Feature Engineering with Hamilton

March 2023 **@ Women Who Code: Data Science**

Stefan Krawczyk - DAGWorks Inc. (YCW23)

whoami

Stefan Krawczyk
Co-creator of **Hamilton** &&
CEO **DAGWorks** Inc. (YCW23)

12+ years in ML & Data platforms

STITCH FIX

IDIBON

IBM

VICTORIA UNIVERSITY OF WELLINGTON
TE HERENGA WAKA

whoami

Stefan Krawczyk
Co-creator of **Hamilton** &&
CEO **DAGWorks** Inc. (YCW23)

12+ years in ML & Data platforms

100+ DS

STITCH FIX

IDIBON

IBM

VICTORIA UNIVERSITY OF WELLINGTON TE HERENGA WAKA

TL;DR: talk overview in 5 slides

```python
# load_data defined off screen...
data = load_data()
data['avg_3wk_spend'] = data['spend'].rolling(3).mean()
data['spend_per_signup'] = data['spend']/data['signups']
spend_mean = data['spend'].mean()
data['spend_zero_mean'] = data['spend'] - spend_mean
spend_std_dev = data['spend'].std()
data['spend_zero_mean_unit_variance'] = data['spend_zero_mean']/spend_std_dev
print(data.to_string())
```

Here is 1% of some project you're inheriting

&

You have two choices...

```python
# new_way.py
def avg_3wk_spend(spend: pd.Series) -> pd.Series:
  """Rolling 3 day 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
```

Left

OR

Right

```python
# load_data defined off screen...
data = load_data()
data['avg_3wk_spend'] = data['spend'].rolling(3).mean()
data['spend_per_signup'] = data['spend']/data['signups']
spend_mean = data['spend'].mean()
data['spend_zero_mean'] = data['spend'] - spend_mean
spend_std_dev = data['spend'].std()
data['spend_zero_mean_unit_variance'] = data['spend_zero_mean']/spend_std_dev
print(data.to_string())
```

Here is 1% of some project you're inheriting

&

You have two choices...

```python
# new_way.py
def avg_3wk_spend(spend: pd.Series) -> pd.Series:
  """Rolling 3 day 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
```

Left OR Right

Left

```python
# load_data defined off screen...
data = load_data()
data['avg_3wk_spend'] = data['spend'].rolling(3).mean()
data['spend_per_signup'] = data['spend']/data['signups']
spend_mean = data['spend'].mean()
data['spend_zero_mean'] = data['spend'] - spend_mean
spend_std_dev = data['spend'].std()
data['spend_zero_mean_unit_variance'] = data['spend_zero_mean']
    /spend_std_dev
print(data.to_string())
```

Right

```python
# new_way.py
def avg_3wk_spend(spend: pd.Series) -> pd.Series:
  """Rolling 3 day 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
```

	Left	Right
Unit/Int. testing	❌	✅
Documentation	❌	✅
Lineage	❌	✅
Reuse/ Modularity	❌	✅

Right: Hamilton

Standardizes how you describe your work:

Data, ML, LLM, Web workflows

Right

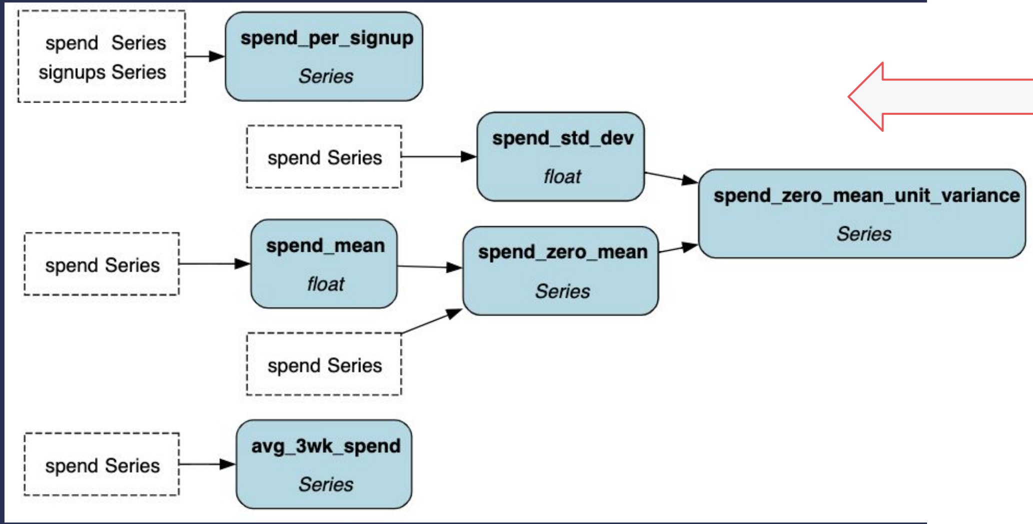
```python
# new_way.py
def avg_3wk_spend(spend: pd.Series) -> pd.Series:
  """Rolling 3 day average spend."""
  return spend.rolling(3).mean()


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
```

And a simple 'driver' to run it:

```python
from hamilton import driver
import new_way
dr = driver.Driver({}, new_way)
outputs = ["spend", "signups", "avg_3wk_spend", "spend_per_signup",
    "spend_zero_mean", "spend_zero_mean_unit_variance"]
result = dr.execute(
  outputs,
  inputs=load_data().to_dict(orient="series")
)
print(result.to_string())
```

Right: Hamilton

Write Code → Get a DAG.



