Hamilton: *Natively* bringing software engineering best practices to python data transformations

Stefan Krawczyk	CEO; Ex-Stitch Fix, Ex-Nextdoor, Ex-LinkedIn
Elijah ben Izzy	CTO; Ex-Stitch Fix, Ex-Two Sigma
DAGWorks (YCW23)	

TL;DR

Q: Doing data transforms in python?
A: Hamilton might be a fit for you!

`pip install` **`sf-hamilton`**

Get started in <15 minutes!
https://hamilton-docs.gitbook.io/

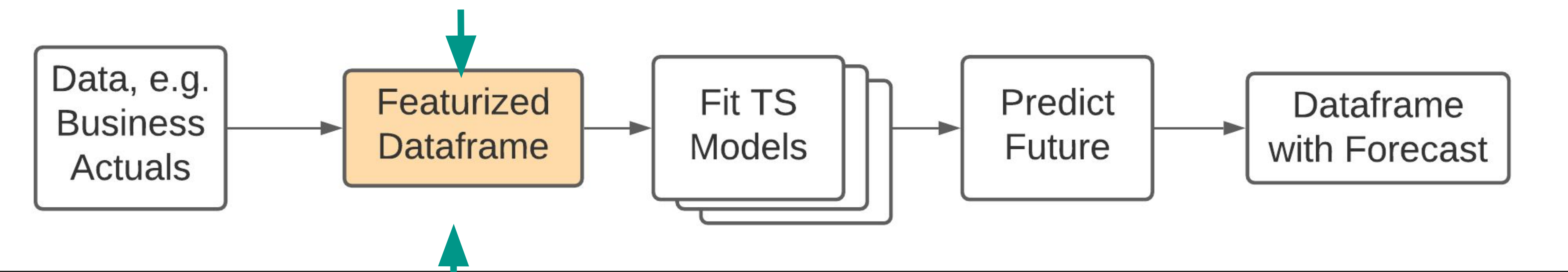
The Agenda

A motivating story of DS pain
The solution: *Hamilton*
Hamilton @ Stitch Fix
General Usage
Native SWE: Problems & how Hamilton helps
Summary
OS Roadmap

Backstory: an old model at Stitch Fix

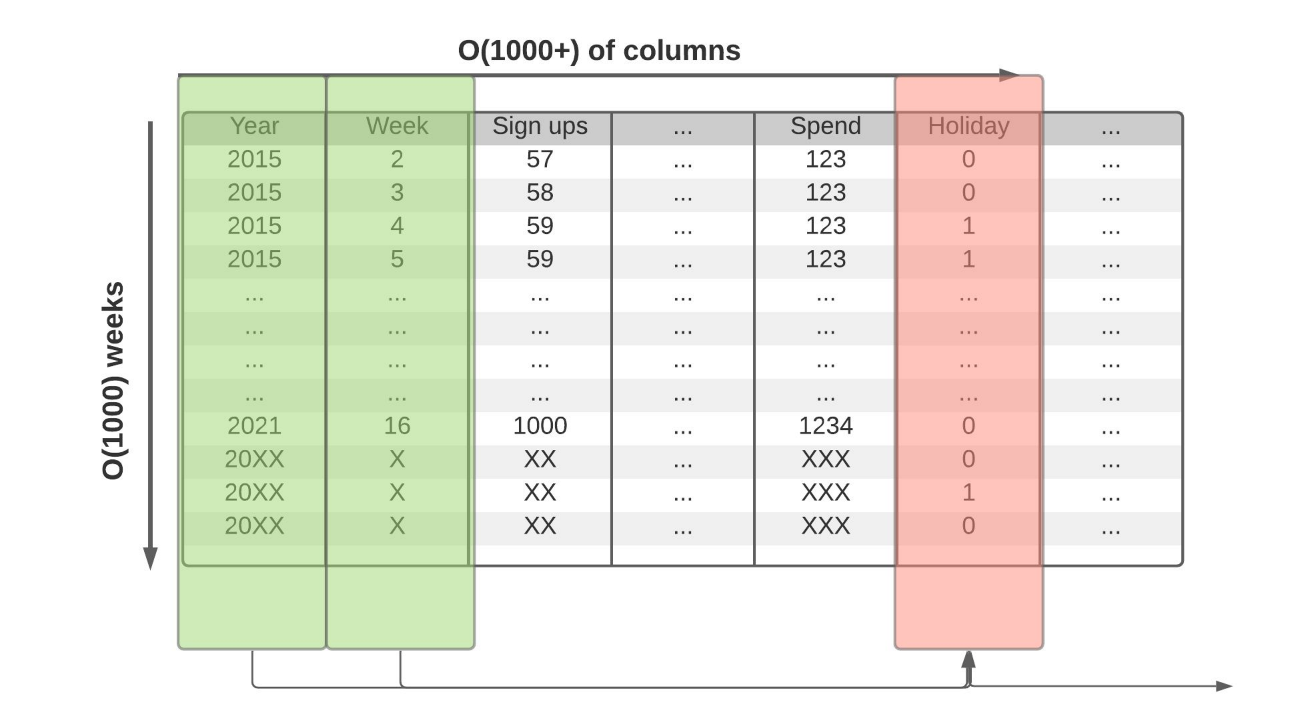
Forecasting the business (demand, signups, churn)

Biggest problems here



What Hamilton helped solve!

