this is"""
7   return (df['A'] - df['B'] + df['C']) * loader.
      get_weights()
8
9 def multiply_columns(col1: pd.Series,
10                      col2: pd.Series) -> pd.Series:
11   """Some documentation explaining what this is"""
12   return col1 * col2
13
14 def run(dates, config):
15     df = loader.load_actuals(dates) # e.g. spend,
       signups
16     if config['region'] == 'UK':
17         df['holidays'] = is_uk_holiday(df['year'], df
       ['week'])
18     else:
19         df['holidays'] = is_holiday(df['year'], df['
       week'])
20     df['avg_3wk_spend'] = df['spend'].rolling(3).mean
       ()
21     df['acquisition_cost'] = df['spend'] / df['
       signups']
22     df['spend_shift_3weeks'] = df['spend'].shift(3)
23     df['special_feature1'] = compute_bespoke_feature(
       df)
24     df['spend_b'] = multiply_columns(df['
       acquisition_cost'], df['B'])
25     save_df(df, "some_location")
26 if __name__ == '__main__':
27     run(dates=..., config=...)
```

Listing 1: Example script that loads data, transforms data into features, and saves them

Listing 1 demonstrates the highly heterogenous nature of data transformation code. The `run` function:

1. loads some data into a central dataframe object (line 15).
2. adds and derives features through various means:
   a) inline code that directly alters the dataframe (lines 20-22).
   b) a function that takes the whole dataframe and assigns the result to a new column (lines 5, 23).
   c) a function that uses columns from the central dataframe and assigns the result to a new column (lines 9, 17, 19, 24).
   d) a conditional branch that changes the implementation used to compute a column based on some configuration (lines 16-19).
3. contains only sporadic documentation.
4. relies heavily on code execution order; line 21 has to occur before line 24.

At only twenty-seven lines, the code in Listing 1 looks innocuous. However, scaling this script from six to 1000+ data transforms (as occurred at Stitch Fix) presents the following problems:

2.1. Inconsistent unit test coverage

Only three of the derived features lend themselves towards straightforward unit testing. One cannot unit test the inline dataframe manipulations without running the entire script, so the code base inevitably has minimal, if any, test coverage. In such a codebase, it is difficult to determine behavioral changes when code changes.

2.2. Code readability and documentation

Well organized code with documentation is critical for a maintainer to understand and contribute to a codebase. It ensures information is not siloed in the original developer's mind, and that newcomers to the codebase can quickly become productive. In Listing 1, code readability and documentation is tragically lost between inline manipulations, functions, and the organization of the `run` function. Identifying the logic used to derive a feature is far from trivial, even with the best developer tools.

2.3. Difficulty in tracing data lineage

At six features, tracing lineage of inputs to a data transform is not particularly difficult. At 1000+ data transforms, however, this is a challenging task. At Stitch Fix, there are chains of transformation that span over fourteen such functions, with the average transformation chain length just over five.

In order to add a new data transform, a developer has to make a decision as to where to put it. It could be at the end of the run function, or ideally near some logical grouping of transforms. However there is no forcing function for a developer to do so, which inevitably leads to critical transform code spread throughout the entire codebase. A "spaghetti" codebase like this results in slow and frustrating debug cycles, requiring the cognitive burden of internalizing a mental map of computation in order to identify and fix problems. Ability to debug is then heavily correlated with tenure on the team!

2.4. Integration testing requires calculating all data transforms

While feature generating scripts such as Listing 1 are initially quick to execute, they grow into a large monolith. In order to test the integration of a new feature, one has to run the entire script. As the script inevitably grows

with the increasing complexity of a problem space, it takes longer to run, and thus longer to iterate on, fix bugs, and improve.

2.5. Code Reuse & Duplication

Because transform logic is not well encapsulated, code reuse is difficult to achieve outside of the current context of the script. Good software engineering practices advise consistently refactoring code for reuse, however this is easy to skip. It is simpler for a data scientist to instead find the relevant code and cut & paste it to their new context, especially when they are scarce for time. Left unchecked, this behavior creates more monolithic scripts and propagates the problem.

