apply(meetup)

An ML Data Engineering Community Event

presented by tecton

[Open Source] Hamilton, a micro framework for creating dataframes, and its application at Stitch Fix

Stefan Krawczyk, Mgr. Model Lifecycle Platform, Stitch Fix

What to keep in mind for the next ~30 minutes?

1. Hamilton is a new paradigm to create dataframes*.

2. Using Hamilton is a productivity boost for teams.

3. It's open source - join us on:
   Github: https://github.com/stitchfix/hamilton
   Discord: https://discord.gg/wCqxqBqn73

* in fact, any python object really.

Talk Outline:

> Backstory: who, what, & why

Hamilton

Hamilton @ Stitch Fix

Pro tips

Extensions

Backstory: who
Forecasting, Estimation, & Demand (FED)Team

- Data Scientists that are responsible for forecasts that help the business make operational decisions.
  - e.g. staffing levels
- One of the oldest teams at Stitch Fix.

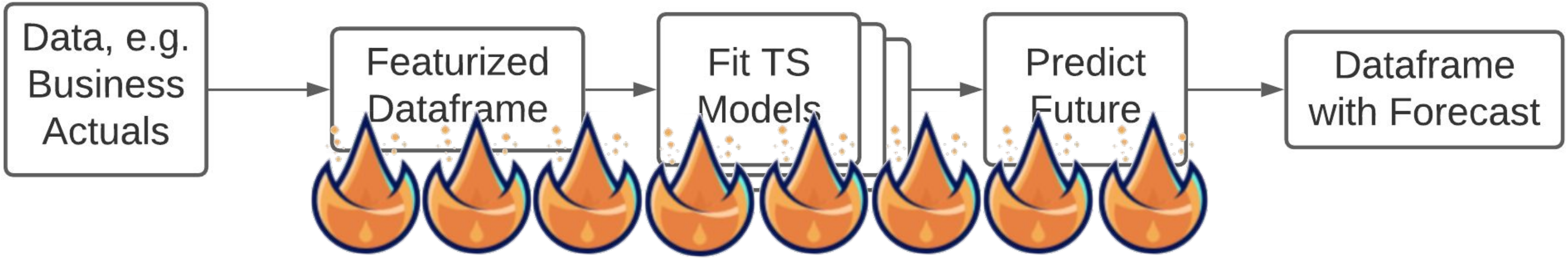
Backstory: what

Forecasting, Estimation, & Demand (FED)Team

FED workflow:

Backstory: what
Forecasting, Estimation, & Demand Team

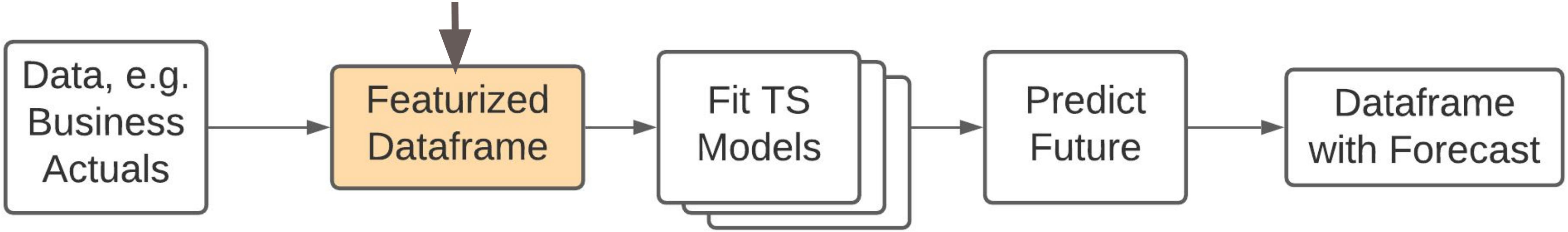
FED workflow:  +  == 



Backstory: what

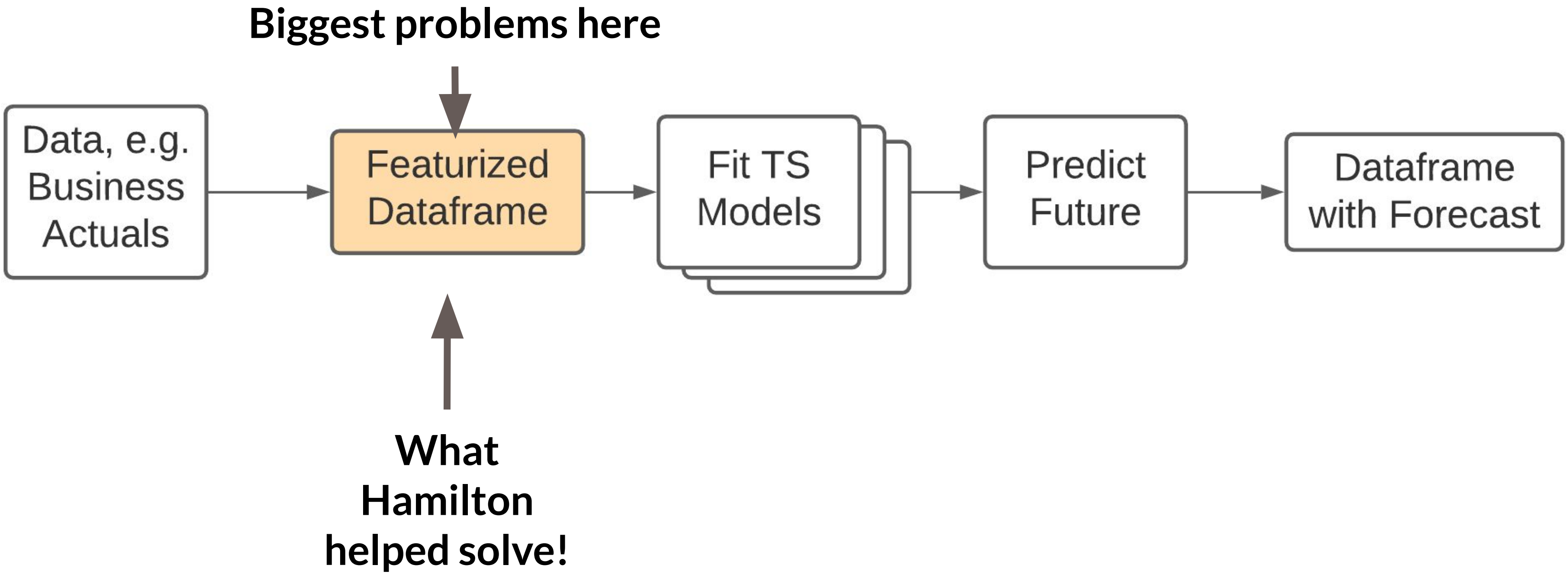
Creating a dataframe for time-series modeling.