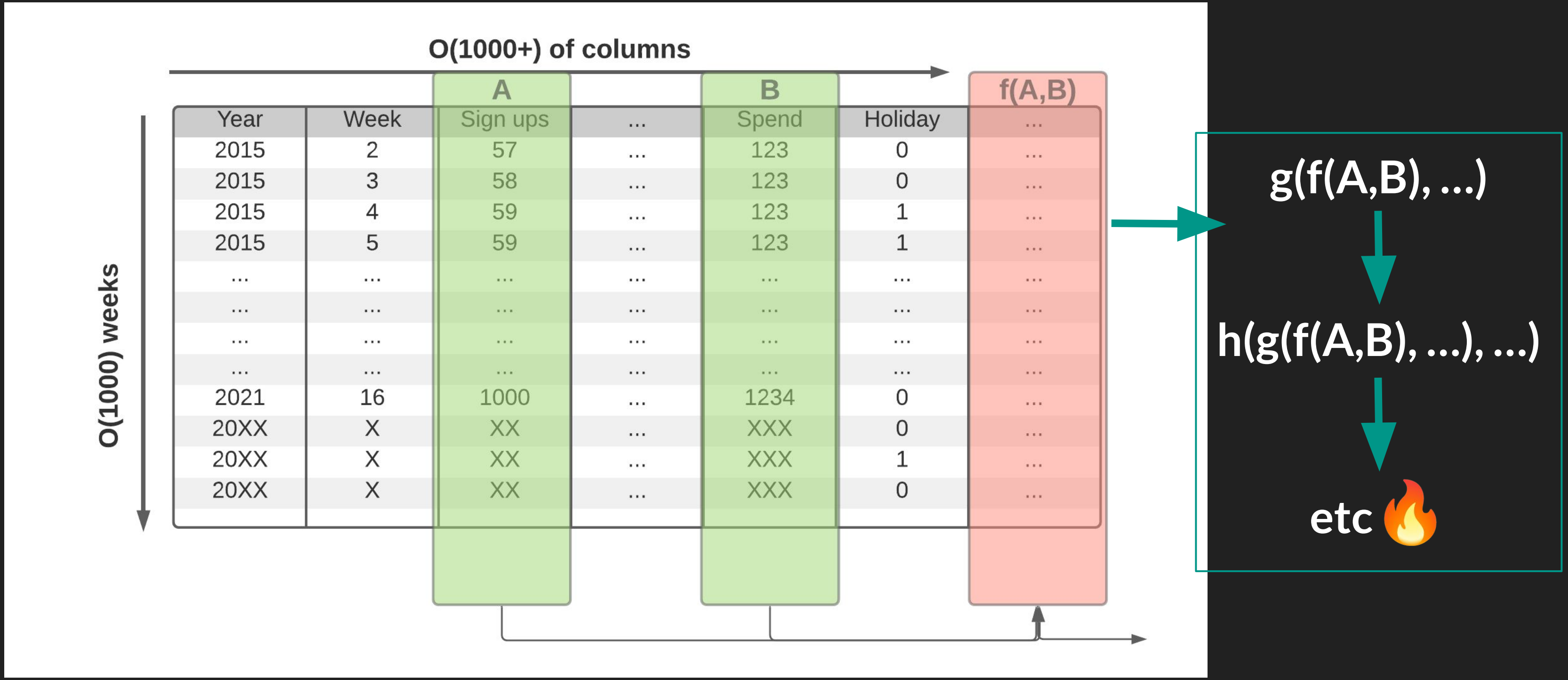
Backstory: TS –> Dataframe creation



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	...
20XX	X	XX	...	XXX	0	...
20XX	X	XX	...	XXX	1	...
20XX	X	XX	...	XXX	0	...

Columns are functions of other columns

Backstory: TS –> Dataframe creation



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	...
20XX	X	XX	...	XXX	0	...
20XX	X	XX	...	XXX	1	...
20XX	X	XX	...	XXX	0	...