3. Related tooling

In industry, there are a few tools that come to mind when discussing some of the pain points above.

3.1. Lineage/Data Catalogs

OpenLineage[7] is an framework for data lineage collection and analysis. It aims to provide an open standard to enable disjoint tools to emit lineage metadata that can then be centrally tracked and curated. It requires a oner to implement the standard, as well as maintain infrastructure to collect the emitted lineage metadata. It is designed for tracking materialization of whole data sets. It cannot track lineage at a columnar level.

Data catalogs like Datahub[8] and Amundsen[9], are systems of record with which one can emit and store lineage and other metadata (e.g. for GDPR purposes). They require one to explicitly integrate with their APIs to capture this information. They are only as useful as the information provided to them, so a developer needs to explicitly consider integration as part of their development workflow.

3.2. Data Quality

When one thinks about data transformations and testing data, one often thinks of Pandera[10], Deequ[11], or Great Expectations[12].

Pandera is a stateless lightweight API for performing data validation on Pandas dataframes (i.e. in memory tables). Its focus is to provide a quick mechanism to define expectations in code to create robust data processing pipelines. It has a small python dependency footprint so is easy to install and embed within a pipeline, enabling it to live close to transform logic.

Deequ is a stateful, heavy-weight framework, that requires peripheral services to operate. It is built on top of Apache Spark and aims to define "unit tests for data" that help validate data quality expectations over large datasets. After a dataset has been constructed, the user defines expectations over that data, that are then checked via execution on Apache Spark.

Great Expectations, like Deequ, is also a heavy-weight framework, but is more broadly applicable to python. It allows one to validate, document, and profile data to ensure data quality. It follows a similar implementation pattern to Deequ, as one needs to explicitly integrate it after dataset construction into a dataflow.

None of the frameworks are meant to be run like unit tests, and thus are not designed for testing transform logic.

As for the user experience, one has to explicitly add data quality test(s) into a dataflow. Determining how to add tests, when to add tests, and how to maintain them as dataflows evolve causes extra burden on the dataflow developer. For example, it is possible to change data transform logic and forget to update data quality expectations if they are defined in separate steps of the dataflow, located in a different file in the code base, or stored externally in a datastore. Analogously, if a data quality check fails, it can be similarly difficult to determine what source code generated the data, if one does not link the data quality test appropriately via naming or documentation.

3.3. Orchestration Frameworks

Similar in approach to Hamilton are orchestration frameworks [13, 14, 15, 16]. They too model their operations via a DAG, however their focus is modeling a user's end to end workflow at a macro-level. Specifically, they model discrete steps at each of which an artifact is created and data is materialized. For example, in one step, raw data is ingested and transformed and saved as a table, and in a subsequent step, a machine learning model is trained on that data and that model is saved.

These frameworks also do not try to address any software engineering pain points a data transformation developer might have.

4. Hamilton Framework

The Hamilton framework alleviates the pain points described in Section 2 through three distinct concepts:

- *Hamilton functions*: the low-level unit of work developers 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 execute Hamilton functions by specifying the functions used to

build the DAG, the inputs to execution, and the parts of the DAG to run.

For those eager to see a simple `Hello World` we direct readers to Listing 5 in the Appendix.

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.

Note (1), Hamilton can be used to model any python object creation. For the remainder of this paper, we will stick to the context of creating pandas dataframes. Note (2), if Hamilton functions have wildly different python dependency requirements, using Hamilton is still possible, one would just partition DAG execution into multiple steps matching the different python dependency requirements.