Biggest problems here



Backstory: what

Creating a dataframe for time-series modeling.



Backstory: why

What is this dataframe & why is it causing 🔥 ?

O(1000+) of columns

O(1000) weeks

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

(not big data)

Backstory: why

What is this dataframe & why is it causing 🔥 ?

O(1000+) of columns

O(1000) weeks

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: why

What is this dataframe & why is it causing 🔥 ?

O(1000+) of columns

O(1000) weeks

		A		B		f(A,B)
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:

g(f(A,B), ...)

h(g(f(A,B), ...), ...)

etc 🔥

Backstory: why
Featurization: some example code

```python
df = load_dates()  # load date ranges
df = load_actuals(df)  # load actuals, e.g. spend, signups
df['holidays'] = is_holiday(df['year'], df['week'])  # holidays
df['avg_3wk_spend'] = df['spend'].rolling(3).mean()  # moving average of spend
df['spend_per_signup'] = df['spend'] / df['signups']  # spend per person signed up
df['spend_shift_3weeks'] = df.spend['spend'].shift(3)  # shift spend because ...
df['spend_shift_3weeks_per_signup'] = df['spend_shift_3weeks'] / df['signups']

def my_special_feature(df: pd.DataFrame) -> pd.Series:
    return (df['A'] - df['B'] + df['C']) * weights

df['special_feature'] = my_special_feature(df)
# ...
```

Backstory: why

Featurization: some example code

```
df = load_dates()  # load date ranges
df = load_actuals(df)  # load actuals, e.g. spend, signups
df['holidays'] = is_holiday(df['year'], df['week'])  # holidays
df['avg_3wk_spend'] = df['spend'].rolling(3).mean()  # [redacted] spend
df['spend_per_signup'] = df['spend'] / df['[redacted] signed up
df['spend_shift_3weeks'] = df.spend[[redacted] spend because ...
df['spend_shift_3weeks[redacted]_3weeks'] / df['signups']

def my_spec[redacted] pd.DataFrame) -> pd.Series:
    return ([redacted]] - df['B'] + df['C']) * weights

df['special_feature'] = my_special_feature(df)
# ...
```

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

Backstory: why

```
df = load_dates()  # load date ranges
df = load_actua
df['holidays']
df['avg_3wk_spe
df['spend_per_s
df['spend_shift
df['spend_shift

def my_special_
    return (df['

df['special_feature'] = my_special_feature(df)
# ...
```

Scaling this type of code results in the following:

- **lots of heterogeneity in function definitions & behaviors**
- **inline dataframe manipulations**
- **code ordering is super important**

- Testing / Unit testing 👎
- Documentation 👎
- Code Reviews 👎
- Onboarding 📈 👎
- Debugging 📈 👎

Backstory - Summary

Code for featurization == 🤯.

Talk Outline:

Backstory: who, what, & why

> Hamilton

The Outcome

Pro tips

Extensions

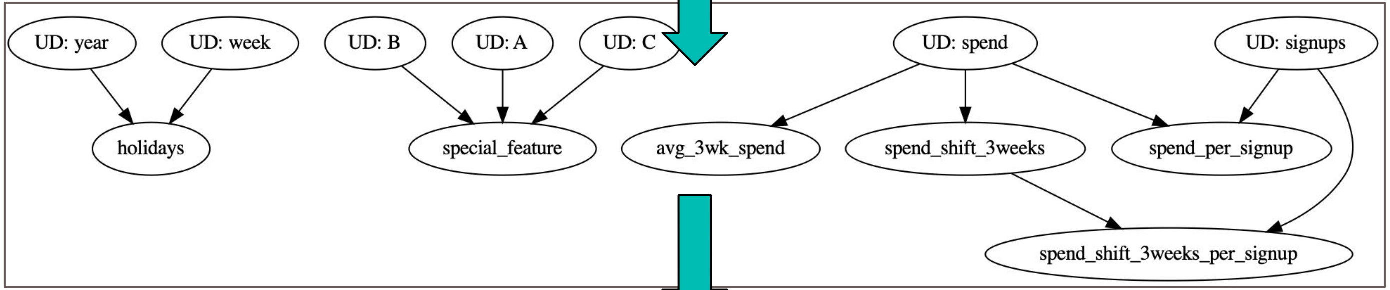
Hamilton: Code → Directed Acyclic Graph → DF

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
```

User

DAG:



Hamilton

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

User

Hamilton: a new paradigm

1. Write functions!
2. Function name

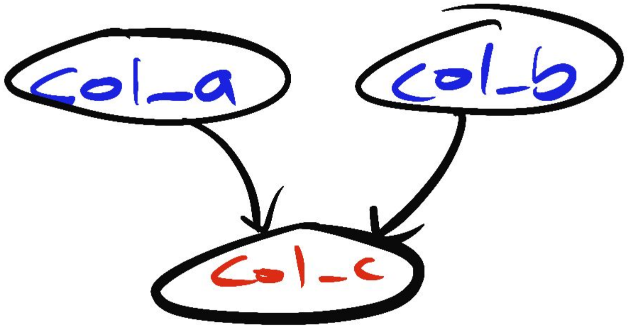
   == output column
3. Function inputs

   == input columns

Code:

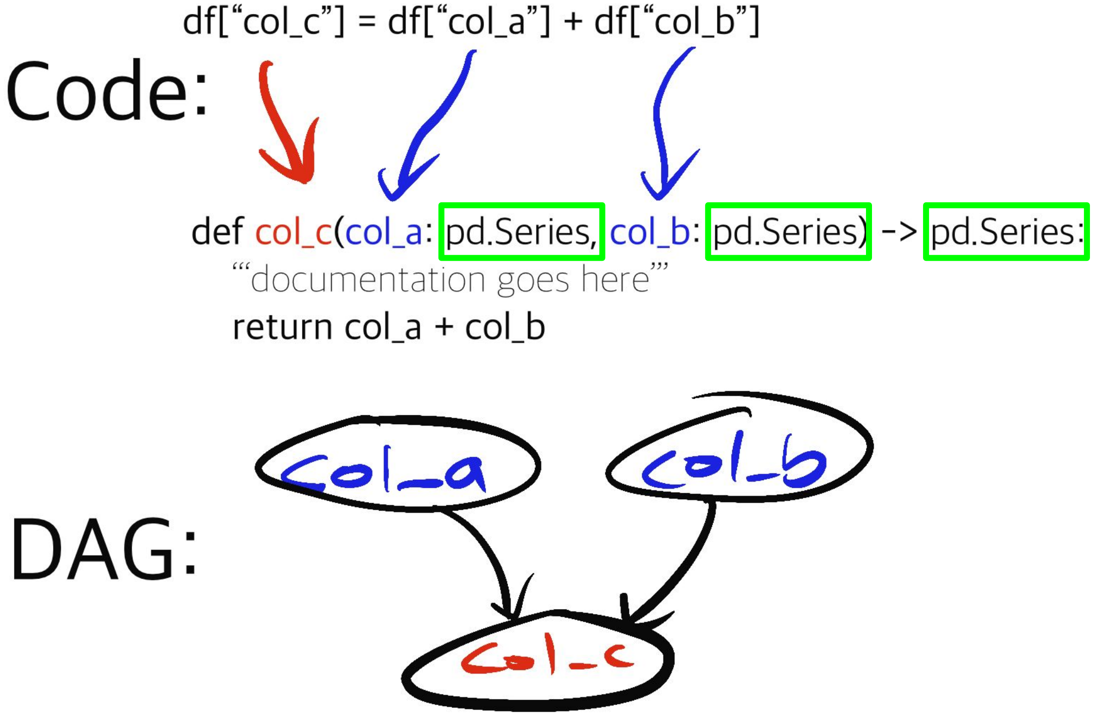
```
df[“col_c”] = df[“col_a”] + df[“col_b”]