Backstory: Creating training table

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

Backstory: Creating training table

```
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'])
```

Backstory: Creating training table

```
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)
```

Backstory: Creating training table

```
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'])
```

Backstory: Creating training table

```
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")
```

Backstory: Creating training table

```
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 scale this code to 1000+ columns & a growing team 😬

<u>Problem</u>: 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 scale this code to 1000+ columns & a growing team 😬

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 scale this code to 1000+ columns & a growing team 😬

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 scale this code to 1000+ columns & a growing team 😬

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 scale this code to 1000+ columns & a growing team

😬

Problem: onboarding 📈

```python
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 scale this code to 1000+ columns & a growing team 😬

Problem: 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 scale this code to 1000+ columns & a growing team
😬

Backstory: an old model at Stitch Fix

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 Stitch Fix, time-series forecasting, or even pandas

Questions for you!

1. Are any of these pains familiar to you? If so, which ones?
2. Do you have some other pains related to modeling?

✋ Raise hand | Unmute !

The Agenda

A motivating story of DS pain
The solution: *Hamilton*
Hamilton @ Stitch Fix
General Usage
Native SWE: Problems & how Hamilton helps
Summary
OS Roadmap

Hamilton: the “A-ha” Moment

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 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
```

(driver code not shown, also Hamilton is python type agnostic)

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

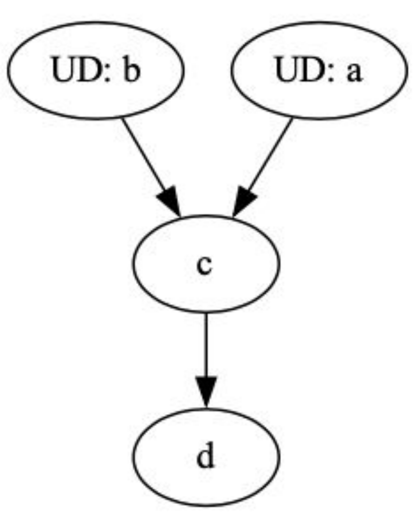
You declare

```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
```

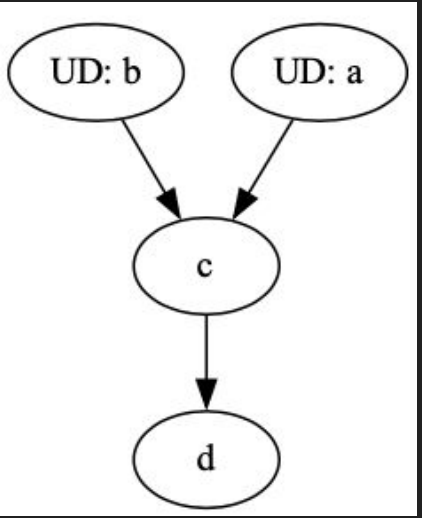


Driver says what/when to execute