```
# 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 2: the core Hamilton programming paradigm with dataframes

Listing 2 shows an example of the Hamilton paradigm and what it is replacing. Hamilton's breakdown of the example function's components is demonstrated in Table 1. By defining functions in this manner, the developer specifies their intended dataflow. This method of writing Python functions has a variety of implications:

4.1.1. Verbosity

This approach increases the lines of code required to describe simple operations. However, the benefits outweigh the cost. Inputs are clearly specified, and logic is automatically encapsulated in named functions.

4.1.2. Unit Testing

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

4.1.3. 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 needs to simply search the code base for a function with that name or which has that as an input parameter.

4.1.4. Vector friendly computation

In the case of 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 naturally leads the developer to write logic that can utilize vector computation, which often speeds up execution.

4.1.5. 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 with which to work with, where it can wrap a 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 (e.g. `@config` in listing 4).
2. Parameterizing function creation. A single function can create multiple nodes.
3. Simplifying function logic by promoting reuse. Syntactic sugar can help reduce verbosity and repeated code (e.g. `@extract_columns` in listing 4.

Table 1
How functions become nodes in a the DAG using the function defined in Listing 2 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

4. Modeling operational concerns in a modular manner. For example, adding metadata for GDPR purposes, or specifying run time data expectations.

Hamilton decorators are extensible and can also be layered to enable highly expressive functions.

Note, as functions are the core interface (see 4.1.5), the abstraction provided by Hamilton's decorator system enables, a platform team for example, a clear and decoupled way to plug into the user's function writing experience, while providing a clear way to manage and service their decorator implementations. Done correctly, user function definitions remains static to platform changes.

With respect to data ecosystems, we will explain two relevant Hamilton decorators: `@check_output()` and `@tag()`. We direct readers to the Hamilton documentation [2] for more information on other decorators.

4.2.1. `@check_output`

In machine learning (ML) dataflows, data quality issues are a common cause of model problems. It is a best practice to setup data expectations to mitigate these problems. However, as explained in section 3.2, one needs to additionally integrate such a concern into a dataflow explicitly. With Hamilton, integrating data quality expectations are less burdensome, as this takes the form of a lightweight Python decorator `@check_output()`, with which one can simply annotate their Hamilton functions. Doing so enables transform logic and data expectations to be co-located, without cluttering the user's dataflow. There is no need to maintain separate code bases and data stores, or manually integrate checks as an explicit step of a dataflow. Therefore maintenance and operational costs are low for adding runtime data quality checks to a dataflow.

At DAG construction time, Hamilton automatically adds nodes to the DAG to check the output of the decorated function. At run time, after executing the user function, Hamilton validates the provided expectations, surfacing data quality errors to the dataflow developer via logging, or stopping execution altogether if desired. See listing 4 for an example of usage.

4.2.2. `@tag`

As data systems and environments change over time, different metadata needs arise. Rather than requiring explicit integrations with metadata systems, or enforcing a specific schema, Hamilton enables a lightweight way to annotate functions with such concerns. `@tag()` takes in string key value pairs, and is thus amenable to annotating functions with anything relevant to your particular data ecosystem. E.g. ownership, source table names, GDPR concerns, project names, etc. These tags are then attached to nodes in the DAG, which then can be used as a basis for querying for nodes, or asking graph questions of the DAG. See listing 4 for an example of usage.

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 (e.g. listing 2) to nodes. In the case of Hamilton functions annotated with one or more decorators, a resolution step occurs to determine how many nodes to create (e.g. in case of a parameterized function), 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. Constructing the DAG

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. Any dependency that does not map to a known node is marked as a required input for execution.

4.3.3. Walking the DAG

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, e.g. checking inputs and matching against function input

types, delegating function computation, and constructing the final object returned from execution.

4.4. Driver Code

Driver code steers execution of the Function DAG, providing a convenient abstraction layer. Thus the developer never has to interact with the DAG itself, and instead utilizes the driver to run and manage their dataflow. It 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 3: Sample Driver code to instantiate a DAG

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 parts of the DAG required for execution.

Note that the developer can pass parameters to the DAG through two Python dictionaries: `inputs` and `overrides`. `Inputs` specifies runtime inputs to the DAG, providing requisite dependencies that are not satisfied by existing nodes. `Overrides` enables the developer 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 parametrization capabilities, Hamilton enables precise control over the dataflow's structure and execution.