def col_c(col_a: pd.Series, col_b: pd.Series) -> pd.Series:
    “““documentation goes here”””
    return col_a + col_b
```

DAG:



Hamilton: a new paradigm

4. Use type hints for typing checking.
5. Documentation is easy and natural.



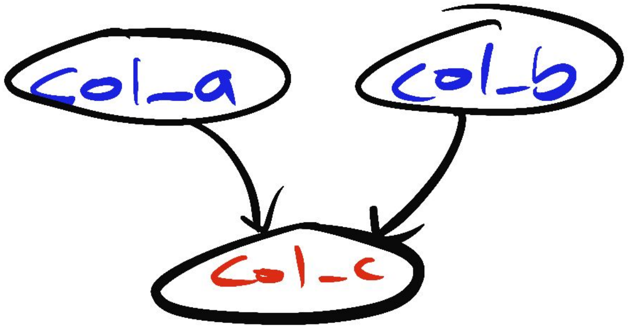
Hamilton: code to directed acyclic graph - how?

1. Inspect module to extract function names & parameters.
2. Nodes & edges + graph theory 101.

Code:

```
df["col_c"] = df["col_a"] + df["col_b"]

def col_c(col_a: pd.Series, col_b: pd.Series) -> pd.Series:
    """documentation goes here"""
    return col_a + col_b
```

DAG:



Hamilton: directed acyclic graph to DF - how?

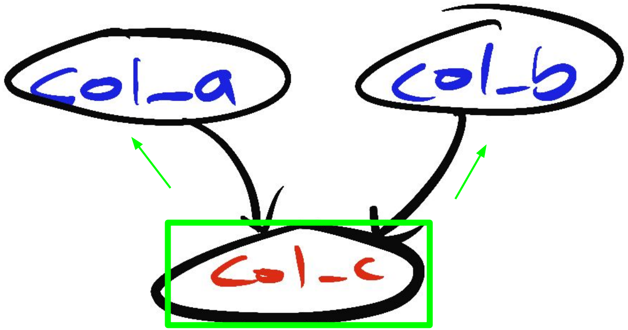
1. Specify outputs & provide inputs.
2. Determine execution path.
3. Execute functions once.
4. Combine at the end.

Code:

```
df["col_c"] = df["col_a"] + df["col_b"]
```

```
def col_c(col_a: pd.Series, col_b: pd.Series) -> pd.Series:
    """documentation goes here"""
    return col_a + col_b
```

DAG:

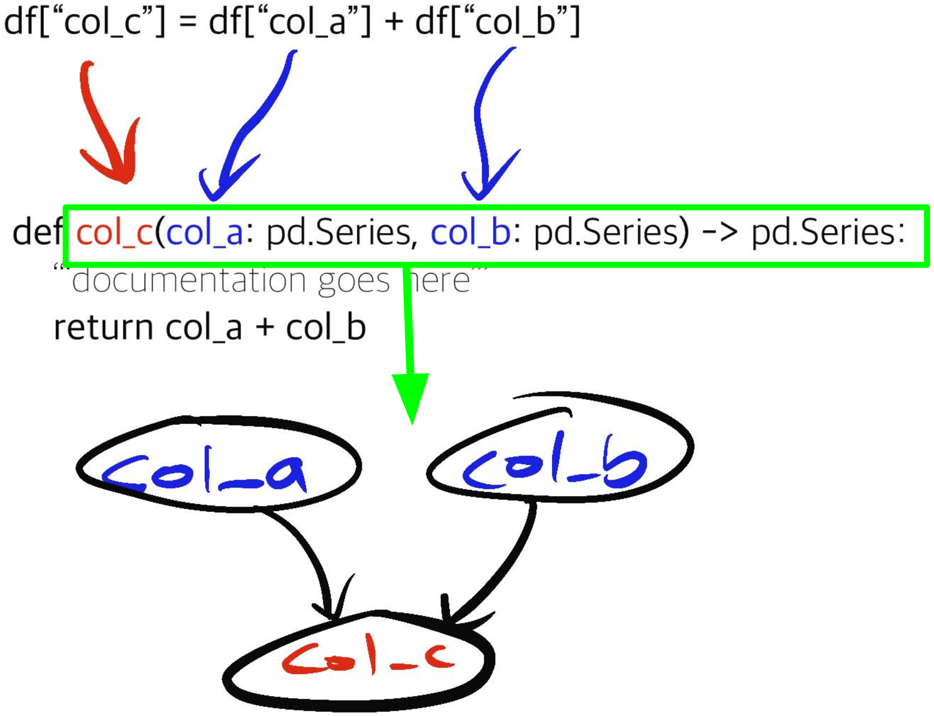


Hamilton: Key Point to remember (1)

Hamilton **requires**:

1. Function names
2. & Function parameter names

to match to stitch together a directed acyclic graph.

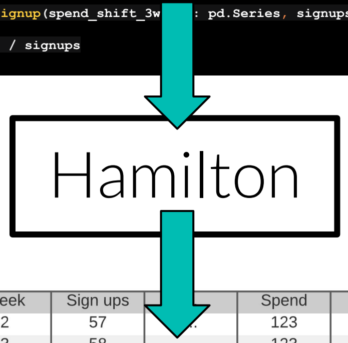


Hamilton: Key Point to remember (2)

Hamilton users:

> **<u>do not</u>** have to maintain how to connect computation with the outputs required.

