Scalable and Sustainable Feature Engineering with Hamilton

Elijah ben Izzy
CTO/Co-founder @ DAGWorks, Inc.

TL;DR

I want to convince you that...

1. Maintaining feature engineering code is difficult
2. Hamilton can help you:
    a. build *sustainable* code
    b. build *scalable* code
3. Hamilton can model your ML workflow end to end
4. Hamilton is easy to get started with/easy to use!

DAGWORKS

At DAGWorks we're making ML pipelines easy to manage.

Nobody should be afraid to inherit data science code.

>>> I'm not selling you anything in this talk! <<<

Hamilton is Open Source!!

```
> pip install sf-hamilton
```

Get started in <15 minutes!

Try it out

https://www.tryhamilton.dev/

Documentation

https://hamilton.readthedocs.io/

Try Hamilton Get Started Reference Documentation DAGWorks Inc. GitHub

https://www.tryhamilton.dev/

Hamilton

Wrangle Pandas codebases into shape.

Learn (5 mins) Github 890+ ⭐

- [x] Write always unit testable code
- [x] Add runtime data validation easily
- [x] Produce readable and maintainable code
- [x] Visualize lineage (click the run button to see)
- [x] Run anywhere python runs: in airflow, jupyter, fastapi, etc...
- [x] Skip the CS degree to use it

Try Hamilton right here in your browser 👇

```python
# Declare and link your transformations as functions....
import pandas as pd

def a(input: pd.Series) -> pd.Series:
    return input % 7

def b(a: pd.Series) -> pd.Series:
    return a * 2

def c(a: pd.Series, b: pd.Series) -> pd.Series:
    return a * 3 + b * 2

def d(c: pd.Series) -> pd.Series:
    return c ** 3
```

```python
# And run them!
import functions
from hamilton import driver
dr = driver.Driver({}, functions)
result = dr.execute(
   ['a', 'b', 'c', 'd'],
   inputs={'input': pd.Series([1, 2, 3, 4, 5])}
)
print(result)
dr.display_all_functions("graph.dot", {})
```

Run me!

The Agenda

A motivating story of DS pain
The solution: *Hamilton*
Hamilton for feature engineering
- ↳ **Sustainable feature management**
- ↳ **Scalable feature pipelines**

Hamilton for end-to-end ML workflows
OS progress + next steps

The Agenda

A motivating story of DS pain

The solution: *Hamilton*

Hamilton for feature engineering

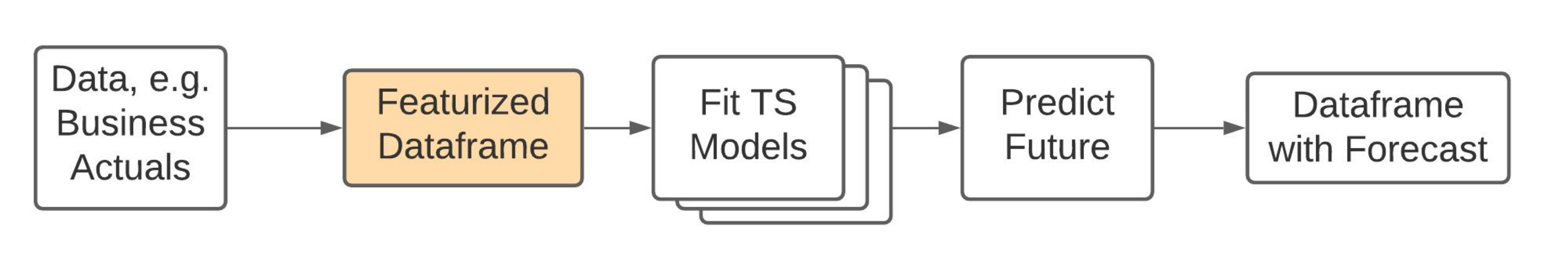
- ↳ Sustainable feature management
- ↳ Scalable feature pipelines

Hamilton for end-to-end ML workflows

OS progress + next steps

A Problem From my Last Job...

Forecasting the business (demand, signups, churn)

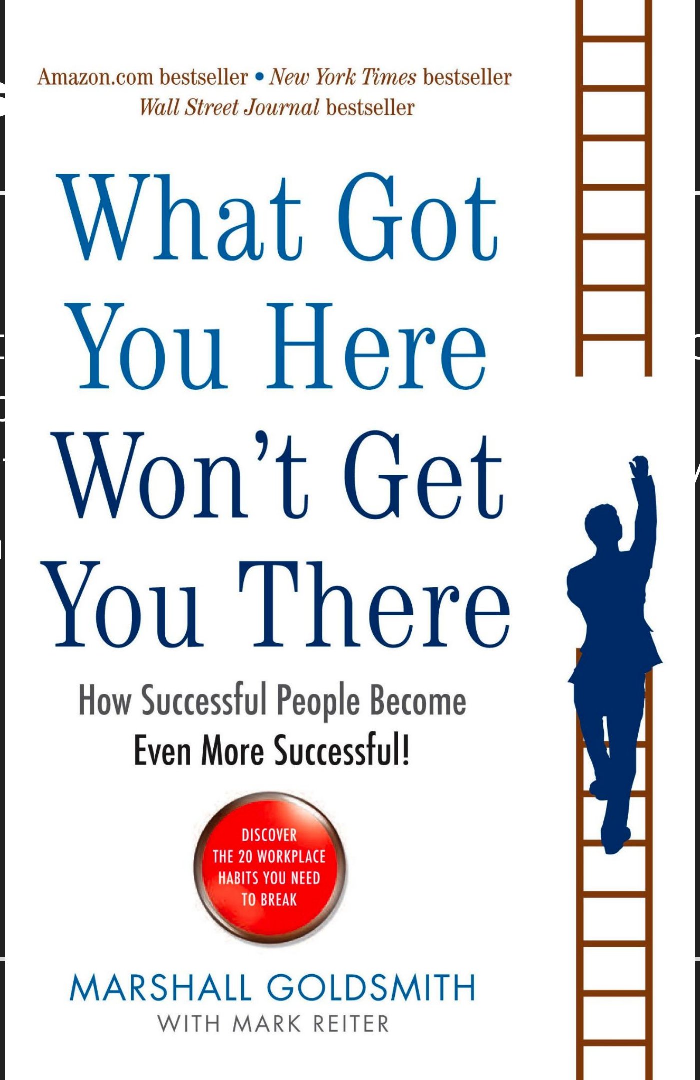


Some Business

Approach

- O(1000+) operat
- Configurations b
- Multiple layers o

A world-class team



aframe

visualize data

Some Business-Critical Tech Debt

Problems with the code?