Right

```python
# new_way.py
def avg_3wk_spend(spend: pd.Series) -> pd.Series:
  """Rolling 3 day 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
```

And a simple 'driver' to run it:

```python
from hamilton import driver
import new_way
dr = driver.Driver({}, new_way)
outputs = ["spend", "signups", "avg_3wk_spend", "spend_per_signup",
    "spend_zero_mean", "spend_zero_mean_unit_variance"]
result = dr.execute(
  outputs,
  inputs=load_data().to_dict(orient="series")
)
print(result.to_string())
```

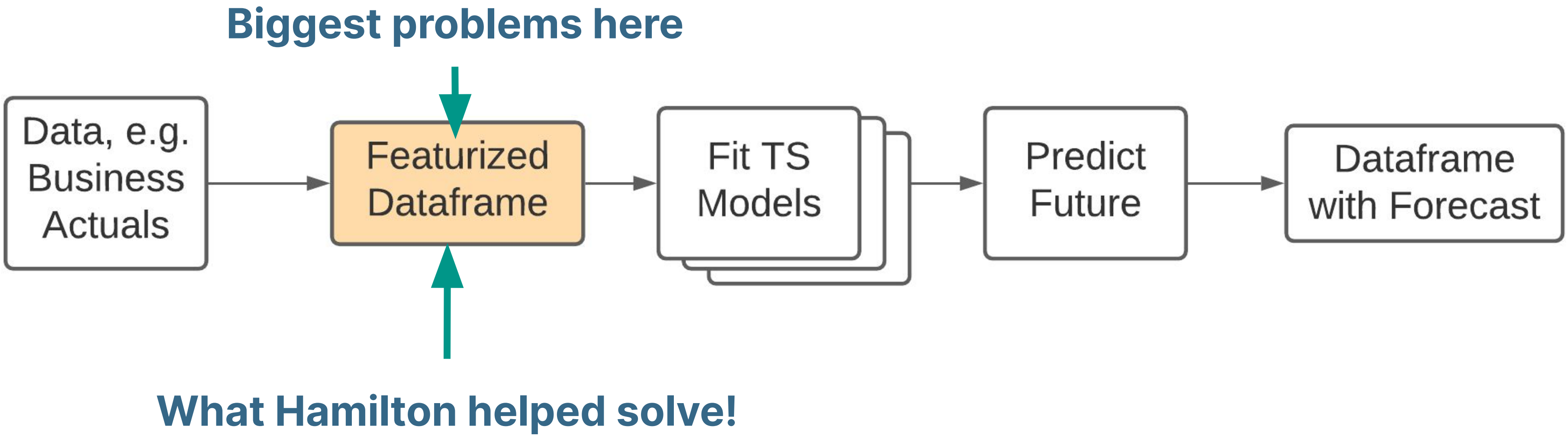
Backstory

How Hamilton came to be

1. **Motivating pain**
2. Hamilton
3. Feature Eng.
4. Summary

Motivating Pain

- You're a DS team that provides operational forecasts for the business.
- The business makes decisions based on your numbers.
- You need to constantly model change in the world.



Example: Creating a dataframe (e.g. for ML training)

```
df = loader.load_actuals(dates) # e.g. spend, signups
```

Example: Creating a dataframe (e.g. for ML training)

```
df = loader.load_actuals(dates) # e.g. spend, signups
```

Example: Creating a dataframe (e.g. for ML training)

```
df = loader.load_actuals(dates) # e.g. spend, signups
if config['region'] == 'UK':
    df['holidays'] = is_uk_holiday(df['year'], df[' week'])
else:
    df['holidays'] = is_holiday(df['year'], df['week'])
```

Example: Creating a dataframe (e.g. for ML training)

```
df = loader.load_actuals(dates) # e.g. spend, signups
if config['region'] == 'UK':
   df['holidays'] = is_uk_holiday(df['year'], df[' week'])
else:
   df['holidays'] = is_holiday(df['year'], df['week'])
df['avg_3wk_spend'] = df['spend'].rolling(3).mean()
df['acquisition_cost'] = df['spend'] / df['signups']
df['spend_shift_3weeks'] = df['spend'].shift(3)
```

Example: Creating a dataframe (e.g. for ML training)

```
df = loader.load_actuals(dates) # e.g. spend, signups
if config['region'] == 'UK':
   df['holidays'] = is_uk_holiday(df['year'], df[' week'])
else:
   df['holidays'] = is_holiday(df['year'], df['week'])
df['avg_3wk_spend'] = df['spend'].rolling(3).mean()
df['acquisition_cost'] = df['spend'] / df['signups']
df['spend_shift_3weeks'] = df['spend'].shift(3)
df['special_feature1'] = compute_bespoke_feature(df)
df['spend_b'] = multiply_columns(df['acquisition_cost'], df['B'])
```

Example: Creating a dataframe (e.g. for ML training)

```
df = loader.load_actuals(dates) # e.g. spend, signups
if config['region'] == 'UK':
   df['holidays'] = is_uk_holiday(df['year'], df[' week'])
else:
   df['holidays'] = is_holiday(df['year'], df['week'])
df['avg_3wk_spend'] = df['spend'].rolling(3).mean()
df['acquisition_cost'] = df['spend'] / df['signups']
df['spend_shift_3weeks'] = df['spend'].shift(3)
df['special_feature1'] = compute_bespoke_feature(df)
df['spend_b'] = multiply_columns(df['acquisition_cost'], df['B'])
save_df(df, "some_location")
```

Example: Creating a dataframe (e.g. for ML training)

```
df = loader.load_actuals(dates) # e.g. spend, signups
if config['region'] == 'UK':
   df['holidays'] = is_uk_holiday(df['year'], df[' week'])
else:
   df['holidays'] = is_holiday(df['year'], df['week'])
df['avg_3wk_spend'] = df['spend'].rolling(3).mean()
df['acquisition_cost'] = df['spend'] / df['signups']
df['spend_shift_3weeks'] = df['spend'].shift(3)
df['special_feature1'] = compute_bespoke_feature(df)
df['spend_b'] = multiply_columns(df['acquisition_cost'], df['B'])
save_df(df, "some_location")
```

😬 Now picture the passage of time: personnel Δ, sophistication ⬆️, etc

Problem: unit & integration testing; data quality 👎

```
df = loader.load_actuals(dates) # e.g. spend, signups
if config['region'] == 'UK':
   df['holidays'] = is_uk_holiday(df['year'], df[' week'])
else:
   df['holidays'] = is_holiday(df['year'], df['week'])
df['avg_3wk_spend'] = df['spend'].rolling(3).mean()
df['acquisition_cost'] = df['spend'] / df['signups']
df['spend_shift_3weeks'] = df['spend'].shift(3)
df['special_feature1'] = compute_bespoke_feature(df)
df['spend_b'] = multiply_columns(df['acquisition_cost'], df['B'])
save_df(df, "some_location")
```

😬 Now picture the passage of time: personnel Δ, sophistication ⬆️, etc

Problem: code readability & documentation 🧐

```
df = loader.load_actuals(dates) # e.g. spend, signups
if config['region'] == 'UK':
    df['holidays'] = is_uk_holiday(df['year'], df[' week'])
else:
    df['holidays'] = is_holiday(df['year'], df['week'])
df['avg_3wk_spend'] = df['spend'].rolling(3).mean()
df['acquisition_cost'] = df['spend'] / df['signups']
df['spend_shift_3weeks'] = df['spend'].shift(3)
df['special_feature1'] = compute_bespoke_feature(df)
df['spend_b'] = multiply_columns(df['acquisition_cost'], df['B'])
save_df(df, "some_location")
```

?