```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_3w   : pd.Series, signups: pd.Series) -> pd.Series:
    """Some docs"""
    return spend_shift_3weeks / signups
```



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

Hamilton: in one sentence

A user friendly dataflow paradigm.

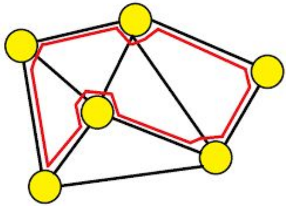
Hamilton: why is it called Hamilton?

Naming things is hard...

1. Associations with "FED":
   a. Alexander Hamilton is the father of the Fed.
   b. FED models business mechanics.

$$H_{operator} = \underbrace{\frac{-\hbar^2}{2m}\frac{\partial^2}{\partial x^2}}_{\text{Operator associated with kinetic energy}} + \underbrace{V(x)}_{\text{Potential energy}}$$

2. We're doing some basic graph theory.



apropos Hamilton

Example Hamilton Code

So you can get a feel for this paradigm...

Basic code - defining “Hamilton” functions

my_functions.py

```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
```

Basic code - defining “Hamilton” functions

my_functions.py

```
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
```

Output Column

Input Column

Driver code - how do you get the Dataframe?

```
from hamilton import driver
config_and_initial_data = {  # pass in config, initial data (or load data via funcs)
    'C': 3,   # a config variable
    'signups': ...,   # can pass in initial data – or pass in at execute time.
      ...
    'year': ...
}
module_name = 'my_functions'   # e.g. my_functions.py; can instead `import my_functions`
module = importlib.import_module(module_name)   # The python file to crawl

dr = driver.Driver(config_and_initial_data, module)   # can pass in multiple modules

output_columns = ['year','week',...,'spend_shift_3weeks_per_signup','special_feature']

df = dr.execute(output_columns) # only walk DAG for what is needed
```

Driver code - how do you get the Dataframe?

```
from hamilton import driver
config_and_initial_data = {  # pass in config, initial data (or load data via funcs)
    'C': 3,  # a config variable
    'signups': ...,  # can pass in initial data – or pass in at execute time.
     ...
    'year': ...
}
module_name = 'my_functions'  # e.g. my_functions.py; can instead `import my_functions`
module = importlib.import_module(module_name)  # The python file to crawl

dr = driver.Driver(config_and_initial_data, module)  # can pass in multiple modules

output_columns = ['year','week',...,'spend_shift_3weeks_per_signup','special_feature']

df = dr.execute(output_columns) # only walk DAG for what is needed
```

Driver code - how do you get the Dataframe?

```python
from hamilton import driver
config_and_initial_data = {  # pass in config, initial data (or load data via funcs)
    'C': 3,  # a config variable
    'signups': ...,  # can pass in initial data – or pass in at execute time.
      ...
    'year': ...
}
module_name = 'my_functions'  # e.g. my_functions.py; can instead `import my_functions`
module = importlib.import_module(module_name)  # The python file to crawl

dr = driver.Driver(config_and_initial_data, module)  # can pass in multiple modules

output_columns = ['year','week',...,'spend_shift_3weeks_per_signup','special_feature']

df = dr.execute(output_columns) # only walk DAG for what is needed
```

Driver code - how do you get the Dataframe?

```
from hamilton import driver
config_and_initial_data = {  # pass in config, initial data (or load data via funcs)
    'C': 3,  # a config variable
    'signups': ...,  # can pass in initial data – or pass in at execute time.
     ...
    'year': ...
}
module_name = 'my_functions'  # e.g. my_functions.py; can instead `import my_functions`
module = importlib.import_module(module_name)  # The python file to crawl

dr = driver.Driver(config_and_initial_data, module)  # can pass in multiple modules

output_columns = ['year','week',...,'spend_shift_3weeks_per_signup','special_feature']

df = dr.execute(output_columns) # only walk DAG for what is needed
```

Driver code - how do you get the Dataframe?

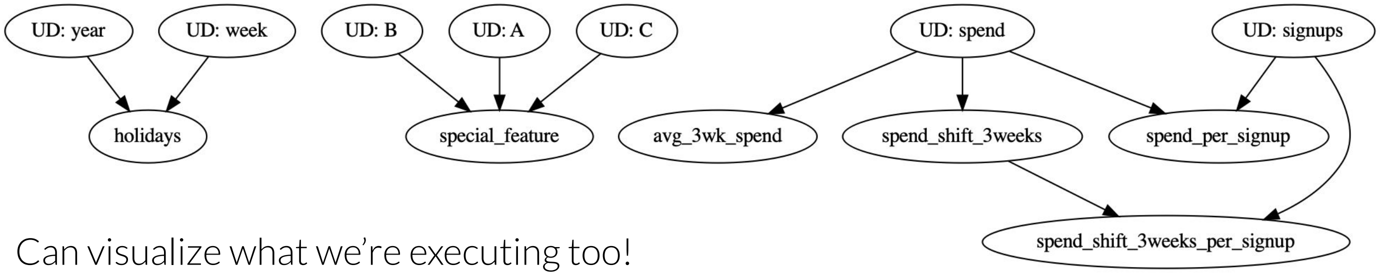
```
from hamilton import driver
config_and_initial_data = {  # pass in config, initial data (or load data via funcs)
    'C': 3,  # a config variable
    'signups': ...,  # can pass in initial data – or pass in at execute time.
      ...
    'year': ...
}
module_name = 'my_functions'  # e.g. my_functions.py; can instead `import my_functions`
module = importlib.import_module(module_name)  # The python file to crawl

dr = driver.Driver(config_and_initial_data, module)  # can pass in multiple modules

output_columns = ['year','week',...,'spend_shift_3weeks_per_signup','special_feature']

df = dr.execute(output_columns) # only walk DAG for what is needed
```

Driver code - how do you get the Dataframe?