- ❏ **Unit testing**: *difficult*
- ❏ **Documentation**: *unnatural, unenforced*
- ❏ **Modularity**: *non-existent*
- ❏ **Data catalogue**: *lots of grepping*
- ❏ **Debugging**: *run the whole pipeline*
- ❏ **Data validation**: *run the whole pipeline, not really done*

Perfect solution to forecasting problem + time = spaghetti code

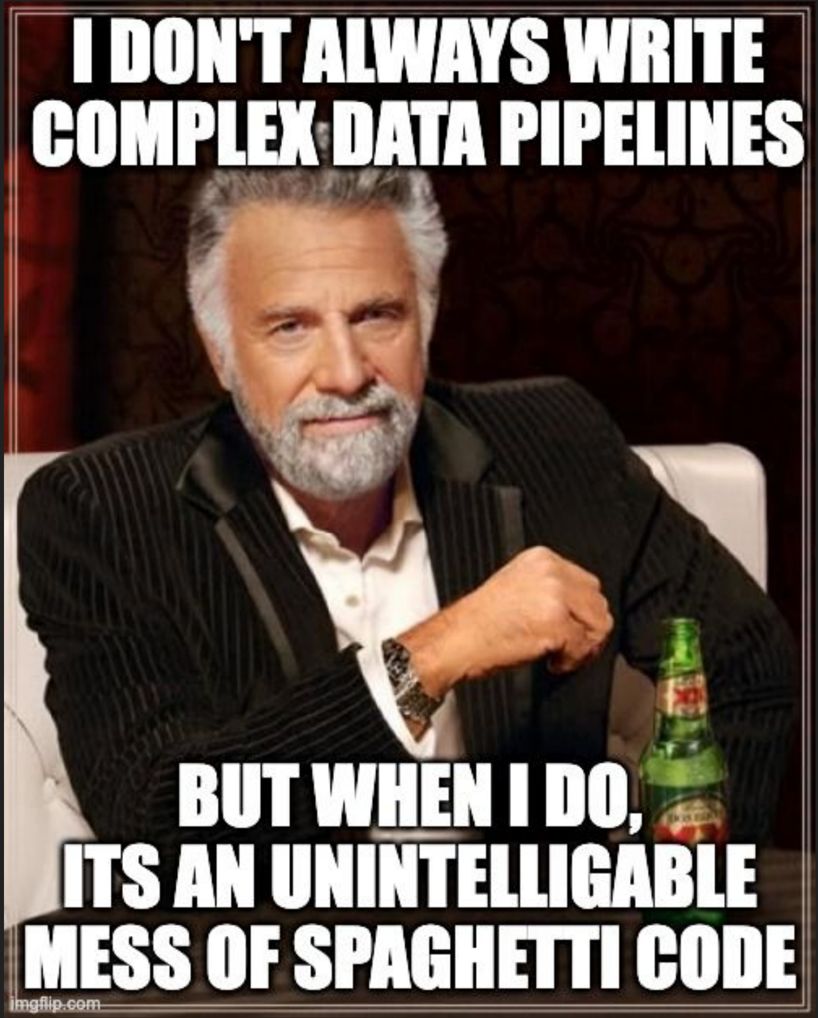
Some Business-Critical Tech Debt

Q: What happens when you have all of those problems, and...

- ❏ You want to expand your models to new regions?
- ❏ You have to add complex scenarios on management’s whim?
- ❏ You have a data bug and very little time to solve it?

A: It wasn’t fun.

+ This is not a unique experience to my prior role, time-series forecasting, or even pandas


I DON'T ALWAYS WRITE COMPLEX DATA PIPELINES
BUT WHEN I DO, ITS AN UNINTELLIGABLE MESS OF SPAGHETTI CODE
imgflip.com

The Agenda

A motivating story of DS pain
The solution: *Hamilton*
Hamilton for feature engineering
- ↳ Sustainable feature management
- ↳ Scalable feature pipelines

Hamilton for end-to-end ML workflows
OS progress + next steps

Hamilton: the "A-ha" Moment

Idea: What if every column corresponded to exactly one python fn?

Idea 2: What if the way that function was written tells you everything you needed to know?

*In Hamilton, the artifact (column) is determined by the **name of the function**. The dependencies are determined by **the parameters**.*

Old Way vs Hamilton Way:

Instead of*

```python
df['c'] = df['a'] + df['b']
df['d'] = transform(df['c'])
```

You declare

```python
def c(a: pd.Series, b: pd.Series) -> pd.Series:
    """Sums a with b"""
    return a + b

def d(c: pd.Series) -> pd.Series:
    """Transforms C to ..."""
    new_column = _transform_logic(c)
    return new_column
```

**Hamilton supports *all* python objects, not just dfs/series!*

Old Way vs Hamilton Way:

Instead of

```python
df['c'] = df['a'] + df['b']
df['d'] = transform(df['c'])
```

Outputs == Function Name

Inputs == Function Arguments

You declare

```python
def c(a: pd.Series, b: pd.Series) -> pd.Series:
    """Sums a with b"""
    return a + b

def d(c: pd.Series) -> pd.Series:
    """Transforms C to ..."""
    new_column = _transform_logic(c)
    return new_column
```

Full Hello World

Functions

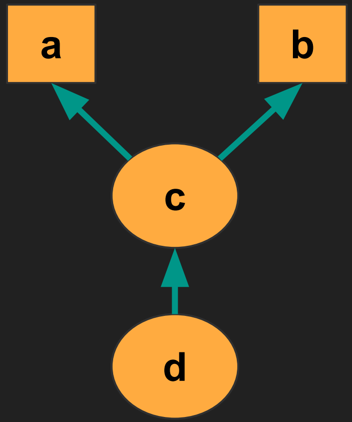
```python
# feature_logic.py
def c(a: pd.Series, b: pd.Series) -> pd.Series:
    """Sums a with b"""
    return a + b

def d(c: pd.Series) -> pd.Series:
    """Transforms C to ..."""
    new_column = _transform_logic(c)
    return new_column
```

Driver says what/when to execute

```python
# run.py
from hamilton import driver
import feature_logic
dr = driver.Driver({'a': ..., 'b': ...}, feature_logic)
df_result = dr.execute(['c', 'd'])
print(df_result)
```

Hamilton TL;DR:

1. For each transform (=), you write a function(s)
2. Functions declare a DAG
3. Hamilton handles DAG execution




```python
# feature_logic.py
def c(a: pd.Series, b: pd.Series) -> pd.Series:
    """Replaces c = a + b"""
    return a + b

def d(c: pd.Series) -> pd.Series:
    """Replaces d = transform(c)"""
    new_column = _transform_logic(c)
    return new_column
```

```python
# run.py
from hamilton import driver
import feature_logic
dr = driver.Driver({'a': ..., 'b': ...},
                   feature_logic)
df_result = dr.execute(['c', 'd'])
print(df_result)
```

Hamilton: Extensions

Q: Doesn't Hamilton make your code more verbose?

A: Yes, but that's not always a bad thing. When it is, we have decorators!

- ❏ `@tag # attach metadata`
- ❏ `@parameterize # curry + repeat a function`
- ❏ `@extract_columns # one dataframe -> multiple series`
- ❏ `@check_output # data validation`
- ❏ `@config.when # conditional transforms`
- ❏ `@subdag # recursively utilize groups of nodes`
- ❏ *`@... # new ones all the time`*

To Summarize...

Hamilton forces you to write transforms in python functions.