😬 Now picture the passage of time: personnel Δ, sophistication ⬆️, etc

Problem: difficulty in tracing lineage 🤯

```
df = loader.load_actuals(dates) # e.g. spend, signups
if config['region'] == 'UK':
   df['holidays'] = is_uk_holiday(df['year'], df[' week'])
else:
   df['holidays'] = is_holiday(df['year'], df['week'])
df['avg_3wk_spend'] = df['spend'].rolling(3).mean()
df['acquisition_cost'] = df['spend'] / df['signups']
df['spend_shift_3weeks'] = df['spend'].shift(3)
df['special_feature1'] = compute_bespoke_feature(df)
df['spend_b'] = multiply_columns(df['acquisition_cost'], df['B'])
save_df(df, "some_location")
```

😬 Now picture the passage of time: personnel Δ, sophistication ⬆️, etc

Problem: code reuse and duplication

```
df = loader.load_actuals(dates) # e.g. spend, signups
if config['region'] == 'UK':
    df['holidays'] = is_uk_holiday(df['year'], df[' week'])
else:
    df['holidays'] = is_holiday(df['year'], df['week'])
df['avg_3wk_spend'] = df['spend'].rolling(3).mean()
df['acquisition_cost'] = df['spend'] / df['signups']
df['spend_shift_3weeks'] = df['spend'].shift(3)
df['special_feature1'] = compute_bespoke_feature(df)
df['spend_b'] = multiply_columns(df['acquisition_cost'], df['B'])
save_df(df, "some_location")
```

😬 Now picture the passage of time: personnel Δ, sophistication ⬆️, etc

Problem: onboarding 📈 & debugging 📈

```
df = loader.load_actuals(dates) # e.g. spend, signups
if config['region'] == 'UK':
   df['holidays'] = is_uk_holiday(df['year'], df[' week'])
else:
   df['holidays'] = is_holiday(df['year'], df['week'])
df['avg_3wk_spend'] = df['spend'].rolling(3).mean()
df['acquisition_cost'] = df['spend'] / df['signups']
df['spend_shift_3weeks'] = df['spend'].shift(3)
df['special_feature1'] = compute_bespoke_feature(df)
df['spend_b'] = multiply_columns(df['acquisition_cost'], df['B'])
save_df(df, "some_location")
```

😬 Now picture the passage of time: personnel Δ, sophistication ⬆️, etc
At Stitch Fix there was 1000+ features…

Question for you!

1. Are any of these pains familiar to you? If so, which ones?
2. Would you be in anguish if you suddenly had to inherit your colleagues code that looked like this?

✋ Raise hand | Unmute !

What is Hamilton?

1. ~~Motivating pain~~
2. **Hamilton**
3. Feature Eng.
4. Summary

What is Hamilton?

Micro-orchestration framework for defining dataflows using declarative functions

SWE best practices: ☑️ testing ☑️ documentation ☑️ modularity/reuse

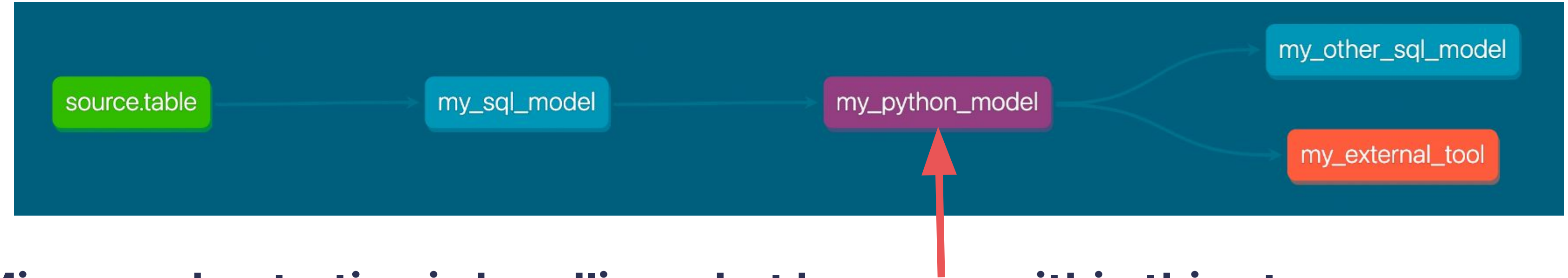
`pip install sf-hamilton [came from Stitch Fix]`

www.tryhamilton.dev ← uses pyodide!

Mirco-orchestration vs Macro-orchestration

Macro-orchestration is handling this whole thing

e.g. airflow, or DBT, etc.:



Micro-orchestration is handling what happens within this step

e.g. code that Airflow / DBT runs.

What's a dataflow?

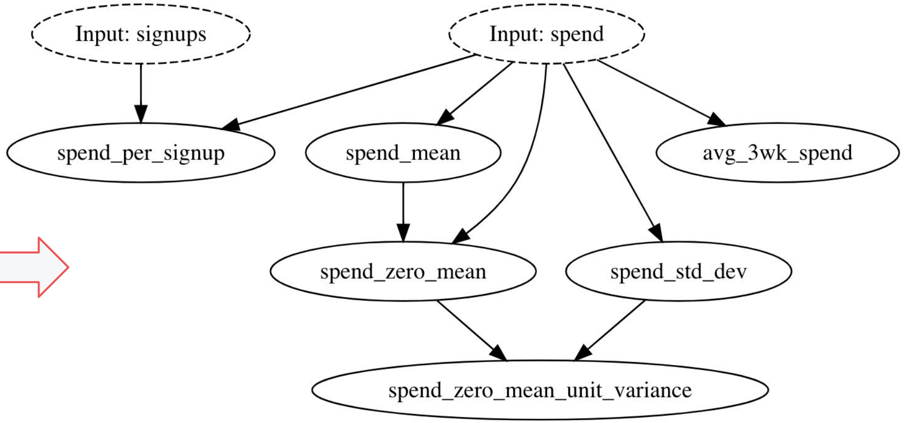
Fancy way of saying:

How data + computation "flow"

Can be expressed as a directed acyclic graph (DAG).

e.g., this is a dataflow:

```
df['avg_3wk_spend'] = df['spend'].rolling
df['spend_per_signup'] = df['spend']/df['
spend_mean = df['spend'].mean()
df['spend_zero_mean'] = df['spend']   spe
spend_std_dev = df['spend'].std()
df['spend_zero_mean_unit_variance'] = df[
```



Declarative functions?

