# Scalable feature engineering with Hamilton on Ray

Stefan Krawczyk, formerly of Stitch Fix

# Hamilton is Open Source

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

Get started in <15 minutes!

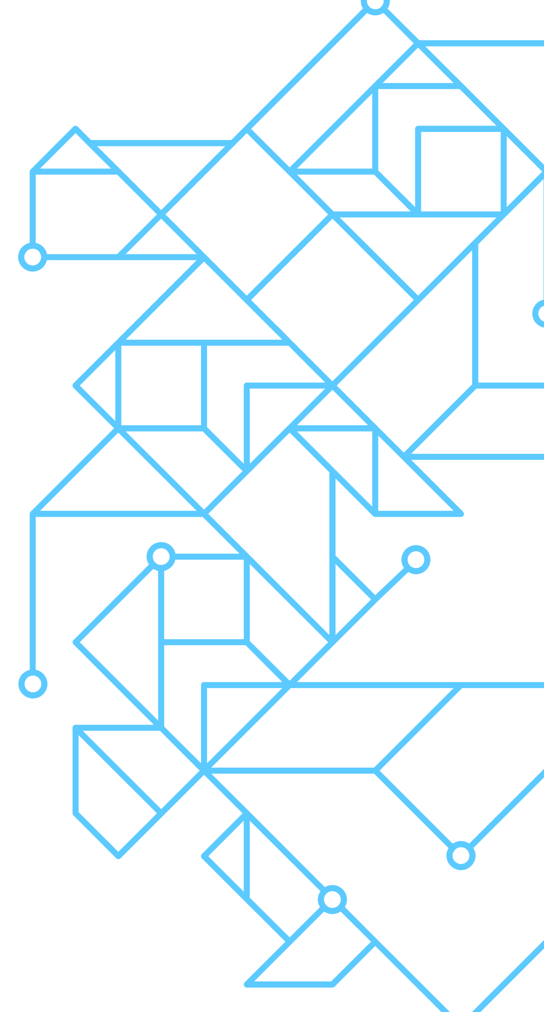
Documentation

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

Lots of examples:

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

# What is Hamilton?



# What is Hamilton?

A declarative dataflow paradigm.

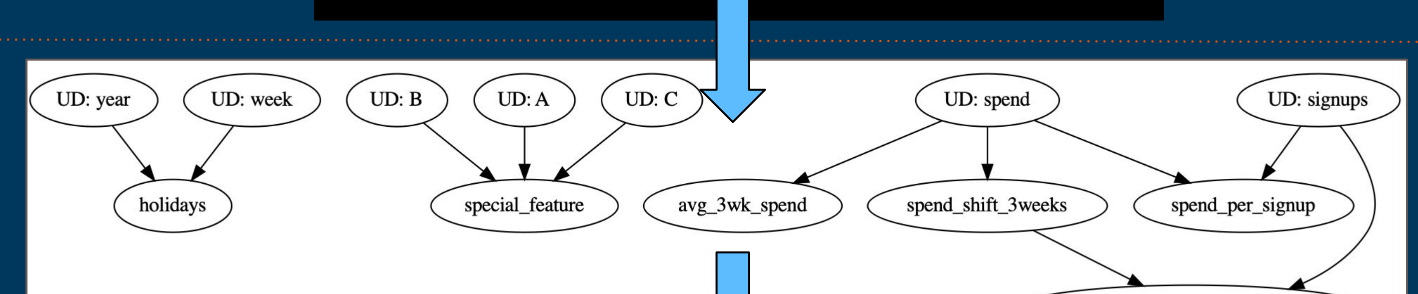
# Hamilton:
# Code → Directed Acyclic Graph → Object

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
```

User

DAG:




Hamilton

Object (e.g. 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

# Old way vs Hamilton way:

Instead of:

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

You declare:

```
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
```

+ some driver code (not shown)

# Old way vs Hamilton way:

Instead of:

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

**Outputs == Function Name**

**Inputs == Function Arguments**

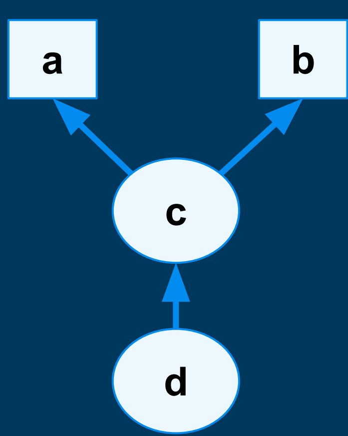
You declare:

```
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
```

# Explanation of Hamilton way

1. Functions names define nodes. [c, d]
2. Function arguments define edges. [c -> a, c -> b; d->c]
3. Via Driver, Hamilton framework creates a DAG & can execute it.