These python functions provide everything you need:

- ❏ **Unit testing**: *simple – plain python functions!*
- ❏ **Documentation**: *use the docstring*
- ❏ **Modularity**: *Small pieces -> by definition*
- ❏ **Data catalogue**: *Code = central feature definition store*
- ❏ **Debugging**: *Execute functions individually + breakpoints*
- ❏ **Trustworthy data**: *Validation included out of the box*

Decorators → powerful, higher-order operations

Driver → decouple transform definition from execution

Initial Use Case

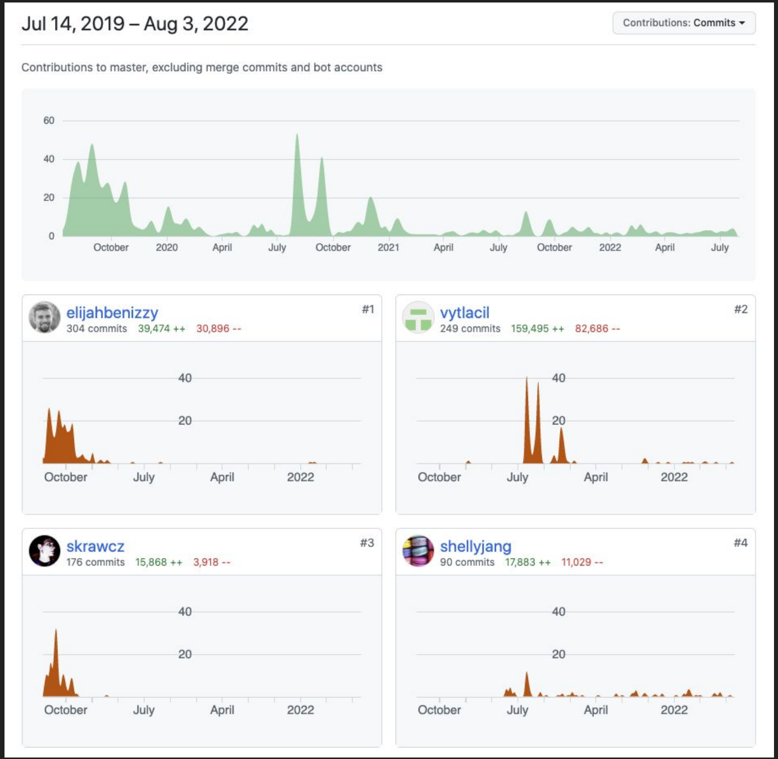
Running in production for **3+** years

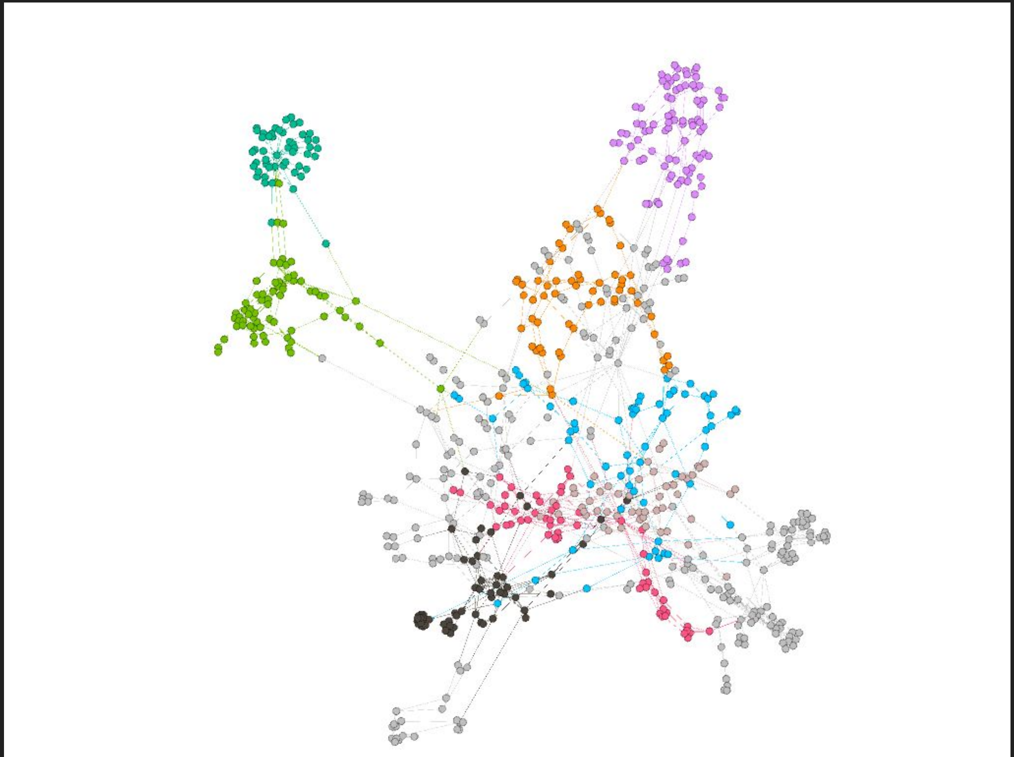
Initial use-case manages **4000+** feature definitions

Data science teams ❤️ it

- ❏ Enabled 4x faster monthly model + feature update
- ❏ Easy to onboard new team members
- ❏ Code reviews are simple
- ❏ Finally have unit tests
- ❏ Fewer bugs/quicker resolutions
- ❏ Better features + models

Initial Use Case



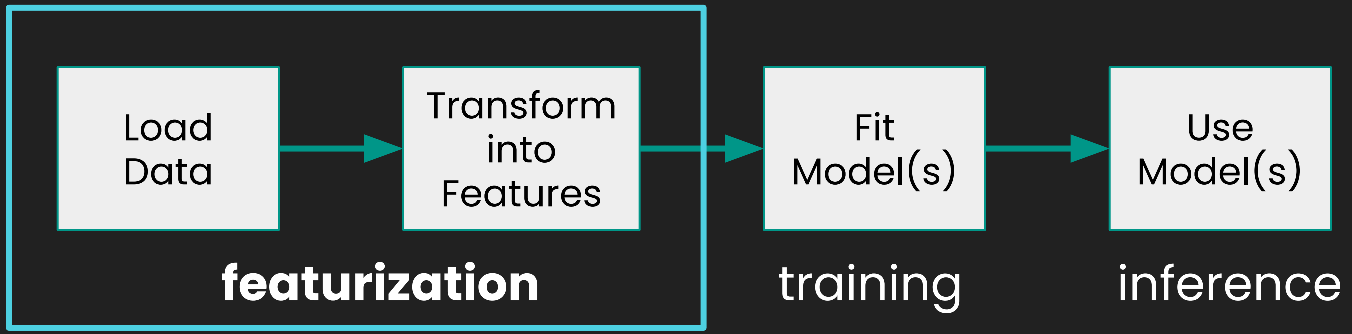


The Agenda

A motivating story of DS pain
The solution: *Hamilton*
Hamilton for feature engineering
- ↳ Sustainable feature management
- ↳ Scalable feature pipelines

Hamilton for end-to-end ML workflows
OS progress + next steps

Hamilton + Feature Engineering: Overview



Note:

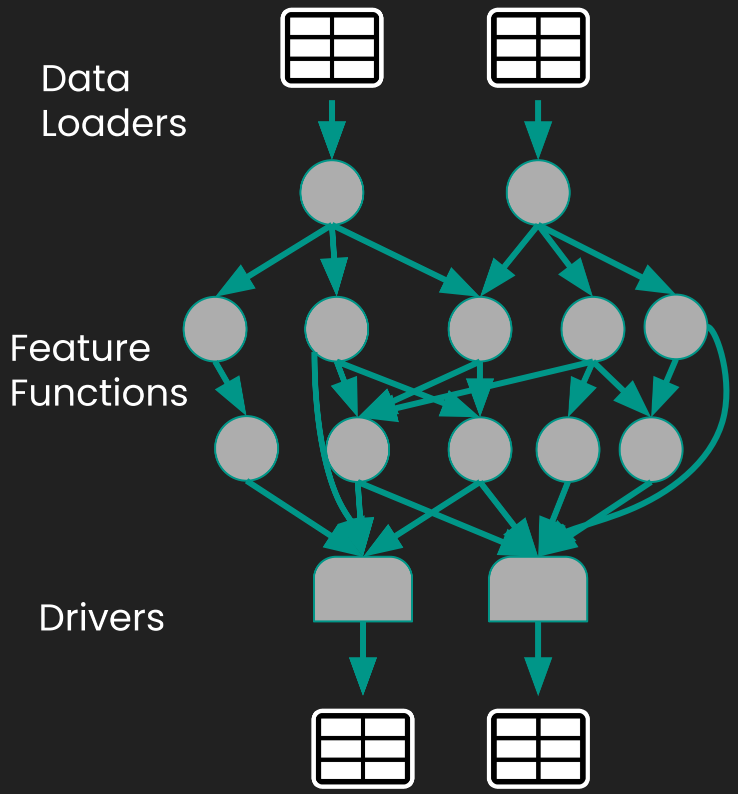
- ❏ Works for any python object type (not just dataframes!)
- ❏ Embeddable anywhere python runs – orchestration systems (airflow, kubeflow, metaflow, flytekit, prefect, dagster, …) + web services!

Modeling Feature Engineering

Code that needs to be written:

1. Functions to load data
2. Transform/feature functions
3. Driver to materialize data

Execute only what's needed...

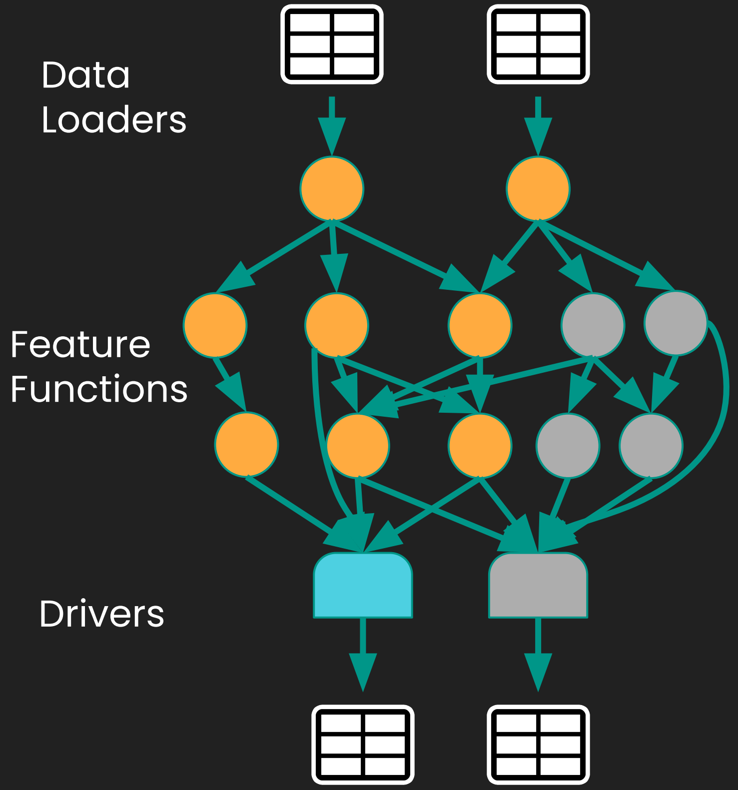


Modeling Feature Engineering

Code that needs to be written:

1. Functions to load data
2. Transform/feature functions
3. Driver to materialize data

Execute only what's needed...



Feature Engineering Challenges

Sustainable code

- ❏ Highly coupled code
- ❏ Difficulty debugging/understanding flows
- ❏ Messy collaboration on complex pipelines
- ❏ Validating your data

Hamilton solves this!

Scaling the data

- ❏ Data is too big to fit in memory
- ❏ Cannot easily parallelize computation

Hamilton has integrations for this!

The Agenda

A motivating story of DS pain
The solution: *Hamilton*
Hamilton for feature engineering
- ↳ **Sustainable feature management**
- ↳ Scalable feature pipelines

Hamilton for end-to-end ML workflows
OS progress + next steps

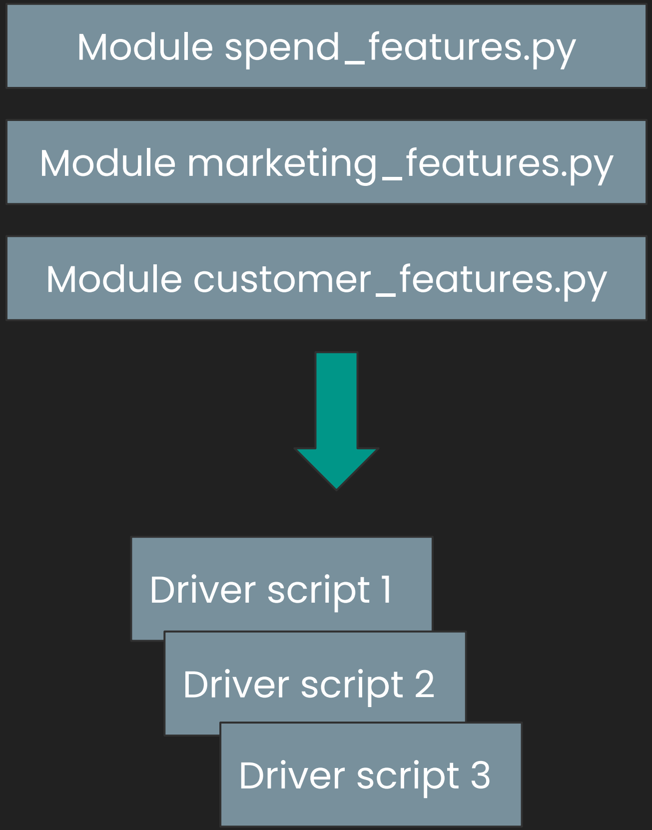
Decoupling Code

From infrastructure

- ❏ Driver handles execution
- ❏ Functions handle business logic

From itself

- ❏ Code organized into functions
- ❏ Functions organized into modules
- ❏ Functions do not know about Hamilton



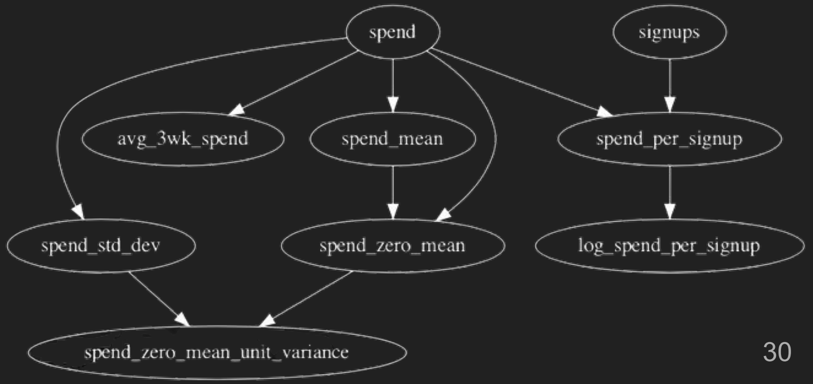
Ease of Debugging

Knocking bugs

- ❑ Rerun just the broken paths
- ❑ Python functions → unit tests + natural debugging
- ❑ Runtime data quality checks
- ❑ Quickly narrow search space of data bugs