Functions *declare:*

- What they create in the dataflow.
- What dependencies are required for computation.
- You don't run the functions directly.

> When you read the function, you'll understand what it does and what it needs.

A-ha moment: debugging a dataframe

Idea: What if every output (column) corresponded to exactly one Python fn?

Addendum: What if you could determine the dependencies from the way that function was written?

In Hamilton, the **output** (e.g., column)

is determined by the **name of the function**.

The **dependencies** are determined by the **input parameters**.

Old Way vs. Hamilton Paradigm:

Instead of

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

Outputs == Function Name

Inputs == Function Arguments

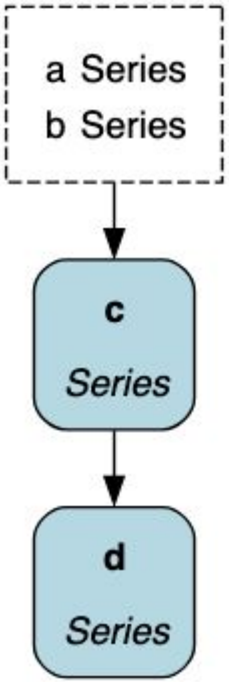
You declare

```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 (Note: works for any python object type)

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)
```

When ***should I*** consider Hamilton?

If you can draw a flowchart (DAG), you can put it into Hamilton:

- Feature engineering (Hamilton's roots) ← **Focus of today**
- Tired of managing scripts that do transformations...
- Describing E2E ML Pipelines + MLOps integrations
- Web request flows
- LLM Workflows! (e.g. replace langchain)

Code & software best practices enthusiasts:

- Hamilton ⬇️ Code Complexity

Things to mention, but I really won't cover:

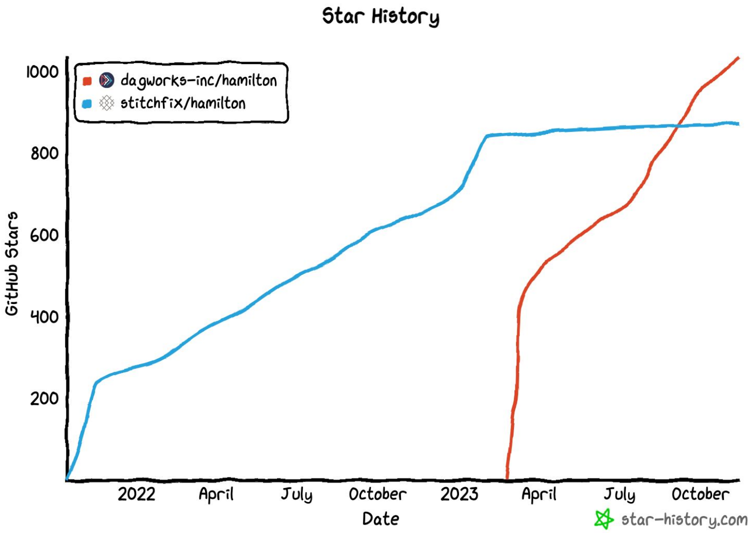
We also have decorators that you add to functions that...

```
● @tag                # attach metadata
● @parameterize       # curry + repeat a function
● @extract_columns    # one dataframe -> multiple series
● @extract_outputs    # one dict -> multiple outputs
● @check_output       # data validation; very lightweight
● @config.when        # conditional transforms
● @subdag             # parameterize parts of your DAG
```

& more... Hamilton code is **portable** & runs **& scales** anywhere python runs.

Some Hamilton stats

~1.8K Unique Stargazers
295+ slack members
173K+ downloads



Note: dbt took 3.5 years to get to 600 stars

Hamilton is used by many, including:

STITCH FIX
Joby AVIATION
BRITISH CYCLING
Adobe
AXIOM CLOUD
IBM
ascena RETAIL GROUP INC.
Government Digital Service
veriff
HABITAT ENERGY
Opendoor
Pacific Northwest NATIONAL LABORATORY
LexisNexis RISK SOLUTIONS

Feature Engineering

1. ~~Motivating pain~~
2. ~~Hamilton~~
3. **Feature Eng.**
4. Summary

Hamilton @ Stitch Fix

Running in production since 2019

One team manages **4000+** feature definitions

Data science teams ❤ it

- Enabled 4x faster monthly model + feature update
- Easy to onboard new team members - lineage & docs!
- Code reviews are faster
- Finally have unit tests
- Auto-generated sphinx documentation