```
from hamilton import driver
config_and_initial_data = {  # pass in config, initial data (or load data via funcs)
    'C': 3,  # a config variable
    'signups': ...,  # can pass in initial data – or pass in at execute() time.
      ...
    'year': ...
}
module_name = 'my_functions'  # e.g. my_functions.py; can instead `import my_functions`
module = importlib.import_module(module_name)  # The python file to crawl

dr = driver.Driver(config_and_initial_data, module)  # can pass in multiple modules

output_columns = ['year','week',...,'spend_shift_3weeks_per_signup','special_feature']

df = dr.execute(output_columns) # only walk DAG for what is needed
```

Driver code - how do you get the Dataframe?

```
from hamilton import driver
config_and_initial_data = {  # pass in config, initial data (or load data via funcs)
    'C': 3,  # a config variable
    's
     .
    'ye
}
module                                                                    functions`
module

dr = d                                                                    odules

output_columns = ['year','week',...,'spend_shift_3weeks_per_signup','special_feature']

df = dr.execute(output_columns) # only walk DAG for what is needed

dr.execute_visualization(output_columns, './dag.dot', {...render config…})
```



Can visualize what we're executing too!

Open Source: try it for yourself!

> **pip install sf-hamilton**

Get started in < 15 minutes!

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

Example
https://github.com/stitchfix/hamilton/tree/main/examples/hello_world

Hamilton: Summary

1. A user friendly [dataflow] paradigm.
2. Users write functions that create a DAG *through* function & parameter names.
3. Hamilton handles execution of the DAG.

Talk Outline:

Backstory: who, what, & why
Hamilton
> Hamilton @ Stitch Fix
Pro tips
Extensions

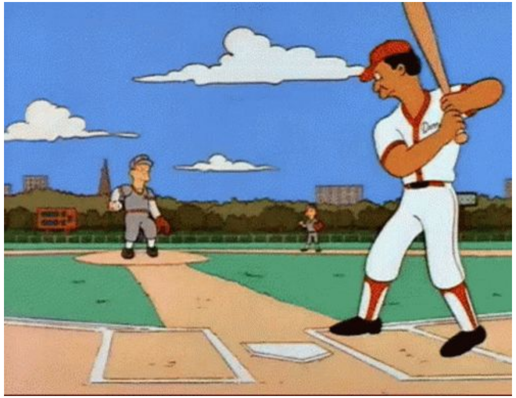
Hamilton @ SF - after 2+ years in production

Stitch Fix FED + Hamilton:

```python
def col_c(col_a: pd.Series, col_b: pd.Series) -> pd.Series:
    """documentation goes here"""
    return col_a + col_b
```

Original project goals:

- Improve ability to test ✅
- Improve documentation ✅
- Improve development workflow ✅



Why was it a home run?

Testing & Documentation

```
def col_c(col_a: pd.Series, col_b: pd.Series) -> pd.Series:
    “““documentation goes here”””
    return col_a + col_b
```

Output “column” → One function:

1. Single place to find logic.
2. Single function that needs to be tested.
3. Function signature makes providing inputs very easy!
   a. Function names & input parameters mean something!
4. Functions naturally come with a place for documentation!

⇒ Everything is **naturally** unit testable!
⇒ Everything is **naturally** documentation friendly!

Workflow improvements

```python
def col_c(col_a: pd.Series, col_b: pd.Series) -> pd.Series:
    """documentation goes here"""
    return col_a + col_b
```

What Hamilton also easily enabled:

- Ability to visualize computation
- Faster debug cycles
- Better Onboarding / Collaboration
    - *Bonus:*
        - Central Feature Definition Store

Visualization

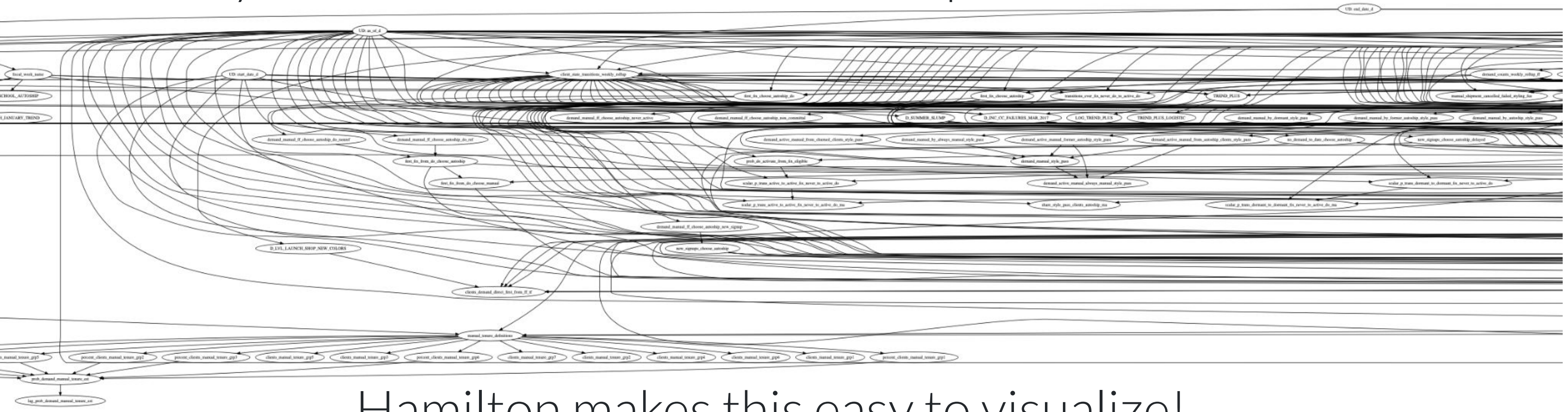
```
def col_c(col_a: pd.Series, col_b: pd.Series) -> pd.Series:
    “““documentation goes here”””
    return col_a + col_b
```

What if you have 4000+ columns to compute?

Visualization