```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" - this actually says what and 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:

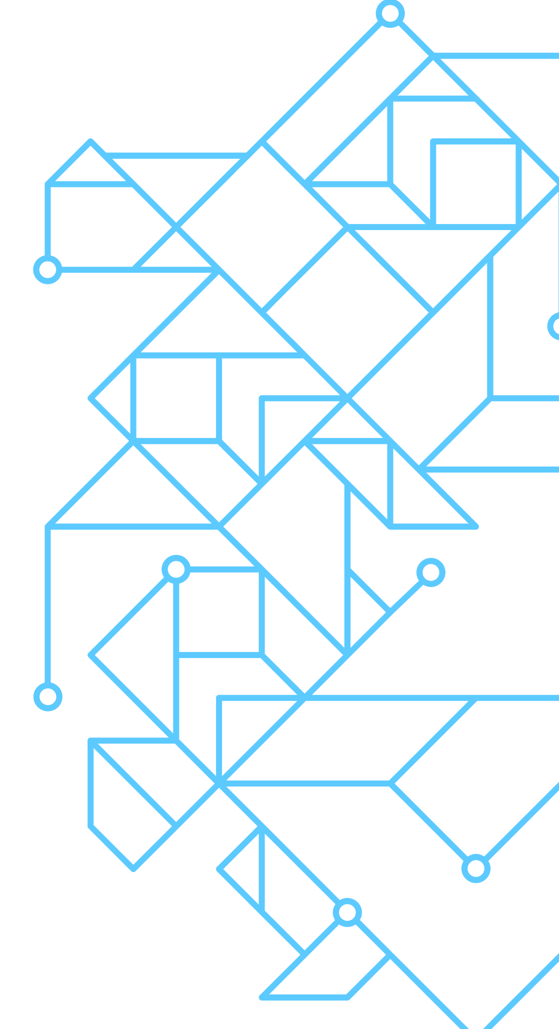
1. For each `=` statement, you write a function(s).
2. Functions declare a DAG.
3. Hamilton handles DAG execution.

```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)
```

a
b
c
d

# Why was Hamilton created?



# Backstory: Time-series Forecasting

Biggest problems here

Data, e.g. Business Actuals → Featurized Dataframe → Fit TS Models → Predict Future → Dataframe with Forecast

What Hamilton helped solve!

# Backstory: TS -> Dataframe creation

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: TS -> Dataframe creation

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 | ... |

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

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

etc 🔥

# Backstory: TS -> DF -> 🍝 Code

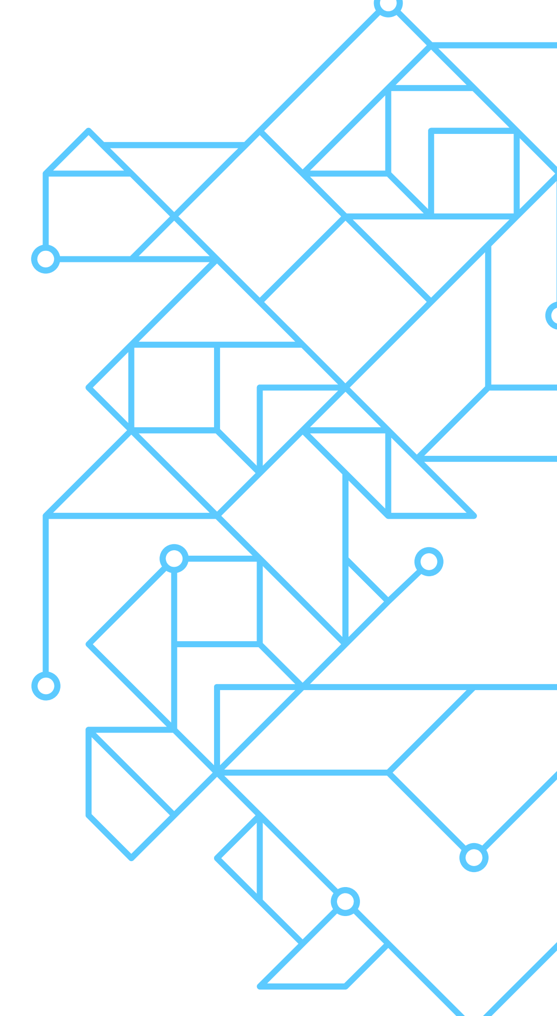
```
df = load_dates()  # load date ranges
df = load_actuals(df)  # load actuals, e.g. spend, signups
df['holidays'] = is_holiday(df['year'], df['week'])  # holidays
df['avg_3wk_spend'] = df['spend'].rolling(3).mean()  # mo...          spend
df['spend_per_signup'] = df['spend'] / df['si...           ...rson signed up
df['spend_shift_3weeks'] = df.spend[...           ...shift spend because ...
df['spend_shift_3weeks_pe...                 ...d_shift_3weeks'] / df['signups']

def my_spec...                 ...: pd.DataFrame) -> pd.Series:
    return (d...'A'] - df['B'] + df['C'])

df['special_feature'] = my_special_featu...
# ...
```