```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)
```

But Wait, There's More...!

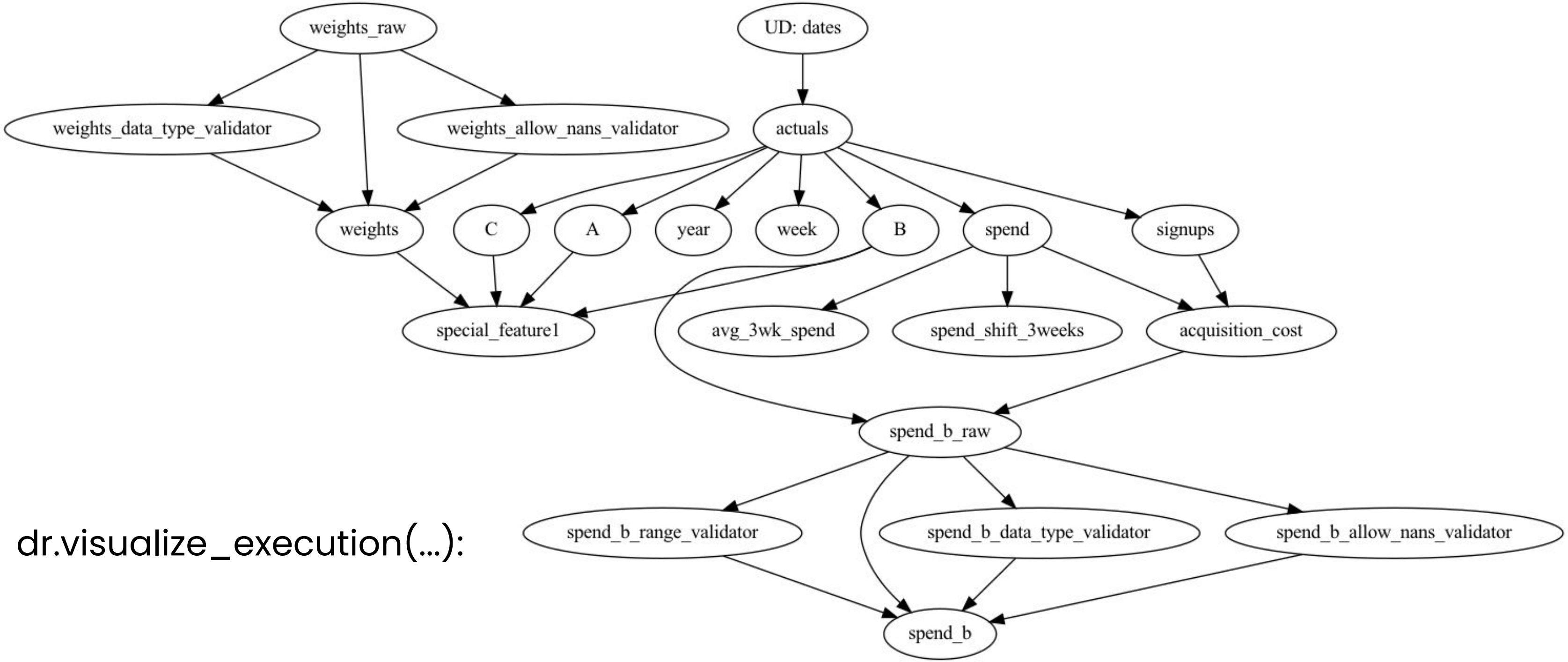
Q: Doesn't Hamilton make your code more verbose?

A: Yes, but 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`
- ❑ `@extract_outputs # one dict -> multiple outputs`
- ❑ `@check_output # data validation; very lightweight`
- ❑ `@config.when # conditional transforms`
- ❑ *`@... # new ones often`*

And then there's visualization: e.g.

dr.visualize_execution(…):



The Agenda

A motivating story of DS pain
The solution: *Hamilton*
Hamilton @ Stitch Fix
General Usage
Native SWE: Problems & how Hamilton helps
Summary
OS Roadmap

Hamilton @ Stitch Fix

Running in production for **3+** years

Initial use-case grew to manage **4000+** feature definitions

Data science teams ❤️ it

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

The Agenda

A motivating story of DS pain
The solution: *Hamilton*
Hamilton @ Stitch Fix
General Usage
Native SWE: Problems & how Hamilton helps
Summary
OS Roadmap

General usage of Hamilton

What is Hamilton good for?

- Anyone having to deal with a lot of transforms
  - Time-series feature engineering (origin)
  - Tired of managing scripts that do transformations…
- Code & software best practices enthusiasts
- *Still scratching the surface here*!
  - E.g. Can logically model a lot of problems, and decide later how to materialize it.

What is Hamilton *not* good for?

- “Dynamic DAGs” that change what should be computed based on the output of the prior step.

Overview: General usage of Hamilton

1. Create functions in module(s).
2. Create drivers to drive execution of those functions.
3. Execute driver code.

Notes:

- Can model ***any*** *python object creation* (not just pandas), e.g. ML flows.
- **Batch**: use Hamilton within Airflow (et al), Jupyter notebook etc.
- **Online**: embed within python streaming / python web services

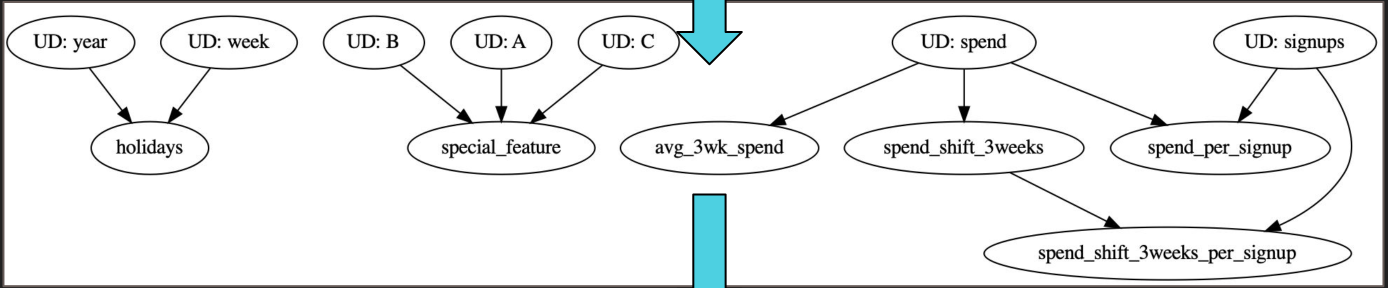
Modeling e.g. featurization

Data loading &
Feature code:

```
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:

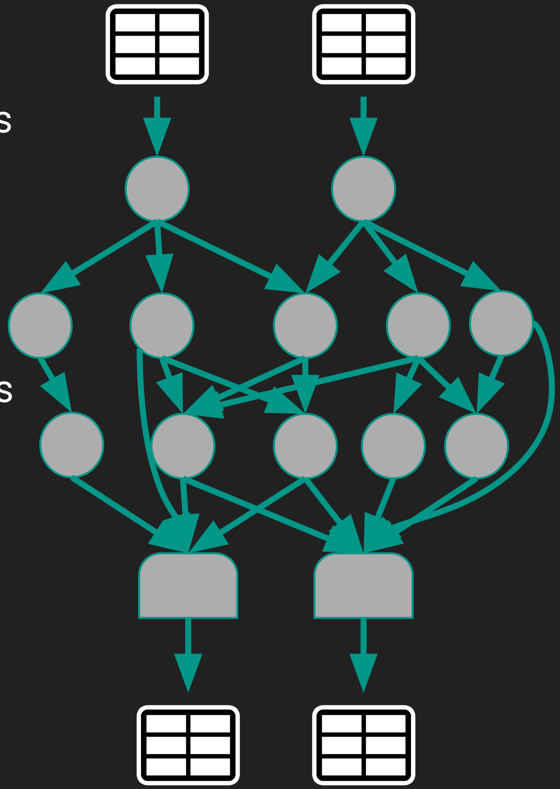
Year	Week	Sign ups	...	Spend	Holiday
2015	2	57	...	123	0
2015	3	58	...	123	0
2015	4	59	...	123	1
2015	5	59	...	123	1
...	...	...	...	...	...
...	...	...	...	...	...
...	...	...	...	...	...
...	...	...	...	...	...
2021	16	1000	...	1234	0

run.py

Modeling e.g. featurization

Code that needs to be written:

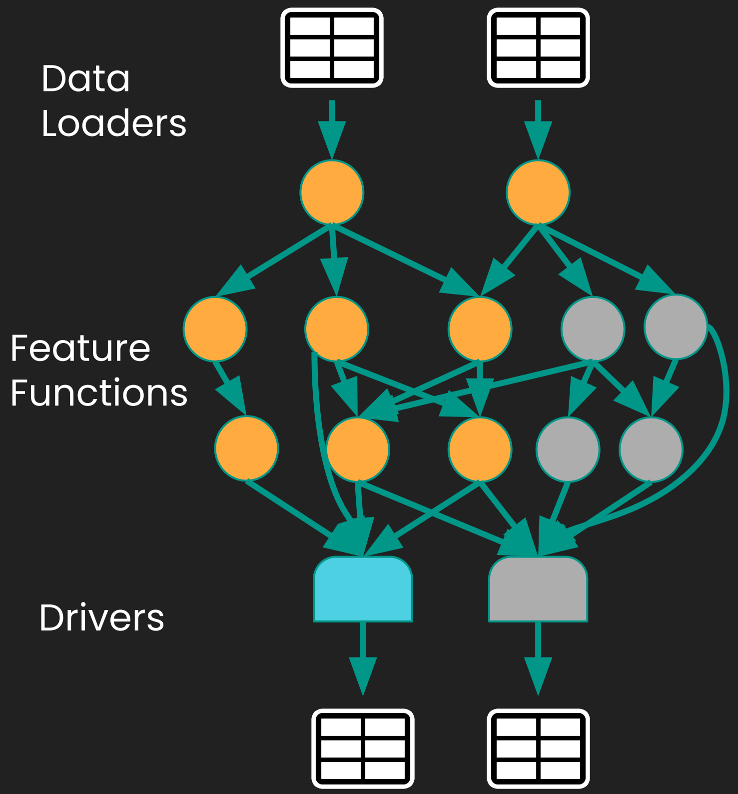
1. Functions to load data
   a. normalize/create common index to join on
2. Feature functions
   a. Optional: model functions.
3. Drivers materialize data
   a. DAG is walked for only what's needed.



Modeling e.g. featurization

Code that needs to be written:

1. Functions to load data
   a. normalize/create common index to join on
2. Feature functions
   a. Optional: model functions.
3. Drivers materialize data
   a. DAG is walked for only what's needed.



The Agenda

A motivating story of DS pain
The solution: *Hamilton*
Hamilton @ Stitch Fix
General Usage
Native SWE: Problems & how Hamilton helps
Summary
OS Roadmap

Native SWE: Problems with Python transform Code

> Human/Team:

- Highly coupled code
- In ability to reuse/understand work
- Broken/unhealthy production pipelines

} **Hamilton** helps here!

> Machines:

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

} **Hamilton** has integrations here!