```python
def col_c(col_a: pd.Series, col_b: pd.Series) -> pd.Series:
    “““documentation goes here”””
    return col_a + col_b
```

What if you have 4000+ columns to compute?



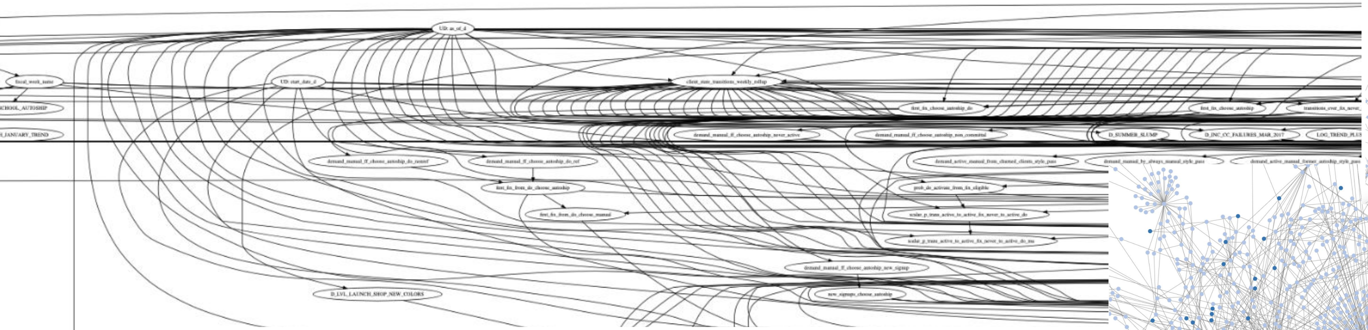
Hamilton makes this easy to visualize!

(zoomed out here to obscure names)

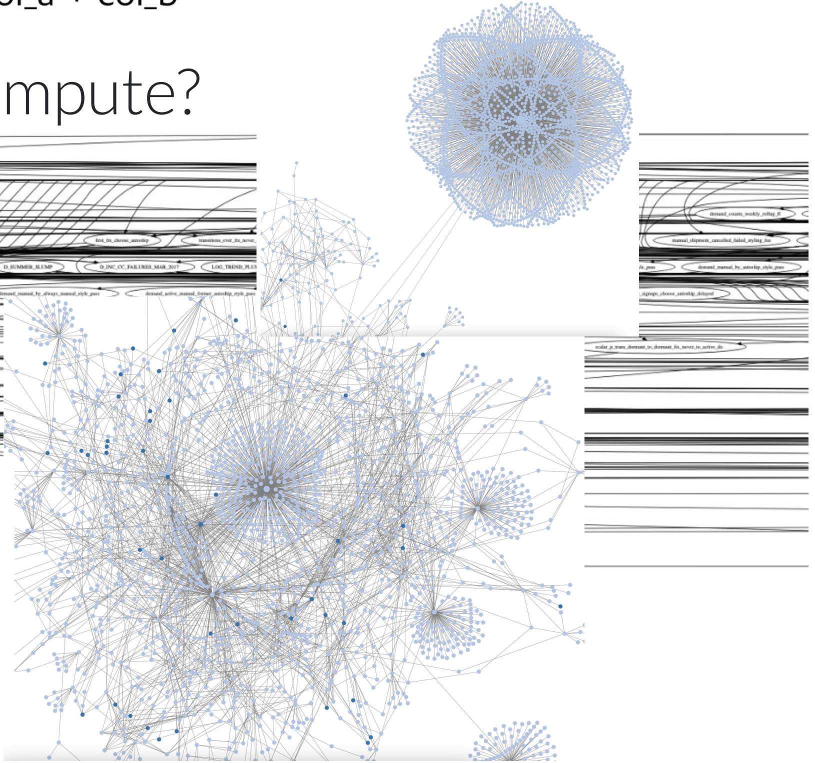
Visualization

```
def col_c(col_a: pd.Series, col_b: pd.Series) -> pd.Series:
    """documentation goes here"""
    return col_a + col_b
```

What if you have 4000+ columns to compute?



can create `DOT` files for export to other visualization packages →



Debugging these functions is easy!

my_functions.py

```
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
    """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
```

Can also import functions into other contexts to help debug.
e.g. in your REPL:

```
from my_functions import spend_shift_3weeks
output = spend_shift_3weeks(...)
```

Collaborating on these functions is easy!

my_functions.py

```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.Serie
    """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
```

Easy to assess impact & changes when:

- names mean something
- adding a new input
- changing the name of a function
- adding a brand new function
- deleting a function

⇒ Code reviews are much faster!
⇒ Easy to pick up where others left off!

Stitch Fix FED's Central Feature Definition Store

A nice byproduct of using Hamilton!

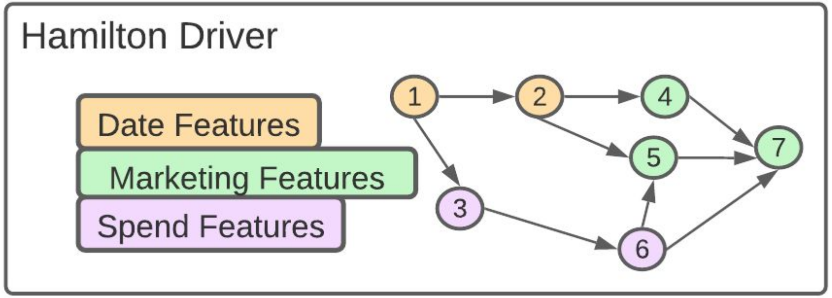
How they use it:

1. Function names follow team convention.
   a. e.g. ***D_*** *prefix* indicates date feature

Stitch Fix FED's Central Feature Definition Store

A nice byproduct of using Hamilton!

How they use it:
1. Function names follow team convention.
2. It's organized into thematic modules, e.g. date_features.py.
   a. Allows for working on different part of the DAG easily



Stitch Fix FED's Central Feature Definition Store

A nice byproduct of using Hamilton!

How they use it:

1. Function names follow team convention.
2. It's organized into thematic modules, e.g. date_features.py.
3. It's in a central repository & versioned by git:
    a. Can easily find/use/reuse features!
    b. Can recreate features from different points in time easily.

FED Testimonials

Just incase you don’t believe me

Testimonial (1)

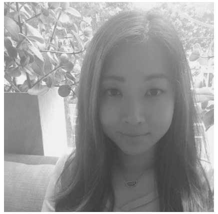
Danielle Q. 

"the encapsulation of the logic in a single named function makes adding nodes/edges simple to understand, communicate, and transfer knowledge"

E.g.:

- Pull Requests are easy to review.
- Onboarding is easy.

Testimonial (2)

Shelly J. 

"I like how easy-breezy it is to add new nodes/edges to the DAG to support evolving business needs."

E.g.

- new marketing push & we need to add a new feature:
    - **this takes minutes**, *not hours*!

Hamilton @ Stitch Fix

FED Impact Summary

FED Impact Summary

```
def col_c(col_a: pd.Series, col_b: pd.Series) -> pd.Series:
    """documentation goes here"""
    return col_a + col_b
```

With Hamilton, the FED Team gained:

- Naturally testable code. *Always.* ✅
- Naturally documentable code. ✅
- Dataflow visualization for free. ✅
- Faster debug cycles. ✅
- A better onboarding & collaboration experience ✅
    - Central Feature Definition Store as a by product! ✅

Total ⚾ Home Run!

FED Impact Summary