Understanding/visualizing

- ❑ Visualize dataflow/execution path
- ❑ Clearly track dependencies



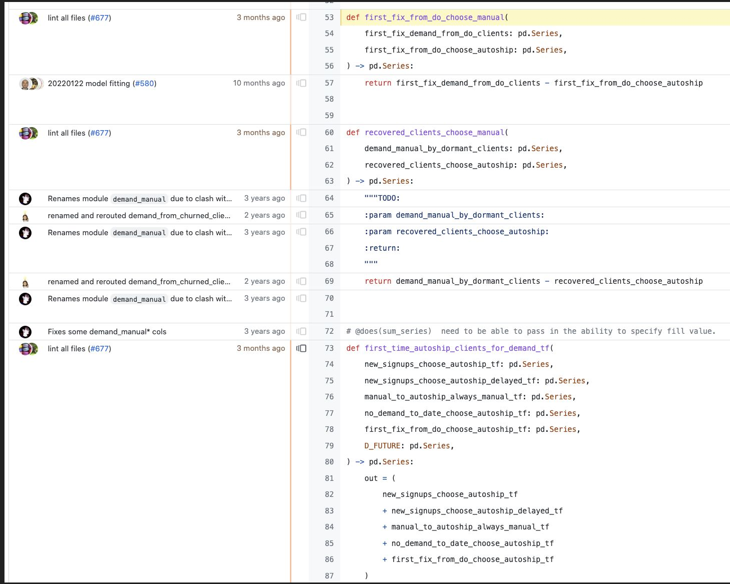
Natural Collaboration

Centralized feature definition store

- ❑ Forces alignment on naming
- ❑ Documentation is included/natural
- ❑ Minimize conflicts when collaborating

Change management

- ❑ Feature versions in git
- ❑ All change ∈ git history
- ❑ PRs are easy to read – trace changes back to functions



Handling Data Validation

Garbage in/garbage out

- ❑ How can you build reliable pipelines if the data is bad?

Solution

- ❑ Runtime data validation decorator!

```python
@check_output(
    data_type=np.float64, # data type
    range=(-1.0, 1.0), # range
    allow_nans=False, # no nans
    importance="warn") # warn, don't fail
def some_data_we_care_about() -> pd.Series:
   return ...
```

Basic Checks

A few custom-built checks for a quick-start:

- ❏ Range
- ❏ Nan-checks (any or percentage)
- ❏ Valid categories
- ❏ Null outputs
- ❏ Plenty more…

For pandas + primitives

Highly pluggable!

Pandera Integration

But wait, there's more! Pandera + Hamilton = happy, powerful checks

```
import pandera as pa
import pandas as pd
from hamilton import function_modifiers

@function_modifiers.check_output(schema=pa.DataFrameSchema(
        {
            'column1': pa.Column(int),
            'column2': pa.Column(float, pa.Check(lambda s: s < -1.2)),
            # you can provide a list of validators
            'column3': pa.Column(str, [
                pa.Check(lambda s: s.str.startswith('value')),
                pa.Check(lambda s: s.str.split('_', expand=True).shape[1] == 2)
            ]),
        },
        index=pa.Index(int),
        strict=True,
    ))
def dataframe_with_schema(...) -> pd.DataFrame:
    ...
```

Data Check Extensibility

Implement base-class to write your own…

`@check_output_custom(MyDataValidationClass(...))`

Goal – add integrations for

- Any type of dataframe/datatype
- Multiple validation frameworks (great expectations, deequ, whylogs…)

Sky's the limit!

```python
@check_output_custom(AllPrimeValidator(...))
def prime_number_generator(number_of_primes_to_generate: int) -> pd.Series:
    pass
```

Summing up Sustainability with some Code…

```python
# client_features.py
@tag(owner='client_recs', pii='False')
@check_output(data_type=np.float64, range=(-5.0, 5.0), allow_nans=False)
def client_height_normalized(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
```

Summing up Sustainability with some Code…

```python
# client_features.py
@tag(owner='client_recs', pii='False')
@check_output(data_type=np.float64, range=(-5.0, 5.0), allow_nans=False)
def client_height_normalized(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
```

The feature **client_height_normalized is** owned by the **client recommendations team,** contains **no PII,** is of type **float,** depends on **2 upstream features,** has **no Nan values** and produces values in **[-5.0, 5.0]**. VCS shows its **change over time**. **Grep the codebase** for downstream deps.

EL

Knowing nothing else about this code besides what you have, how would you translate it into english?

```
# client_features.py
@tag(owner='client_recs', pii='False')
@check_output(data_type=np.float64, range=(-5.0, 5.0), allow_nans=False)
def client_height_normalized(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
```

This is a Python code that defines a function called `client_height_normalized`. The purpose of this function is to return the zero mean unit variance value of height, based on two input variables: `height_zero_mean` and `height_std_dev`, which are both pandas Series objects.

Before the function definition, there are two function decorators that are applied to the `client_height_normalized` function. These decorators modify the behavior of the function in some way.

The first decorator, `@tag(owner='client_recs', pii='False')`, adds metadata to the function, indicating that it is owned by a client record system and that it does not handle personally identifiable information (PII).

The second decorator, `@check_output(data_type=np.float64, range=(-5.0, 5.0), allow_nans=False)`, performs input/output validation for the function. It checks that the output of the function is a NumPy float64 data type, falls within the range of -5.0 and 5.0, and does not allow NaN (not-a-number) values.

In the function definition, `height_zero_mean` is divided by `height_std_dev` to calculate the zero mean unit variance value of height. The resulting value is returned as a pandas Series object.

The Agenda

A motivating story of DS pain
The solution: *Hamilton*
Hamilton for feature engineering
- ↳ Sustainable feature management
- ↳ **Scalable feature pipelines**

Hamilton for end-to-end ML workflows
OS progress + next steps

Scaling Compute/Data

Approach: *delegate*

Ray/Dask

- ❏ Run in parallel
- ❏ Single machine → multiprocessing → cluster

Pandas on Spark (ex-koalas)

- ❏ Scale horizontally per dataset

Switching only requires making a **driver-side change***

**pandas on spark requires changing of data loaders as well…*

Scaling with Ray

```python
# run.py
from hamilton import driver
import data_loaders
import date_features
import spend_features
config = {...} # config, e.g. data_location
dr = driver.Driver(config,
                  data_loaders,
                  date_features,
                  spend_features)
features_wanted = [...] # choose subset wanted
feature_df = dr.execute(features_wanted)
save(feature_df, 'prod.features')
```

Scaling with Ray

```
# run_on_ray.py
…
from hamilton import base, driver
from hamilton.experimental import h_ray
…
ray.init()
config = {...}
rga = h_ray.RayGraphAdapter(
    result_builder=base.PandasDataFrameResult())
dr = driver.Driver(config,
                  data_loaders, date_features, spend_features,
                  adapter=rga)
features_wanted = [...]  # choose subset wanted
feature_df = dr.execute(features_wanted,
                        inputs=date_features)
save(feature_df, 'prod.features')
ray.shutdown()
```