Native SWE: Scaling Humans/Teams

Hamilton Functions:

```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: pd.Series) -> pd.Series:
    """Zero mean unit variance value of height"""
    return height_zero_mean / height_std_dev
```

Hamilton Features:

- Unit testing ✅ always possible
- Documentation ✅ tags, visualization, function doc
- Modularity/reuse ✅ module curation & decoupled drivers; extensibility & decorators
- Central definition store (in code) ✅ naming, curation, versioning
- Data quality ✅ runtime checks

Example: @config - encapsulation of logic

Before

```
if config['region'] == 'UK':
    df['holidays'] = ...
else:
    df['holidays'] = ...
```

After

```
@config.when(region="US")
def holidays__us(dep1: pd.Series, dep2: str) -> pd.Series:

@config.when(region="UK")
def holidays__uk(dep1: pd.Series, other_dep: str) -> pd.Series:
```

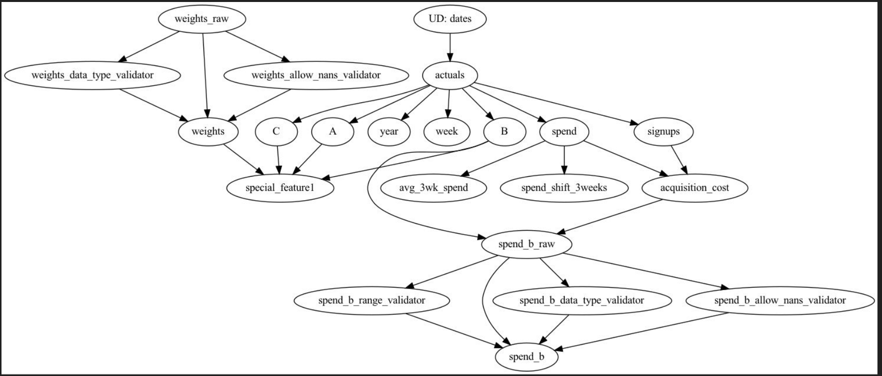
Example: Documentation

Before

```
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")
```

- Discovery of what's there?
- Who owns things?
- Where is the code that created this output?
- Where do I start?
- Onboarding/Offboarding

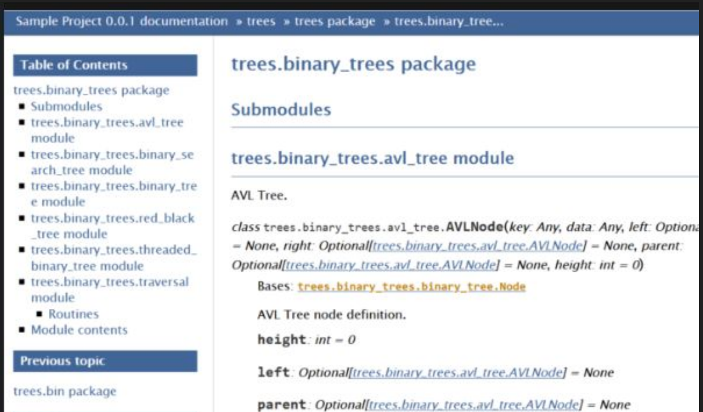
Example: Documentation



After

```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: pd.Series) -> pd.Series:
    """Zero mean unit variance value of height"""
    return height_zero_mean / height_std_dev
```

- Module name
- @tag & @check_output
- Function & parameter names
- Function doc strings → sphinx docs
- 1-1 output to function mapping



Example: data quality

Before

1. Execute code to create data
2. Run data through various tests
3. If error, find code to debug …

Updates:

1. Update code, forget to update data tests.
2. Run data through various tests
3. If error, update test.

After (shift left)

1. Put expectation on function
2. Execute code – error / warn.
3. If error, know exactly where in your code to start debugging from

Updates:

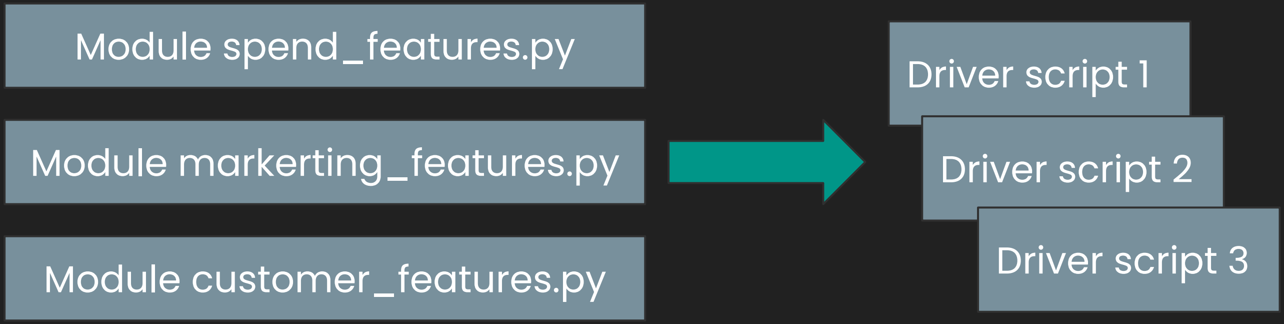
1. Update code and update expectation in same PR!

```
@check_output(schema=...)
def height_feature(...) -> pd.Series:
    # some logic
```

Native SWE: Scaling Humans/Teams

Code base implications:

1. Functions are always in modules
2. Driver script, i.e execution script, is decoupled from functions.



> Code reuse from day one!
> Low maintenance to support many driver scripts

Example: driver contexts - decoupling concerns

Before

```
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")
```

Easy to couple:

1. Where data comes from.
2. Logic to process it.
3. Different concerns because of code inertia - “just append”.

Hard to reuse logic.