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

Human scaling 😔:

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

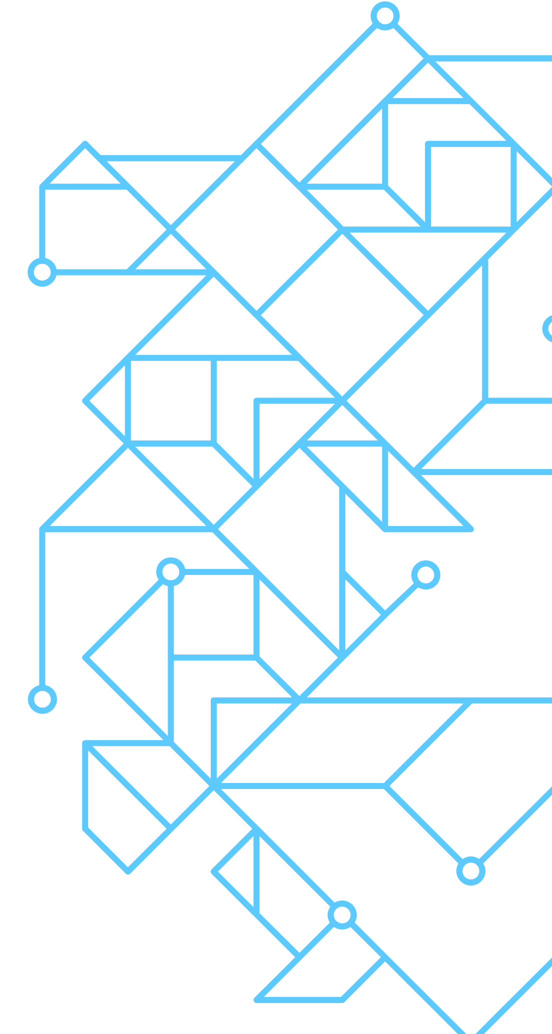
# Hamilton @ Stitch Fix



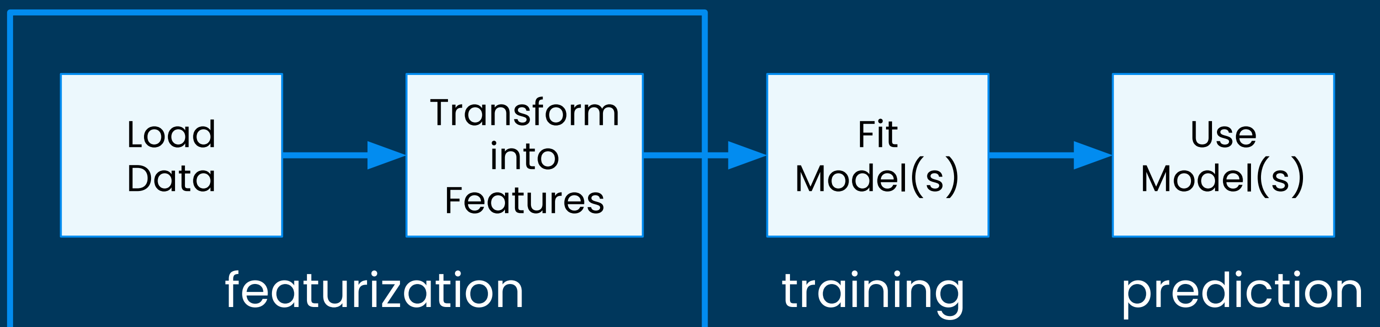
# Hamilton @ Stitch Fix

- Running in production for 2.5+ years
- Manages 4000+ feature definitions
- All feature definitions are:
  - Unit testable
  - Documentation friendly
  - Centrally curated, stored, and versioned in git.
- Data Science team ❤️s it:
  - Enabled a monthly task to be completed 4x faster
  - Easy to onboard new team members
  - Code reviews are simpler

# Overview: Feature Engineering with Hamilton



# Hamilton + Feature Engineering: Overview




- Can model this all in Hamilton (if you wanted to)
- We'll just focus on featurization
  - FYI: Hamilton works for any object type.
    - Here we'll assume pandas for simplicity.
  - Batch: use Hamilton within Airflow, Dagster, Prefect, Flyte, Metaflow, Kubeflow, etc.
  - Online: embed in python web services.