Scaling with Dask

```
# run_on_dask.py
…
from hamilton import base, driver
from hamilton.experimental import h_dask
…
client = Client(Cluster(...)) # dask cluster/client
config = {...}
dga = h_dask.DaskGraphAdapter(client,
    result_builder=base.PandasDataFrameResult())
dr = driver.Driver(config,
                   data_loaders, date_features, spend_features,
                   adapter=dga)
features_wanted = [...]  # choose subset wanted
feature_df = dr.execute(features_wanted,
                        inputs=date_features)
save(feature_df, 'prod.features')
client.shutdown()
```

Scaling with Pandas-on-Spark

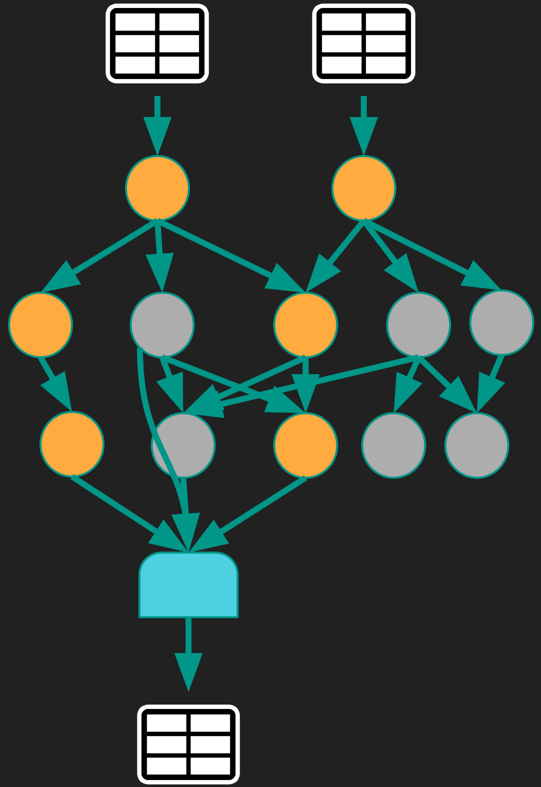
```
# run_on_pandas_on_spark.py
…
import pyspark.pandas as ps
from hamilton import base, driver
from hamilton.experimental import h_spark
…
spark = SparkSession.builder.getOrCreate()
ps.set_option(...)
config = {...}
skga = h_dask.SparkKoalasGraphAdapter(spark, spine='COLUMN_NAME',
    result_builder=base.PandasDataFrameResult())
dr = driver.Driver(config,
                   spark_data_loaders, date_features,spend_features,
                   adapter=skga)
features_wanted = [...]  # choose subset wanted
feature_df = dr.execute(features_wanted,
                        inputs=date_features)
save(feature_df, 'prod.features')
spark.stop()
```

Hamilton + Ray/Dask: How Does it Work?

```
# FUNCTIONS
def c(a: pd.Series, b: pd.Series) -> pd.Series:
    """Sums a with b"""
    return a + b

def d(c: pd.Series) -> pd.Series:
    """Transforms C to ..."""
    new_column = _transform_logic(c)
    return new_column
```

```
# DRIVER
from hamilton import base, driver
from hamilton.experimental import h_ray
…
ray.init()
config = {...}
rga = h_ray.RayGraphAdapter(
    result_builder=base.PandasDataFrameResult())
dr = driver.Driver(config,
                   data_loaders,
                   date_features,
                   spend_features,
                    adapter=rga)
features_wanted = [...]  # choose subset wanted
feature_df = dr.execute(features_wanted,
                        inputs=date_features)
save(feature_df, 'prod.features')
ray.shutdown()
```

DAG

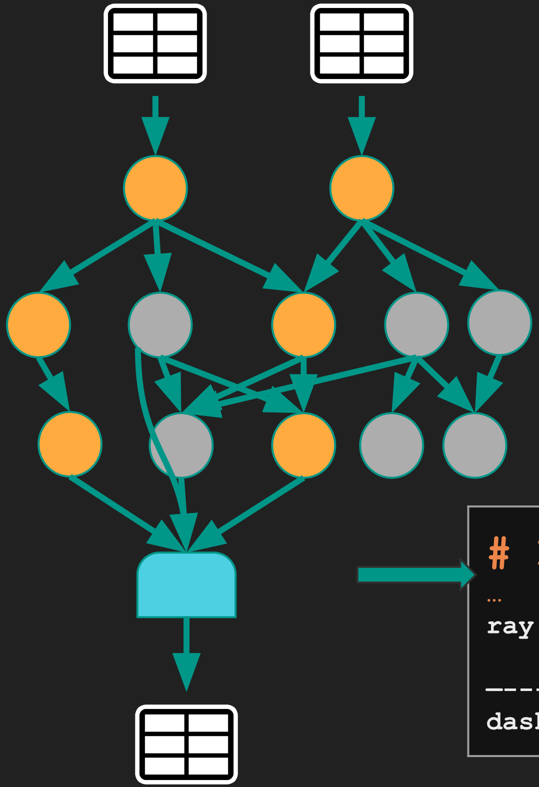


Hamilton + Ray/Dask: How Does it Work?

```python
# FUNCTIONS
def c(a: pd.Series, b: pd.Series) -> pd.Series:
    """Sums a with b"""
    return a + b

def d(c: pd.Series) -> pd.Series:
    """Transforms C to ..."""
    new_column = _transform_logic(c)
    return new_column
```

```python
# DRIVER
from hamilton import base, driver
from hamilton.experimental import h_ray
…
ray.init()
config = {...}
rga = h_ray.RayGraphAdapter(
    result_builder=base.PandasDataFrameResult())
dr = driver.Driver(config,
                   data_loaders,
                   date_features,
                   spend_features,
                   adapter=rga)
features_wanted = [...]  # choose subset wanted
feature_df = dr.execute(features_wanted,
                        inputs=date_features)
save(feature_df, 'prod.features')
ray.shutdown()
```

DAG



```python
# Delegate to Ray/Dask
…
ray.remote(
    node.callable).remote(**kwargs)
--------------------
dask.delayed(node.callable)(**kwargs)
```

Hamilton + Spark: How Does it Work?

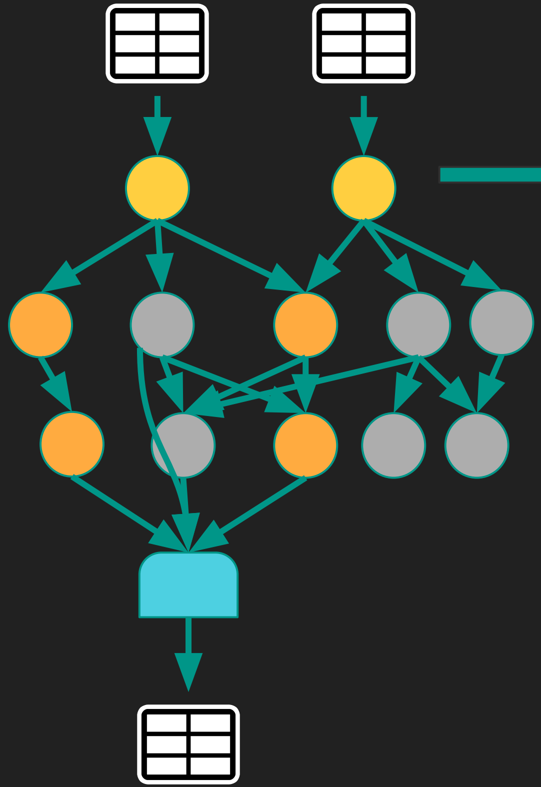
FUNCTIONS

```python
def c(a: pd.Series, b: pd.Series) -> pd.Series:
   """Sums a with b"""
   return a + b

def d(c: pd.Series) -> pd.Series:
   """Transforms C to ..."""
   new_column = _transform_logic(c)
   return new_column
```

DRIVER

```python
from hamilton import base, driver
from hamilton.experimental import h_ray
…
ray.init()
config = {...}
rga = h_ray.RayGraphAdapter(
    result_builder=base.PandasDataFrameResult())
dr = driver.Driver(config,
                   data_loaders,
                   date_features,
                   spend_features,
                   adapter=rga)
features_wanted = [...]  # choose subset wanted
feature_df = dr.execute(features_wanted,
                        inputs=date_features)
save(feature_df, 'prod.features')
ray.shutdown()
```

DAG



With Spark

…

Change these to load Spark "Pandas" equivalent object instead.