After

logic_modules*.py

```
def avg_3wk_spend(spend: pd.Series) -> pd.Series:

@config.when(region="US")
def holidays(dep1: pd.Series, dep2: str) -> pd.Series:
```

us_driver.py

uk_driver.py

Hard to couple:

1. Where data comes from.
2. Different needs in the same code.

Easy to add new contexts and reuse existing logic.

Native SWE: Scaling Compute/Data with Hamilton

Hamilton has the following integrations out of the box:

- Ray
  - Single process -> Multiprocessing -> Cluster
- Dask
  - Single process -> Multiprocessing -> Cluster
- Pandas on Spark
  - Uses enables using Pandas Spark API with your Pandas code easily
- Switching to run on Ray/Dask/Pandas on Spark requires:

  - **Only changing driver.py code***
    - Pandas on Spark also needs changing how data is loaded.

Native SWE? *Decoupling of dataflow from execution.*

Hamilton + Ray/Dask/Spark: Driver only change

```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')
```

Hamilton + Ray: Driver only change

```
# 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()
```

Hamilton + Dask: Driver only change

```python
# 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()
```

Hamilton + Spark: Driver change + loader

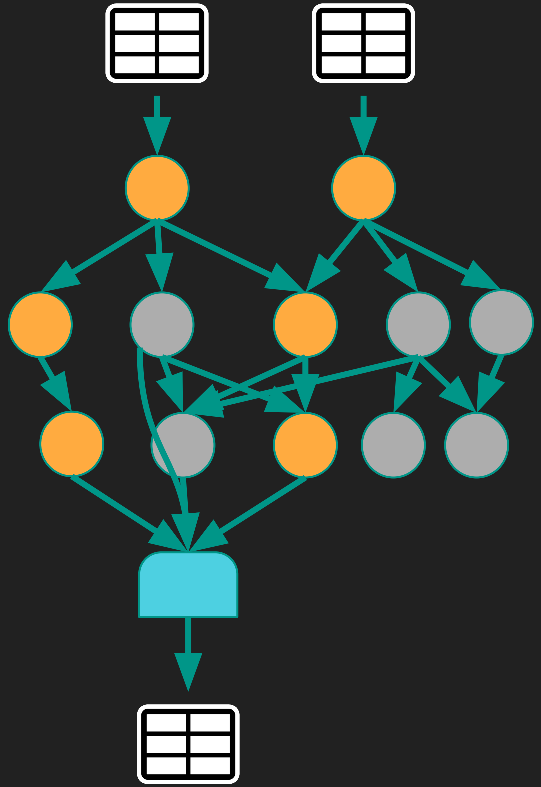
```
# run_on_pandas_on_spark.py
…
import pyspark.pandas as ps
from hamilton import base, driver
from hamilton.experimental import h_spark
…
spark = SparkSession.builder.getOrCreate()
ps.set_option(...)
config = {...}
skga = h_dask.SparkKoalasGraphAdapter(spark, spine='COLUMN_NAME',
    result_builder=base.PandasDataFrameResult())
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

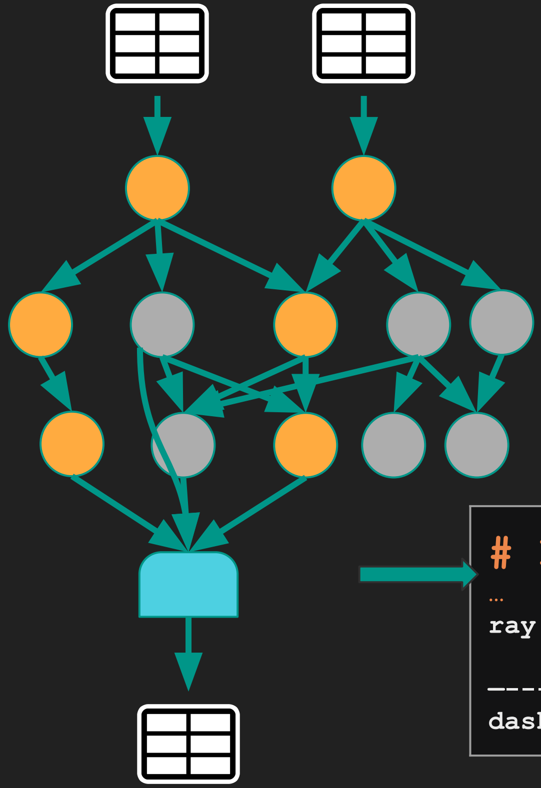


Hamilton + Ray/Dask: How does it work?