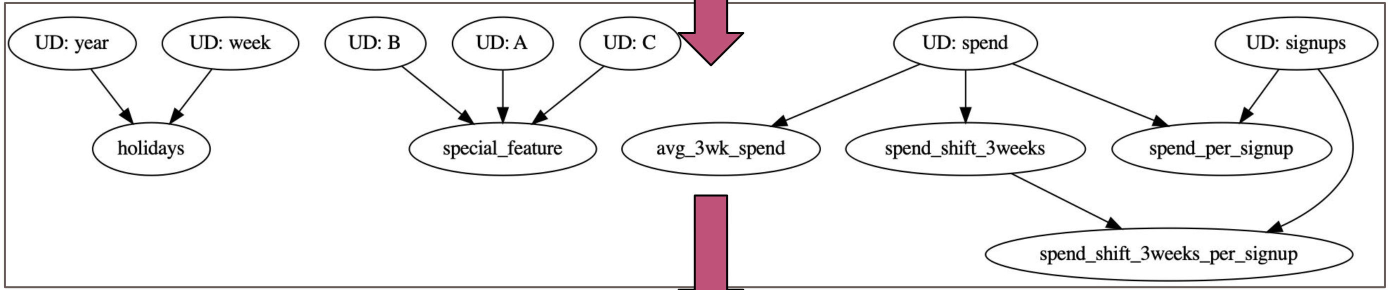
Feature Engineering with Hamilton

Data loading &
Feature code:

```python
def holidays(year: pd.Series, week: pd.Series) -> pd.Series:
    """Some docs"""
    return some_library(year, week)
def avg_3wk_spend(spend: pd.Series) -> pd.Series:
    """Some docs"""
    return spend.rolling(3).mean()
def spend_per_signup(spend: pd.Series, signups: pd.Series) -> pd.Series:
    """Some docs"""
    return spend / signups
def spend_shift_3weeks(spend: pd.Series) -> pd.Series:
    """Some docs"""
    return spend.shift(3)
def spend_shift_3weeks_per_signup(spend_shift_3weeks: pd.Series, signups: pd.Series) -> pd.Series:
    """Some docs"""
    return spend_shift_3weeks / signups
```

features*.py

Via
Driver:



Feature
Dataframe:

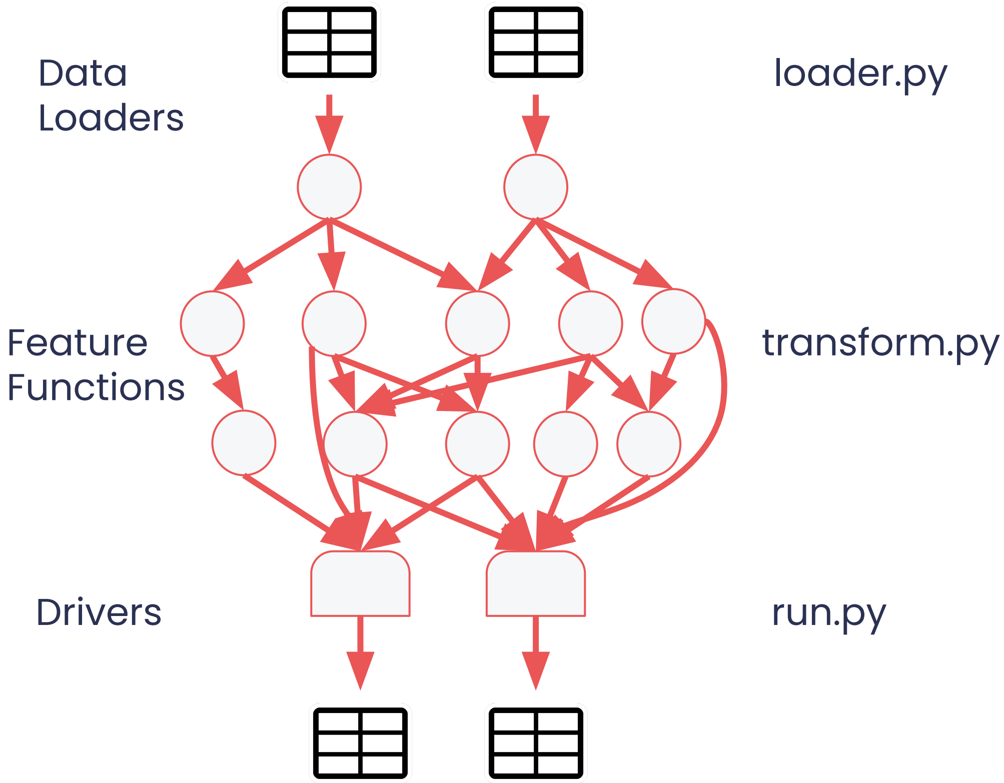
Year	Week	Sign ups	...	Spend	Holiday
2015	2	57	...	123	0
2015	3	58	...	123	0
2015	4	59	...	123	1
2015	5	59	...	123	1
...	...	...	...	...	...
...	...	...	...	...	...
...	...	...	...	...	...
...	...	...	...	...	...
2021	16	1000	...	1234	0

run.py

Feature Engineering with Hamilton

Code that needs to be written:

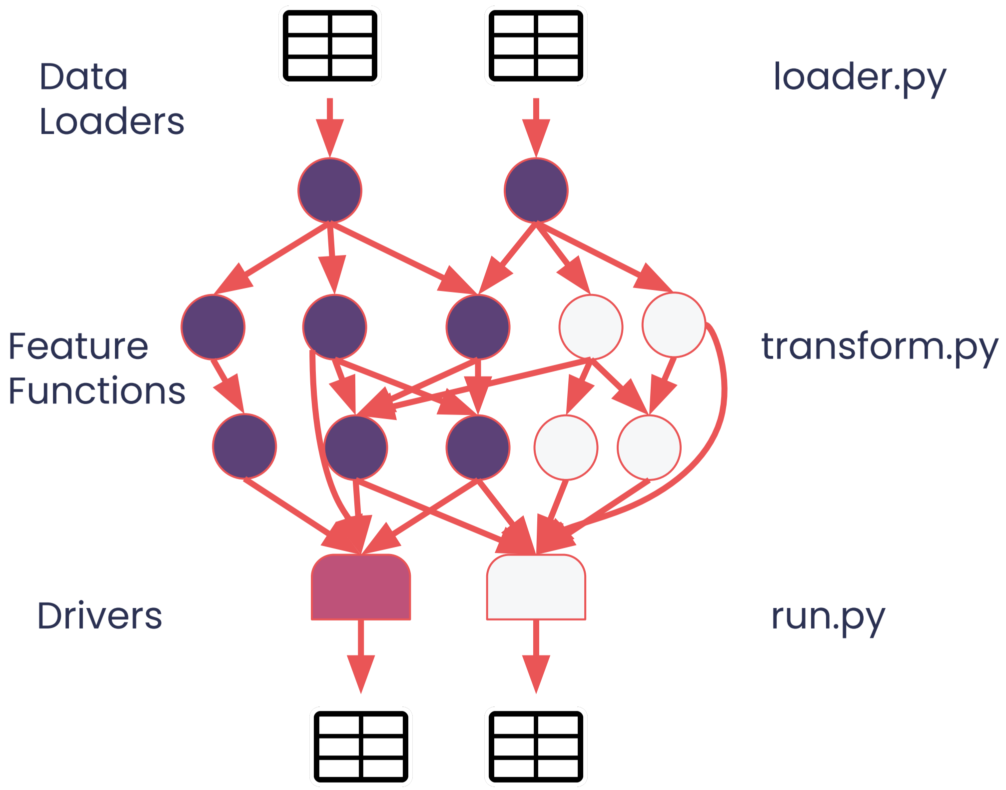
1. Functions to load data
2. Feature functions
3. Drivers materialize data