```python
def col_c(col_a: pd.Series, col_b: pd.Series) -> pd.Series:
    """documentation goes here"""
    return col_a + col_b
```

With H

- Nat
- Nat
- Dat
- Fas
- A b
  -

[claim]
By using Hamilton, the FED team can ***continue to scale*** their code base, without impacting team productivity
[/claim]

<u>Question</u>: is that true of your feature code base?

Total ⚾ Home Run!

Talk Outline:

Backstory: who, what, & why

Hamilton

Hamilton @ Stitch Fix

> Pro Tips

Extensions

Pro Tips - Five things to help you use Hamilton

1. Using it within your own ETL system
2. Migrating to Hamilton
3. Three key things to grok
4. Code organization & python modules
5. Function modifiers

1. Using Hamilton within your ETL system

ETL Framework compatibility:

- all ETL systems that run python 3.6+.

E.g. Airflow ✅
Metaflow ✅
Dagster ✅
Prefect ✅
Kubeflow ✅
etc. ✅

1. Using Hamilton within your ETL system

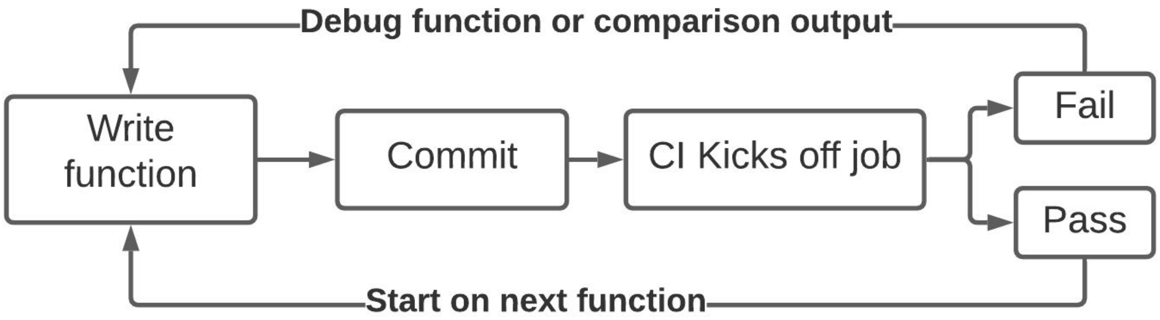
ETL Recipe:

1. Write Hamilton functions & “driver” code.
2. Publish your Hamilton functions in a package,
   or import via other means (e.g. checkout a repository).
3. Include `sf-hamilton` as a python dependency
4. Have your ETL system execute your “driver” code.
5. Profit.

2. Migrating to Hamilton: (1) CI for comparisons

Create a way to easily & frequently compare results.

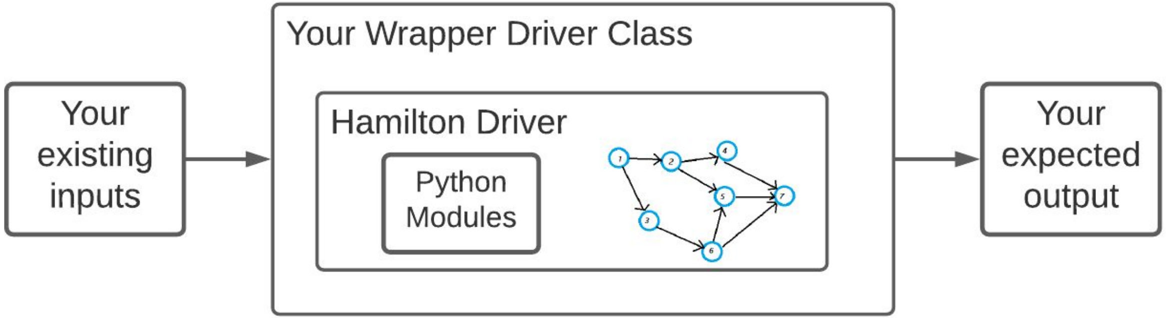
1. Integrate with continuous integration (CI) system if you can.
2. 🔎🐛 Helps diagnose bugs in your old & new code early & often.



2. Migrating to Hamilton: (2) Custom Wrapper

Wrap Hamilton in a *custom class* to match your existing API.

1. When migrating, avoid making too many changes.
2. Allows you to easily insert Hamilton into your context.



3. Key Concepts to Grok: (1) Common Index

If creating a DataFrame as output:

- Hamilton *relies* on the **series index** to join columns properly.

Best practice:

1. Load data.
2. Transform/ensure indexes match.
3. Continue with transformations.

At Stitch Fix – this meant a common DateTime index.

3. Key Concepts to Grok (2): Naming

Function Naming:

1. Creates your DAG.
2. Drives collaboration & code reuse.
3. Serves as documentation itself.

Key thought:

- Don’t need to get this right the first time
  - Can easily search & replace code as your thinking evolves.
- But it is something to converge thinking on!

3. Key Concepts to Grok: (3) Output immutability

Functions are only called once:

1. To preserve "*immutability*" of outputs,
   <u>*don't mutate*</u> *passed in data structures*.
   e.g. if you get passed in a pandas series, don't *mutate* it.
2. Otherwise YMMV with debugging.

Best practice:

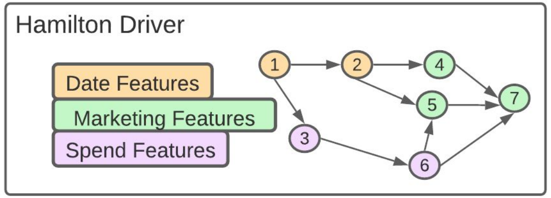
1. Test for this in your unit tests!
2. Clearly document mutating inputs if you do.

4. Code Organization & Python Modules

1. Functions are grouped into modules.
2. Modules are used as input to create a DAG.

Use this to your *development* advantage!

1. Use modules to model team thinking, e.g. date_features.py.
2. Helps isolate what you're working on.
3. Enables you to replace parts of your DAG easily for different contexts.