Spark will take care of the rest.

Hamilton + Ray/Dask/Pandas on Spark: Caveats

Serialization

- ❏ Uses serialization methodology of delegated frameworks

Memory:

- ❏ Defaults should work – fine tuning at fn level not yet supported

Python dependencies:

- ❏ You need to manage them

Looking to graduate these APIs from experimental status

🔶🔶 Contributions wanted here to extend support in Hamilton! 🔶🔶

The Agenda

A motivating story of DS pain
The solution: *Hamilton*
Hamilton for Feature Engineering
- ↳ Sustainable feature management
- ↳ Scalable feature pipelines

Hamilton for end-to-end ML workflows
OS progress + next steps

End-to-End ML Pipelines

What does an ML pipeline look like? Fancy ETL:

- ❏ [E] Load data from a feature store
- ❏ [T] Transform features
- ❏ [T] Train model
- ❏ [T] Run model Inference
- ❏ [T] Evaluate model performance
- ❏ [L] Save metrics
- ❏ [L] Save artifacts
- ❏ [L] Save training data

End-to-End ML Pipelines

Centralize logic, abstract integrations

- ❏ [E] Load data from a feature store
- ❏ [T] Transform features
- ❏ [T] Train model
- ❏ [T] Run model Inference
- ❏ [T] Evaluate model performance
- ❏ [L] Save metrics
- ❏ [L] Save artifacts
- ❏ [L] Save training data

} Express as functions

} Decouple from logic

End-to-End ML Pipelines

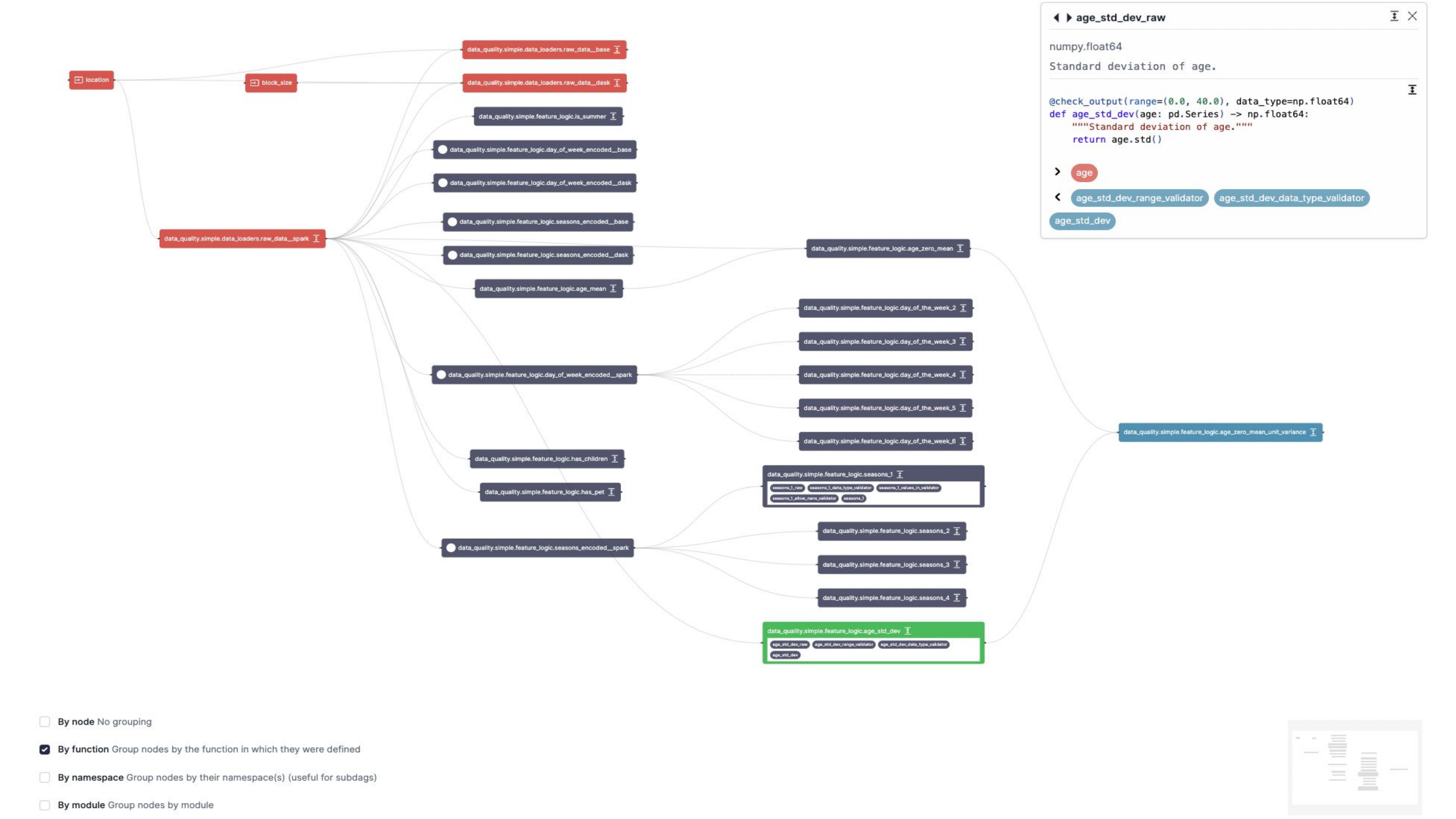
Decouple from logic

- ❏ Swap out
    - ❏ Model store
    - ❏ Metrics store
    - ❏ Feature source/store
- ❏ Execute same code online/offline
    - ❏ Map ops 1:1
    - ❏ Join -> data load using DAG config
    - ❏ aggregation -> data load/hardcoded