Feature Engineering with Hamilton

Code that needs to be written:

1. Functions to load data
2. Feature functions
3. Drivers materialize data



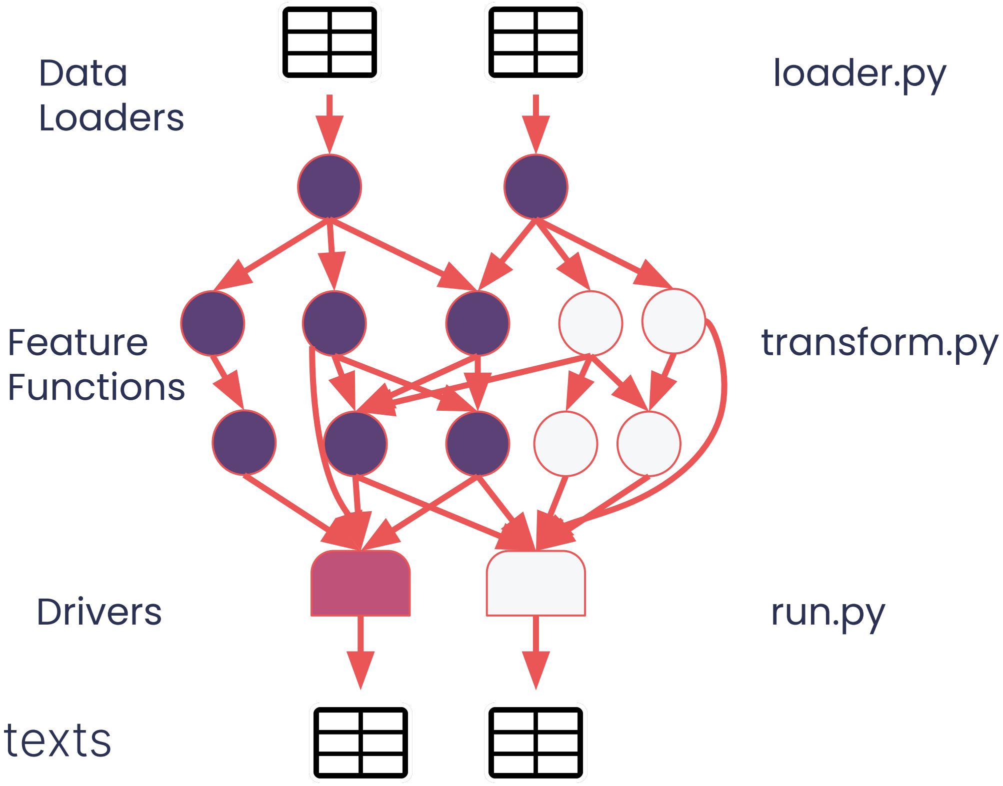
Feature Engineering with Hamilton

Code that needs to be written:

1. Functions to load data
2. Feature functions
3. Drivers materialize data

Code base implications:

- Natural structure emerges
- Logic modules vs execution contexts



Benefits of using Hamilton:

General: Testing & Documentation

```python
# client_features.py

def height_zero_mean_unit_variance(height_zero_mean: pd.Series,
                                   height_std_dev: float) -> pd.Series:
   return height_zero_mean / height_std_dev
```

Testing: easier to unit & integration test.

```python
# test_client_features.py

def test_height_zero_mean_unit_variance():
   actual = height_zero_mean_unit_variance(pd.Series([1,2,3]), 2.0)
   expected = pd.Series([0.5,1.0, 1.5])
   assert actual == expected
```

General: Testing & Documentation

```python
# client_features.py

@check_output(data_type=np.float64, range=(-5.0, 5.0), allow_nans=False)
def height_zero_mean_unit_variance(height_zero_mean: pd.Series,
                                   height_std_dev: float) -> pd.Series:
    return height_zero_mean / height_std_dev
```

Testing: easier to unit & integration test.

Data Quality Tests: runtime checks via annotation*; Pandera supported.

General: Testing & Documentation

```python
# client_features.py

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

Testing: easier to unit & integration test.

Data Quality Tests: runtime checks via annotation*; Pandera supported.

Self-documenting: naming, doc strings, annotations, & visualization

General: Testing & Documentation

```python
# client_features.py

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

Testing: easier to unit & integration test.