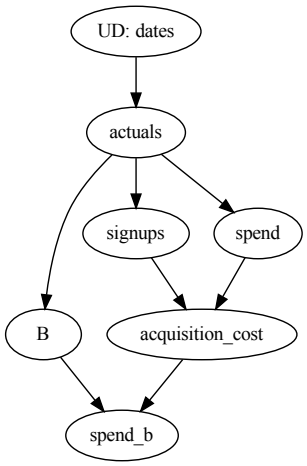

Figure 1: Example rendering produced by running `visualize_execution()` on an instantiated DAG, if one was interested in computing *spend_b* from Listing 1 as implemented in Hamilton in Listing 4. Hamilton makes it straightforward to determine what is required to compute a feature. UD refers to user defined input. Note: for diagram legibility, we omitted displaying the validation nodes that the `@check_output()` decorator would add to the DAG.

```
# in a module, e.g. my_functions.py

@tag(source="prod.denormalized", owner="team:DE")
@extract_columns('year', 'week', 'spend', 'signups',
      'A', 'B', 'C')
def actuals(dates: 'a_date_type') -> pd.DataFrame:
    return loader.load_actuals(dates)

@check_output(data_type=np.float64, 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
```

```python

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

def special_feature1(A: pd.Series, B: pd.Series, C: pd.Series, weights: pd.Series) -> pd.Series:
    """Some documentation explaining what this is"""
    return (A - B + C) * weights

@check_output(data_type=np.float64, range=(0.0, 100.0), allow_nans=False)
def spend_b(acquisition_cost: pd.Series, B: pd.Series) -> pd.Series:
    """documentation to explain this function"""
    return acquisition_cost * B

## In a separate script/module, e.g. run.py,
## code to create and execute the DAG
from hamilton import driver
import my_functions

config = {...} # configuration
modules = [my_functions] # modules to crawl
dr = driver.Driver(config, *modules)
df = dr.execute(['year', 'week', 'holidays', 'acquisition_cost', ...]) # materialize
save_df(df, "some_location") # save result
```

Listing 4: Hamilton version of the earlier example script in Listing 1, with four decorators used to show example usage.

4.5. Benefits of Hamilton

With respect to a data scientist's workflow, we have found the following benefits when using Hamilton.

4.5.1. Incremental Development

Rather than requiring execution of a monolithic script, Hamilton pushes the dataflow creator towards incremental, test-driven, 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 developer need only request computation of the new node via the Hamilton driver to integration test the new addition.

4.5.2. Debugging

Hamilton makes debugging dataflows simpler by providing a standard methodical approach. One can isolate bugs 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. Standard debugging procedures (such as code-diffing, breakpoints, and bisection) gain in value due to Hamilton's logical mapping of code to produced data. For example, to debug *spend_b* from our contrived example (listing 1), it is straightforward to visualize it's execution path, Figure 1, and thus determine what needs to be debugged.

4.5.3. Documentation

The confluence of:

- using function documentation strings
- one-to-one mapping of outputs to functions
- the ability to visualize the DAG and execution paths
- the `@tag()` decorator for adding extra metadata

enables a clear and straightforward means to document transform logic in a standardized way. The function documentation string is perfect for long form explanations, and can be exposed via tooling such as sphinx[17]. The mapping of function names to outputs ensures that function names and input parameters are meaningful while also enabling one to quickly locate the definition of an output. The ability to visualize the DAG and execution paths helps provide a big picture mental model for those learning the code base. The `@tag()` decorator makes it easy to add additional metadata concerns, without cluttering the transform logic itself.

4.5.4. Central Definition Store

A common problem for machine learning practitioners is that of leveraging other's work. Most industry solutions target materialized data, e.g. [18], rather than the code itself. As the code in Hamilton maps directly to outputs, module organization is highly incentivized. Curating all modules into a single repository (as the FED team did at Stitch Fix) provides a straightforward approach for a team to refer to and reuse work.

4.5.5. 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.

4.5.6. Source Code Based Lineage

The declarative nature of Hamilton enables an entire end to end ML workflow to be modeled. Column level lineage from source, to machine learning feature, to model that consumes it, generally requires additional integration work to ensure it's emission and storage, e.g. with Amundsen. With Hamilton, no such integration or system is required. The declarative functions can model this entire process with any tooling that is python based, as the *function source code* becomes the source of truth. To build a standalone lightweight lineage system, one need to only pair the function definitions, driver code and configuration, with a source code version control system (e.g. git) to snapshot the code (e.g. git commit) when an artifact is created, to enable reconstruction of the DAG for lineage querying purposes.