End-to-End ML Pipelines

Gain Visibility

- ❏ Lineage
  - ❏ Trace fn params for managing dependencies
  - ❏ Trace data from source -> transforms -> sinks
  - ❏ Tag metadata-> understand properties of dependencies
- ❏ Catalogue
  - ❏ Browse features == browse through code
  - ❏ Documentation attached to artifact name


location
block_size
data_quality.simple.data_loaders.raw_data__base
data_quality.simple.data_loaders.raw_data__dask
data_quality.simple.feature_logic.is_summer
data_quality.simple.feature_logic.day_of_week_encoded__base
data_quality.simple.feature_logic.day_of_week_encoded__dask
data_quality.simple.data_loaders.raw_data__spark
data_quality.simple.feature_logic.seasons_encoded__base
data_quality.simple.feature_logic.seasons_encoded__dask
data_quality.simple.feature_logic.age_mean
data_quality.simple.feature_logic.age_zero_mean
data_quality.simple.feature_logic.day_of_the_week_2
data_quality.simple.feature_logic.day_of_the_week_3
data_quality.simple.feature_logic.day_of_week_encoded__spark
data_quality.simple.feature_logic.day_of_the_week_4
data_quality.simple.feature_logic.day_of_the_week_5
data_quality.simple.feature_logic.day_of_the_week_6
data_quality.simple.feature_logic.age_zero_mean_unit_variance
data_quality.simple.feature_logic.has_children
data_quality.simple.feature_logic.seasons_1
seasons_1_raw
seasons_1_data_type_validator
seasons_1_values_in_validator
seasons_1_allow_nans_validator
seasons_1
data_quality.simple.feature_logic.has_pet
data_quality.simple.feature_logic.seasons_encoded__spark
data_quality.simple.feature_logic.seasons_2
data_quality.simple.feature_logic.seasons_3
data_quality.simple.feature_logic.seasons_4
data_quality.simple.feature_logic.age_std_dev
age_std_dev_raw
age_std_dev_range_validator
age_std_dev_data_type_validator
age_std_dev
age_std_dev_raw
numpy.float64
Standard deviation of age.
@check_output(range=(0.0, 40.0), data_type=np.float64)
def age_std_dev(age: pd.Series) -> np.float64:
"""Standard deviation of age."""
return age.std()
age
age_std_dev_range_validator
age_std_dev_data_type_validator
age_std_dev
By node No grouping
By function Group nodes by the function in which they were defined
By namespace Group nodes by their namespace(s) (useful for subdags)
By module Group nodes by module

Our Vision

Unifying layer for ML ETLs

Compile to orchestration frameworks

Integrate with data quality vendors/OS options

Trivially load from/to a variety of sources/destinations

SQL support for full ETL

The Agenda

A motivating story of DS pain
The solution: *Hamilton*
Hamilton for feature engineering
- ↳ Sustainable feature management
- ↳ Scalable feature engineering

Hamilton to model end-to-end ML workflows
OS progress + next steps

OS Progress

Early stages, but thriving community

- ❏ Some exciting users
- ❏ Growing set of core contributors
- ❏ Full company dedicated to building it!

Looking for

- ❏ Contributors
- ❏ Bug hunters
- ❏ User feedback

STITCH FIX

BRITISH CYCLING

IBM®

Government Digital Service

Pacific Northwest NATIONAL LABORATORY

LexisNexis® RISK SOLUTIONS

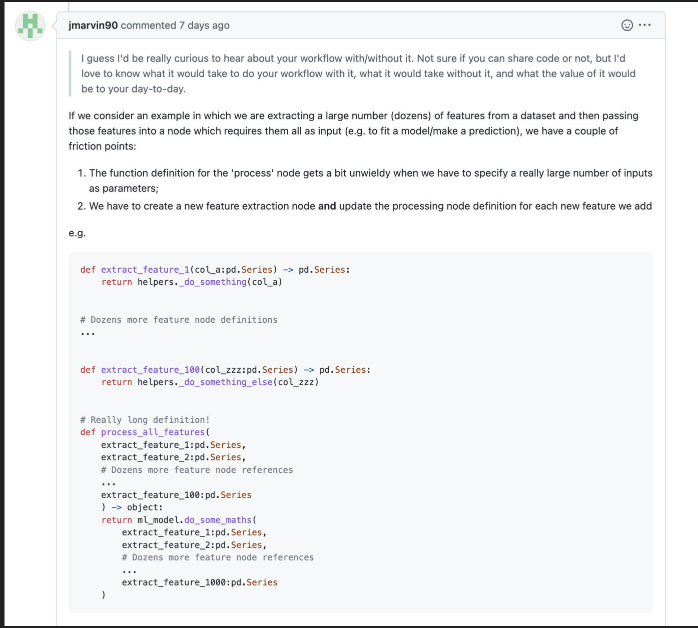
In Progress

Expressive APIs

- ❑ Schema for artifact metadata
- ❑ Adapter for SQL
- ❑ <Your idea here!>

Execution

- ❑ Compilation/dataflow specification
- ❑ Streaming/generator support
- ❑ First-class pyspark integration
- ❑ <Your idea here!>



jmarvin90 commented 7 days ago

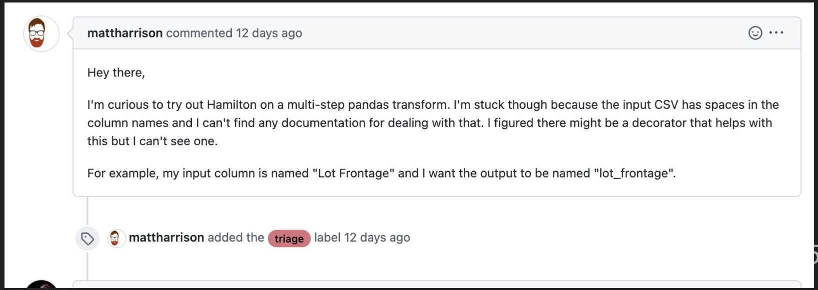
> I guess I'd be really curious to hear about your workflow with/without it. Not sure if you can share code or not, but I'd love to know what it would take to do your workflow with it, what it would take without it, and what the value of it would be to your day-to-day.

If we consider an example in which we are extracting a large number (dozens) of features from a dataset and then passing those features into a node which requires them all as input (e.g. to fit a model/make a prediction), we have a couple of friction points:

1. The function definition for the 'process' node gets a bit unwieldy when we have to specify a really large number of inputs as parameters;
2. We have to create a new feature extraction node **and** update the processing node definition for each new feature we add

e.g.

```
def extract_feature_1(col_a:pd.Series) -> pd.Series:
    return helpers._do_something(col_a)


# Dozens more feature node definitions
...


def extract_feature_100(col_zzz:pd.Series) -> pd.Series:
    return helpers._do_something_else(col_zzz)


# Really long definition!
def process_all_features(
    extract_feature_1:pd.Series,
    extract_feature_2:pd.Series,
    # Dozens more feature node references
    ...
    extract_feature_100:pd.Series
    ) -> object:
    return ml_model.do_some_maths(
        extract_feature_1:pd.Series,
        extract_feature_2:pd.Series,
        # Dozens more feature node references
        ...
        extract_feature_1000:pd.Series
    )
```



mattharrison commented 12 days ago

Hey there,

I'm curious to try out Hamilton on a multi-step pandas transform. I'm stuck though because the input CSV has spaces in the column names and I can't find any documentation for dealing with that. I figured there might be a decorator that helps with this but I can't see one.

For example, my input column is named "Lot Frontage" and I want the output to be named "lot_frontage".

mattharrison added the triage label 12 days ago

Give Hamilton a Try! We'd Love Your Feedback.

www.tryhamilton.dev

> `pip install sf-hamilton`

⭐ on github (https://github.com/dagworks-inc/hamilton)

☑️ create & vote on issues on github

📣 join us on on Slack (https://join.slack.com/t/hamilton-opensource/shared_invite/zt-1bjs72asx-wcUTgH7q7QX1igiQ5bbdcg)

Thank you.

Questions?

Yell at me online https://twitter.com/elijahbenizzy

Connect with me https://www.linkedin.com/in/elijahbenizzy/

Code with me https://github.com/dagworks-inc/hamilton

Use sparingly :) elijah@dagworks.io