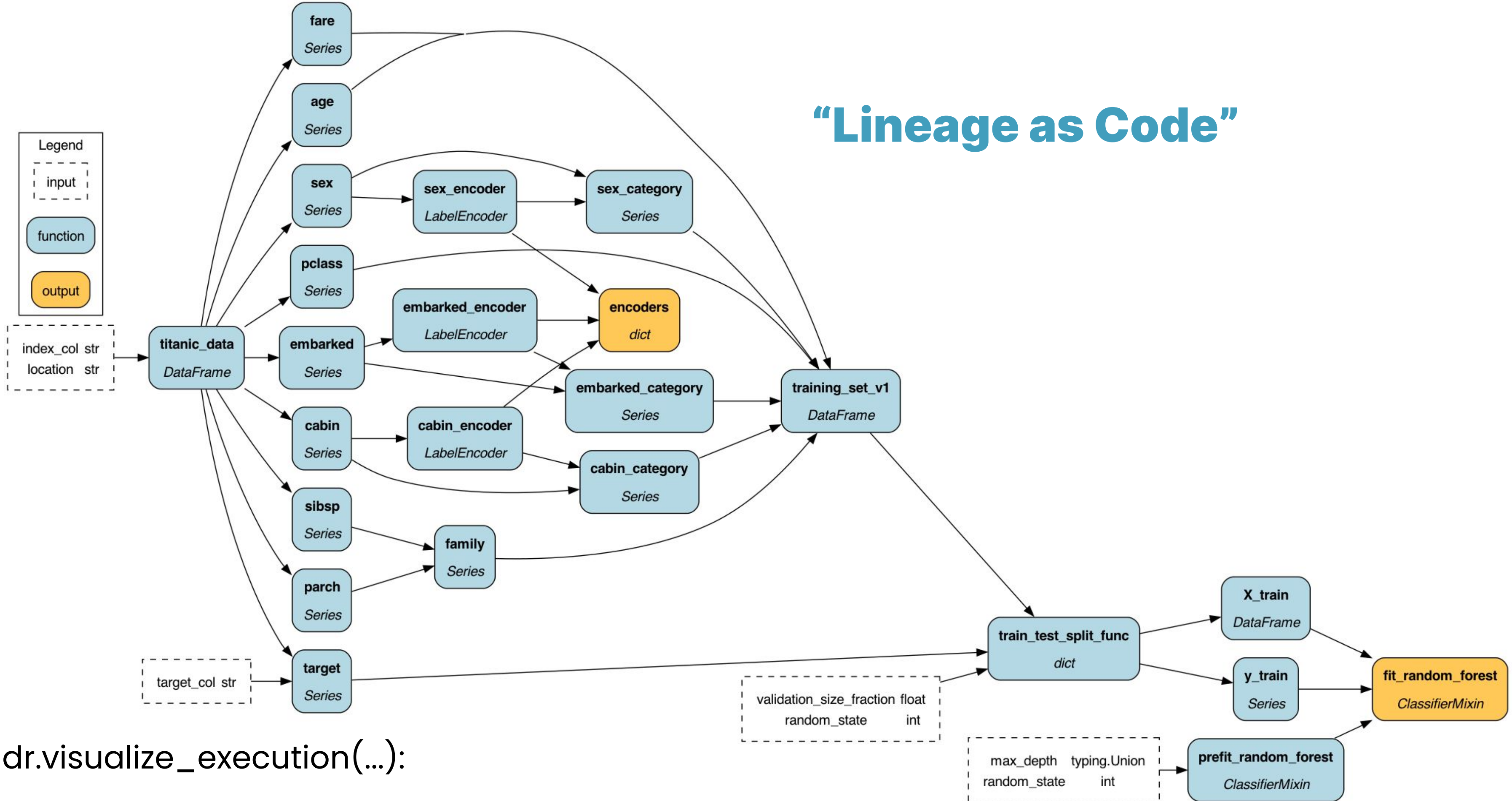
Data Quality Tests: runtime checks via annotation*; Pandera supported.

Self-documenting: naming, doc strings, annotations, & visualization

Scale: all these enable you to scale the team & code.

Visualization is first class

"Lineage as Code"



dr.visualize_execution(...):

General: Deployment & Reuse

```python
# client_features.py

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

This code is runnable everywhere python runs:

- Jupyter Notebooks, Python Scripts, Airflow, Ray, PySpark, web-services

→ **Can share features definitions in multiple contexts**

See https://blog.dagworks.io/p/feature-engineering-with-hamilton
https://blog.dagworks.io/p/expressing-pyspark-transformations

Comparing to the code from earlier:

```
df = loader.load_actuals(dates) # e.g. spend, signups
if config['region'] == 'UK':
   df['holidays'] = is_uk_holiday(df['year'], df[' week'])
else:
   df['holidays'] = is_holiday(df['year'], df['week'])
df['avg_3wk_spend'] = df['spend'].rolling(3).mean()
df['acquisition_cost'] = df['spend'] / df['signups']
df['spend_shift_3weeks'] = df['spend'].shift(3)
df['special_feature1'] = compute_bespoke_feature(df)
df['spend_b'] = multiply_columns(df['acquisition_cost'], df['B'])
save_df(df, "some_location")
```

👎 : testing, documentation, visualization, lineage, portability, ...

Comparing to the code from earlier:

```python
@extract_columns("year", "week", "spend", "signups", "col_a")
@check_output(schema=..., target_="load_actuals")
def load_actuals(dates: list) -> pd.DataFrame:
    """Loads the actual data for given dates."""
    return loader.load_actuals(dates)

@config.when(country="UK")
def holidays__uk(year: pd.Series, week: pd.Series) -> pd.Series:
    """UK holiday feature."""
    return _is_uk_holiday(year, week)

@config.when(country="US")
def holidays__us(year: pd.Series, week: pd.Series) -> pd.Series:
    """US holiday feature."""
    return _is_holiday(year, week)

def avg_3wk_spend(spend: pd.Series) -> pd.Series:
    """Calculates the rolling 3-week average spend. 3 is important because…"""
    return spend.rolling(3).mean()
```

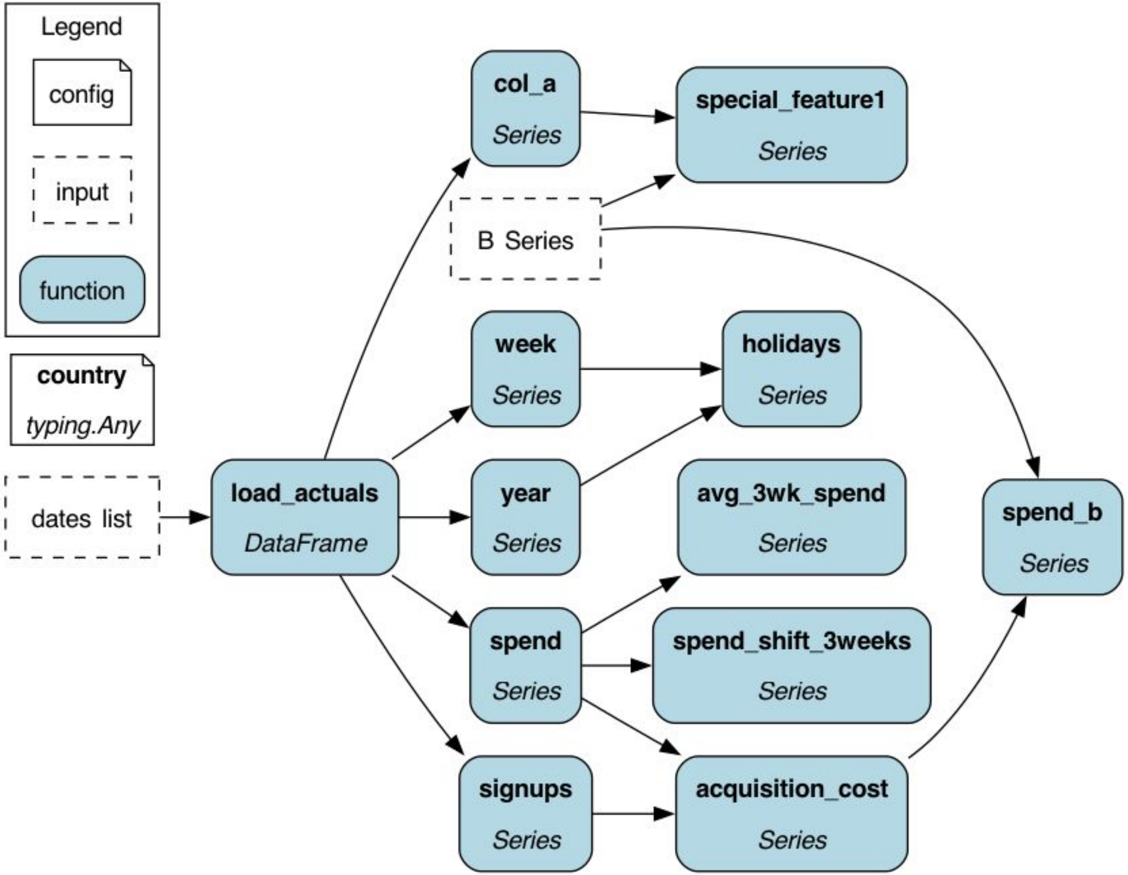
Comparing to the code from earlier:

```
def acquisition_cost(spend: pd.Series, signups: pd.Series) -> pd.Series:
    """Calculates the acquisition cost."""
    return spend / signups

def spend_shift_3weeks(spend: pd.Series) -> pd.Series:
    """Shifts the spend by 3 weeks."""
    return spend.shift(3)

def special_feature1(col_a: pd.Series, B: pd.Series) -> pd.Series:
    """Computes a bespoke feature."""
    return _compute_bespoke_feature(col_a, B)

def spend_b(acquisition_cost: pd.Series, B: pd.Series) -> pd.Series:
    """Multiplies acquisition cost with column B."""
    return _multiply_columns(acquisition_cost, B)
```

Comparing to the code from earlier:

Notes:

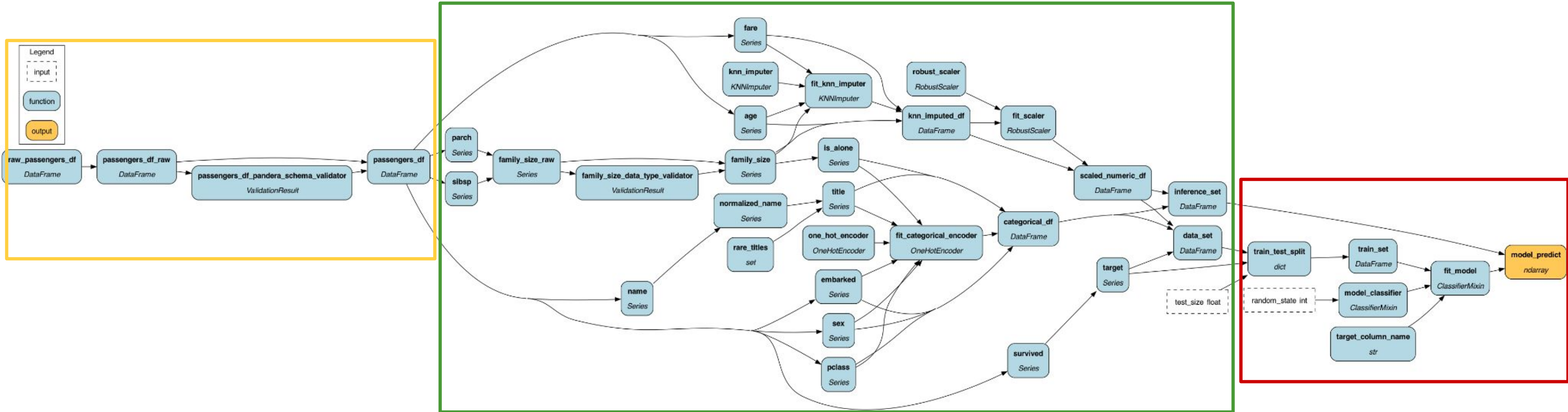
- Unit testable
- Documentation friendly
- Lineage is clear
- Visualization →
- Reusable code
- Simpler to maintain



Benefit: can model whole ML/LLM Pipeline too

Can group functions into modules, e.g.:

1. Data loading & preprocessing
2. Feature engineering
3. Model fitting



Recap of this Talk

1. ~~Motivating pain~~
2. ~~Hamilton~~
3. ~~General Usage~~
4. ~~Native SWE~~
5. **Summary**

TL;DR: Summary - F.E. with Hamilton

1. Hamilton is a lightweight library to declaratively express transforms
    - Great for feature engineering!
    - Write code that people aren't terrified of inheriting!
2. The Hamilton paradigm: ⬆ SWE Best Practices ⬆ value of your work
    - Understand features: naturally testable & documentation friendly functions with lineage.
    - Reuse features: naturally reusable and modular code so you can move faster.
    - Standardized way to iterate and add to a code base.
3. Can integrate anywhere that python runs
    - Develop in a notebook, deploy on PySpark, reuse in a web-service.
    - Can help DS & Engineering teams collaborate more efficiently

What I’m building on top of Hamilton

With a one-line code change you get:

- Versioning (code)
- Lineage (code & artifacts)
- Catalog (code & artifacts)
- Observability (code & data)

DAGWORKS

Sign up for free @ www.dagworks.io

Fin. Thanks for listening!

> `pip install sf-hamilton` or ⌨️ on tryhamilon.dev

Questions?

⭐ Star us please: https://github.com/dagworks-inc/hamilton

📣 Join us on on Slack or subscribe to blog.dagworks.io!

📚 Documentation: hamilton.dagworks.io

🎓 Self-paced tutorial https://github.com/DAGWorks-Inc/hamilton-tutorials/tree/main/2023-10-09

🛣️ Follow us: https://twitter.com/hamilton_os

👉 https://www.dagworks.io (sign up! We're building on top of Hamilton!)

📯 https://twitter.com/stefkrawczyk ⛓️ https://www.linkedin.com/in/skrawczyk/