4.5.7. Lineage for Data Privacy/Provenance Concerns

Hamilton unlocks the ability to provide fine grained lineage of computation. With the growth of privacy concerns and data regulation, organizations need to know what data comes in, where it goes, and how it is used. Hamilton functions can be marked (via `@tag()` with privacy or regulation concerns, e.g. that it contains Personally Identifiable Information (PII), enabling one to easily surface answers to questions of data usage and data impact from the structure of the DAG.

5. Evaluation

5.1. Adoption

To enjoy the benefits of Hamilton, one must use the paradigm. For existing systems, this means a migration needs to occur, which has been the largest friction point to adopting Hamilton. Internally, teams with active feature development for time-series forecasting have been the most prolific adopters, as they are the willing to pay the migration/adoption cost to reap the paradigm's benefits. Externally (since October 2021), at minimum, teams using Pandas and wanting to improve software engineering hygiene have been Hamilton's best adopters.

5.2. Quantitative assessment

A quantitative assessment of Hamilton's benefits to a team is challenging, as one would have to construct a tightly controlled experiment, e.g. like [19]. In an industry environment, however, it is hard to secure resourcing for such an endeavor. That said, anecdotally, for the FED team, a monthly feature engineering task of adding and adjusting data transformations 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!

5.3. Qualitative assessment

The initial 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. The team manages over 4000 data transforms, which represents almost a decade of work, written by at least fifteen different team members.

6. Summary

Hamilton is a novel dataflow framework that makes data transformation engineering in Python straightforward. By representing dataflows as a series of simple Python functions, Hamilton produces code that is easy to read and decoupled from execution. This results in transform logic that is always unit testable and documentation friendly, provides lineage out of the box, enables lightweight run time data quality checks, and unlocks fast iteration and debug cycles. It has enabled the FED team at Stitch Fix to scale, managing over 4000 data transforms that create features for time-series modeling.

In addition, Hamilton provides a layer of indirection that transparently scales computation onto various distributed computation frameworks (such as Ray, Spark, and Dask) as materialization is decoupled from function transform definitions. This opens the door for exciting future work.

7. Future Work

Here we highlight three avenues of future work. For more, see open issues in Hamilton's github repository.

7.1. Source code based data governance

With Hamilton, one can encode a rich repository of metadata (see section 4.5.7) into the source code directly. Because source code is required to perform data transformations, keeping transform logic synchronized with tags, data quality checks, and documentation is a simpler proposition than having that metadata in separate independent steps of a dataflow or separate systems. Therefore the source code itself could conceivably be used as a reliable base for data governance.

However, how to expose this information for consumption requires more thought. Does one build directly on top of the source code? Or does one emit this information to an existing system, such as a data catalog? For the former, a new system would need to be built. For the latter, one could integrate a continuous integration system that publishes changes when source code is snapshot (i.e. committed), or augment the Hamilton driver/DAG walking methodology to emit this information at DAG instantiation/execution time.

Similarly, data access/use policies could also be a target for source code based governance. By tagging functions that ingest data sources with appropriate data policies, one could, prior to DAG execution, walk the DAG to ensure the requesting user and requested DAG execution meets the policy requirements for those data sources.

7.2. Compiling to an orchestration framework

A common problem with ML tooling is choosing an orchestration system. This is a big decision, because companies rarely change this infrastructure. As Hamilton functions do not define or set materialization concerns, it cannot be used in place of an orchestration framework such as Airflow[15], 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 into any existing framework. Programmatically defining orchestration would also unlock the possibility for low cost infrastructure migrations, while avoiding vendor lock in.

7.3. Modeling your entire data warehouse independently of materialization concerns

Common industry data tools and orchestration frameworks leak materialization concerns into the user experience. For example, using SQL, the end user has to think in tables. This naturally cascades to how data is materialized and transferred between workflows. What if, instead, one could model the dependencies of one's data transforms, independently of how and where the data is stored? The declarative nature of Hamilton unlocks this possibility.