```
dr = driver.Driver(config_and_initial_data, dates, marketing, spend)
```

5. Function Modifiers; a.k.a. decorators

The **@(...)** above a function:

- Hamilton has a bunch to modify function behavior [docs]

```python
from hamilton.function_modifiers import extract_columns
@extract_columns(*my_list_of_column_names)
def load_spend_data(location: str) -> pd.DataFrame:
    """Some docs"""
    return pd.read_csv(location, ...)
```

Learn to use them:

- Functionality:
  - e.g. splitting a dataframe into columns
- Keeping code DRY
- FED favorite @config.when

Talk Outline:

Backstory: who, what, & why

Hamilton

Hamilton @ Stitch Fix

Pro Tips

> Extensions

Extensions - Why?

Initial Hamilton shortcomings:

1. Single threaded.
2. Could not scale to “big data”.
3. Could only produce Pandas DataFrames.
4. Does not leverage all the richness of metadata in the graph.

Extensions

1. Recent work
   - Scaling Computation
   - Removing the need for pandas
   - “Row based” execution
2. Planned extensions

Extensions: Recent Work

Covering a few things we recently released

Extensions - Scaling Computation

Hamilton grew up with a single core, in memory limitation

- Blocker to adoption for some.

Goal: to not modify Hamilton code to scale.

E.g. for creating Pandas DFs "it should just work" (on Spark, Dask, Ray, etc.)

Take this code – and scale it without changing it

my_functions.py

```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
```

Extensions - Scaling Computation

Hamilton grew up with a single core, in memory limitation

- Blocker to adoption for some.

Goal: to not modify Hamilton code to scale.

⭐ **Lucky for us:**

- Hamilton functions are generally very amenable for distributed computation.
- Pandas has a lot of support for scaling.

Extensions - Scaling Computation

What's in the **1.3.0 Release**:

- ***Experimental*** versions of Hamilton on:
  - [Dask] (cores + data)
  - [Koalas [Pandas API]] on Spark 3.2+ (cores + data)
  - [Ray] (cores + data*)

TL;DR:

- Can scale Pandas** out of the box!

* Cluster dependent
** Pandas use & Dask/Koalas dependent

Just how easy it is:
Example using Dask – only modify the “driver” script

```
from dask.distributed import Client
from hamilton import driver
from hamilton.experimental import h_dask
dag_config = {...}

bl module = importlib.import module('my functions')  # business logic functions
loader_module = importlib.import_module('data_loader')  # functions to load data

client = Client(...)
adapter = h_dask.DaskGraphAdapter(client)

dr = driver.Driver(dag_config, bl_module, loader_module, adapter=adapter)

output_columns = ['year','week',...,'spend_shift_3weeks_per_signup','special_feature']

df = dr.execute(output_columns) # only walk DAG for what is needed
```

Extensions - Custom Return Objects

What if I don't want a Pandas dataframe returned?

What's in the **1.3.0 Release**:

- Control over what the final object is returned!
  E.g.
  - Dictionary
  - Numpy matrix
  - Your custom object!

Just how easy it is:
Example Custom Object– only modify "driver" script

```
from dask.distributed import Client
from hamilton import driver
from hamilton import base
dag_config = {...}

bl_module = importlib.import_module('my_functions')  # business logic functions
loader_module = importlib.import_module('data_loader')  # functions to load data

adapter = base.SimplePythonGraphAdapter(base.DictResult())# or your custom class

dr = driver.Driver(dag_config, bl_module, loader_module, adapter=adapter)

output_columns = ['year','week',...,'spend_shift_3weeks_per_signup','special_feature']

# creates a dict of {col -> function result}
result_dict = dr.execute(output_columns)
```

Extensions - “Row Based” Execution

What if:

- I can’t fit everything into memory?
- Want to reuse my graph and call *execute* within a for loop with differing input?

What’s in the **1.3.0 Release**:

1. Enables you to configure the DAG once,
2. Then call `.*execute()*` with different inputs.

Enables data chunking & use cases like image processing or NLP.

Just how easy it is:
Example Row Execution– only modify “driver” script

```
from hamilton import driver
config_and_initial_data = {...}

module_name = 'my_functions'  # e.g. my_functions.py; can instead `import my_functions`
module = importlib.import_module(module_name)  # The python file to crawl

dr = driver.Driver(config_and_initial_data, module)  # instantiate driver once.

output_columns = ['year','week',...,'spend_shift_3weeks_per_signup','special_feature']

dataset = load_dataset()

for data_chunk in dataset:
    df = dr.execute(output_columns, inputs=data_chunk) # rerun execute on data chunks
    print(df)
```

Extensions - Recent Work Summary

Available as of 1.3.0 release:

- Distributed execution
  - ***Experimental*** versions of: Dask, Koalas on Spark, Ray
- **General purpose framework** with custom return objects:
  - Can return [numpy, pytorch, dicts, etc]
- Row based execution!
  - Chunk over large data sets
  - Process things one at at time, e.g. images, text.

Extensions: Planned Work

What we're thinking about next

Extensions - Planned Work

In no particular order:

- Numba integration (github issue)
- Data quality ala pandera (github issue)
- Lineage surfacing tools (github issue)

Extensions - Planned Work

Numba:

- Numba makes your code run much faster.
  Task: wrap Hamilton functions with *numba.jit* and compile the graph for speedy execution!

E.g. Scale your numpy & simple python code to:

- GPUs
- C/Fortran like speeds!

Extensions - Planned Work

Data Quality:

- Runtime inspection of data is a possibility.
  Task: incorporate expectations, ala [Pandera](), on functions.

e.g.

```
@check_output({'type': float, 'range': (0.0, 10000.0)})
def SOME_IMPORTANT_OUTPUT(input1: pd.Series, input2: pd.Series) -> pd.Series:
    """Does some complex logic"""
```

Extensions - Planned Work

Lineage surfacing tools:

- Want to ask questions of the metadata we have
  Task: provide classes/functions to expose this information.

E.g.
GDPR/PII questions:

- Where is this PII used and how?

Development questions:

- What happens if I change X, what impacts could it have?, etc.

Extensions - Planned Work

Please vote (❤️, 👍, etc) for what extensions we should prioritize!

https://github.com/stitchfix/hamilton/issues

To Conclude

Some TL:DRs

To Conclude

```python
def col_c(col_a: pd.Series, col_b: pd.Series) -> pd.Series:
    “““documentation goes here”””
    return col_a + col_b
```

1. Hamilton is a new paradigm to describe data flows.
2. It grew out of a need to tame a feature code base.
3. The Hamilton paradigm can provide teams with multiple productivity improvements & scales with code bases.
4. With the 1.3.0 release it’s now a scalable general purpose framework.

Thanks for listening – would love your feedback!

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

⭐ on github
☑️ create & vote on issues on github
📣 join us on [discord](https://discord.gg/wCqxqBqn73)
(https://discord.gg/wCqxqBqn73)

Thank you! Questions?

@stefkrawczyk
linkedin.com/in/skrawczyk

Try out Stitch Fix → goo.gl/Q3tCQ3

STITCH FIX