```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



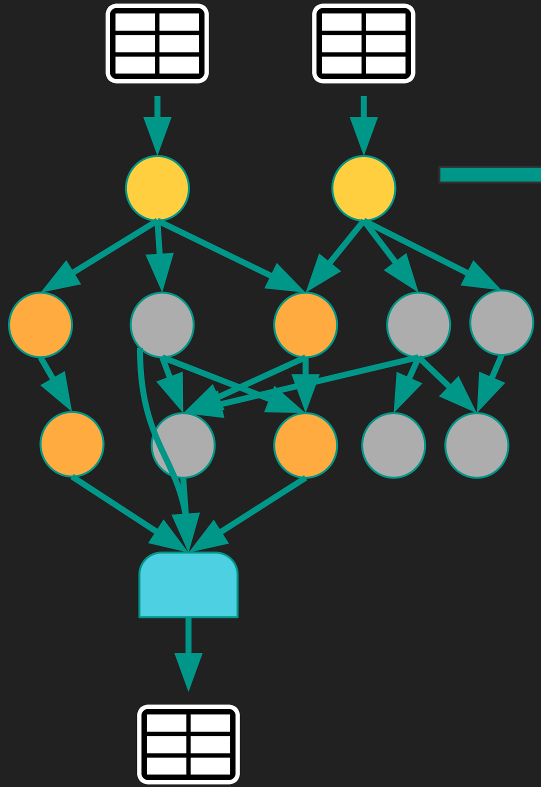
```python
# Delegate to Ray/Dask
…
ray.remote(
    node.callable).remote(**kwargs)
--------------------
dask.delayed(node.callable)(**kwargs)
```

Hamilton + Spark: How does it work?

```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



```
# 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

Things to think about:

1. Serialization:
    a. Hamilton defaults to serialization methodology of these frameworks.
2. Memory:
    a. Defaults should work. But fine tuning memory on a “function” basis is not exposed.
3. Python dependencies:
    a. You need to manage them.
4. Looking to graduate these APIs from ***experimental status***

>> Looking for contributions here to extend support in Hamilton! <<

Otherwise `**modin**` is also an option – but requires changing imports.

Native SWE - How Hamilton Helps: Summary

Hamilton forces you to write transforms as python functions.

These python functions provide everything you need:

- ❏ **Unit testing**: *simple – plain python functions!*
- ❏ **Documentation**: *use the docstring & create visualizations*
- ❏ **Modularity**: *Small pieces -> by definition*
- ❏ **Catalog**: via *Code -> "definition store"*
- ❏ **Debugging**: Methodical
- ❏ **Trustworthy data**: *Validation included out of the box with @check_output*

Decorators → powerful, higher-order operations (didn't cover here)

Driver → decouple transform definition from execution

The Agenda

A motivating story of DS pain
The solution: *Hamilton*
Hamilton @ Stitch Fix
General Usage
Native SWE: Problems & how Hamilton helps
Summary
OS Roadmap

Summary:
Hamilton natively brings SWE best practices

- Hamilton is a declarative paradigm to describe data/feature transformations
  - Embeddable anywhere that runs python.
- It grew out of a need to tame a feature (i.e. transform) code base
  - it'll make yours better too!
- The Hamilton paradigm scales humans/teams through software engineering best practices that come naturally.
- **Hamilton** paired with a system (e.g. modin, ray, etc) enables one to:

scale humans/teams **and** *scale data/compute.*

The Agenda

A motivating story of DS pain
The solution: *Hamilton*
Hamilton @ Stitch Fix
General Usage
Native SWE: Problems & how Hamilton helps
Summary
OS Roadmap

OS Progress

Early stages, but thriving community

- ❏ Being used in production in multiple companies (see below)
- ❏ ⭐ 800+ stars on github

Looking for

- ❏ Contributors
- ❏ Bug hunters
- ❏ User feedback

IBM – UK Govt. Digital Services – British Cycling Team – Transfix – Pacific Northwest National Laboratories – Stitch Fix – ...

Our Vision

The connecting layer that makes it easy to connect with:

Connect with orchestration frameworks

Integrate with data quality vendors/OS options

Integrate loading from a variety of upstream sources

SQL support (+duckdb)

Roadmap

More Dataframe support

- ❏ Polars
- ❏ Better integration with PySpark UDFs

New decorators

- ❏ Reuse sub-dag (pushed), e.g. compute grains.
- ❏ More natural SQL support (WIP)

Execution related

- ❏ Profiling
- ❏ Caching
- ❏ <Your idea here!>

Give Hamilton a Try!
We'd love your Feedback

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

⭐ on github (https://github.com/stitchfix/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?

https://twitter.com/hamilton_os

https://github.com/stitchfix/hamilton

https://hamilton-docs.gitbook.io/

https://twitter.com/stefkrawczyk

https://www.linkedin.com/in/skrawczyk/

https://www.dagworks.io (sign up!)