References

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

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

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

[4] M. Zaharia, R. S. Xin, P. Wendell, T. Das, M. Armbrust, A. Dave, X. Meng, J. Rosen, S. Venkataraman, M. J. Franklin, A. Ghodsi, J. Gonzalez, S. Shenker, I. Stoica, Apache spark: a unified engine for big data processing, Commun. ACM 59 (2016) 56–65. URL: http://doi.acm.org/10.1145/2934664. doi:10.1145/2934664.

[5] Various, Dask: Library for dynamic task scheduling, 2016. URL: https://dask.org.

[6] Pandas dev. team, pandas-dev/pandas: Pandas, 2020. URL: https://doi.org/10.5281/zenodo.3509134. doi:10.5281/zenodo.3509134.

[7] Various, An open framework for data lineage collection and analysis, 2017. URL: https://openlineage.io/.

[8] Various, Datahub, 2020. URL: https://github.com/datahub-project/datahub.

[9] Various, Amundsen, 2019. URL: https://github.com/amundsen-io/amundsen.

[10] Niels Bantilan, pandera: Statistical Data Validation of Pandas Dataframes, in: Meghann Agarwal, Chris Calloway, Dillon Niederhut, David Shupe (Eds.), Proceedings of the 19th Python in Science Conference, 2020, pp. 116 – 124. doi:10.25080/Majora-342d178e-010.

[11] S. Schelter, D. Lange, P. Schmidt, M. Celikel, F. Biessmann, A. Grafberger, Automating large-scale data quality verification, Proceedings of the VLDB Endowment 11 (2018) 1781–1794.

[12] Various, Great expectations, 2017. URL: https://github.com/great-expectations/great_expectations.

[13] Various, Metaflow: a framework for real-life data science, 2020. URL: https://github.com/Netflix/metaflow.

[14] Various, Prefect workflow management system, 2017. URL: https://github.com/PrefectHQ/prefect.

[15] Various, Apache airflow, 2015. URL: https://github.com/apache/airflow.

[16] Various, Dagster: An orchestration platform for the development, production, and observation of data assets, 2020. URL: https://github.com/dagster-io/dagster.

[17] Georg Brandl, Sphinx documentation, 2008. URL:

https://www.sphinx-doc.org/en/master/.

[18] T. Kakantousis, A. Kouzoupis, F. Buso, G. Berthou, J. Dowling, S. Haridi, Horizontally scalable ml pipelines with a feature store, in: Proc. 2nd SysML Conf., Palo Alto, USA, 2019.

[19] D. L. Moody, Cognitive load effects on end user understanding of conceptual models: An experimental analysis, in: A. Benczúr, J. Demetrovics, G. Gottlob (Eds.), Advances in Databases and Information Systems, Springer Berlin Heidelberg, Berlin, Heidelberg, 2004, pp. 129–143.

A. A full Hamilton Hello World Example

```python
## --- in my_functions.py
import pandas as pd

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


def spend_per_signup(spend: pd.Series, signups: pd.Series) -> pd.Series:
    """The cost per signup in relation to spend."""
    return spend / signups


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

## in run.py
import pandas as pd
from hamilton import driver
import my_functions  # we import user functions here

initial_columns = {  # load from actuals or wherever -- this is our initial data we use as input.
    # Note: these values don't have to be all series, they could be scalar.
    'signups': pd.Series([1, 10, 50, 100, 200, 400]),
    'spend': pd.Series([10, 10, 20, 40, 40, 50]),
}
# instantiate the DAG - multiple modules can be passed
dr = driver.Driver(initial_columns, my_functions)
# we need to specify what we want in the final dataframe
output_columns = [
    'spend',
    'signups',
    'avg_3wk_spend',
    'spend_per_signup',
    'spend_zero_mean_unit_variance'
]
# by default execution returns a dataframe
df = dr.execute(output_columns)
print(df.to_string())

# To visualize do `pip install sf-hamilton[visualization]` if you want these to work
dr.visualize_execution(output_columns, './my_dag.dot', {})
dr.display_all_functions('./my_full_dag.dot')
```

Listing 5: A full hello world example.