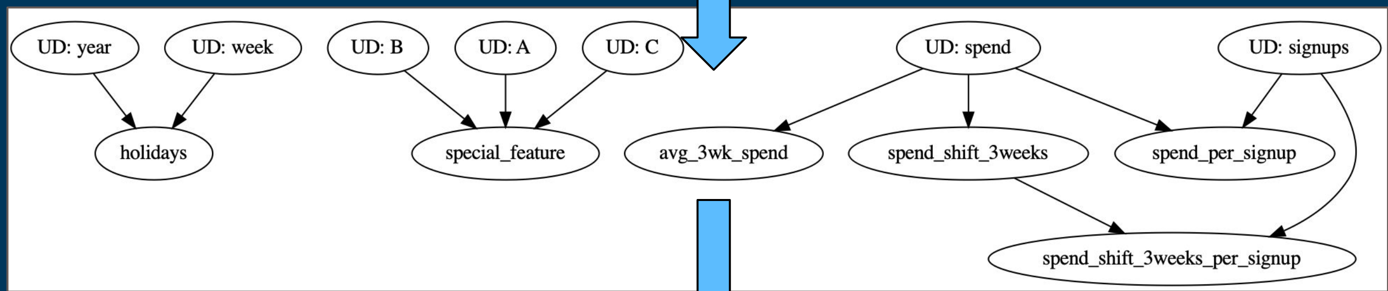
# Modeling featurization

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:

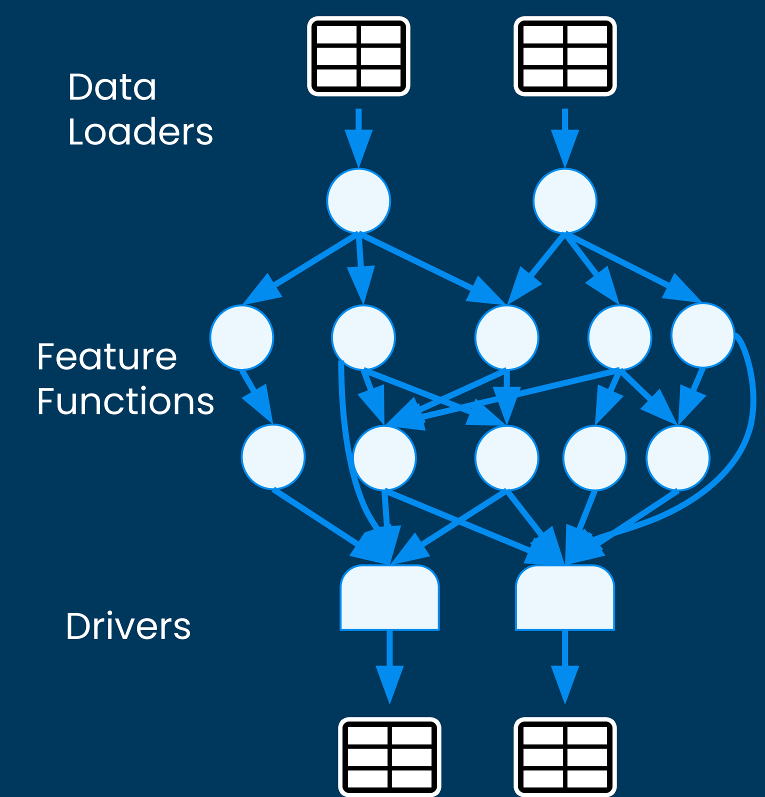
| Year | Week | Sign ups | ... | Spend | Holiday |
|---|---|---|---|---|---|
| 2015 | 2 | 57 | ... | 123 | 0 |
| 2015 | 3 | 58 | ... | 123 | 0 |
| 2015 | 4 | 59 | ... | 123 | 1 |
| 2015 | 5 | 59 | ... | 123 | 1 |
| ... | ... | ... | ... | ... | ... |
| ... | ... | ... | ... | ... | ... |
| ... | ...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

run.py

# Modeling featurization

Code that needs to be written:

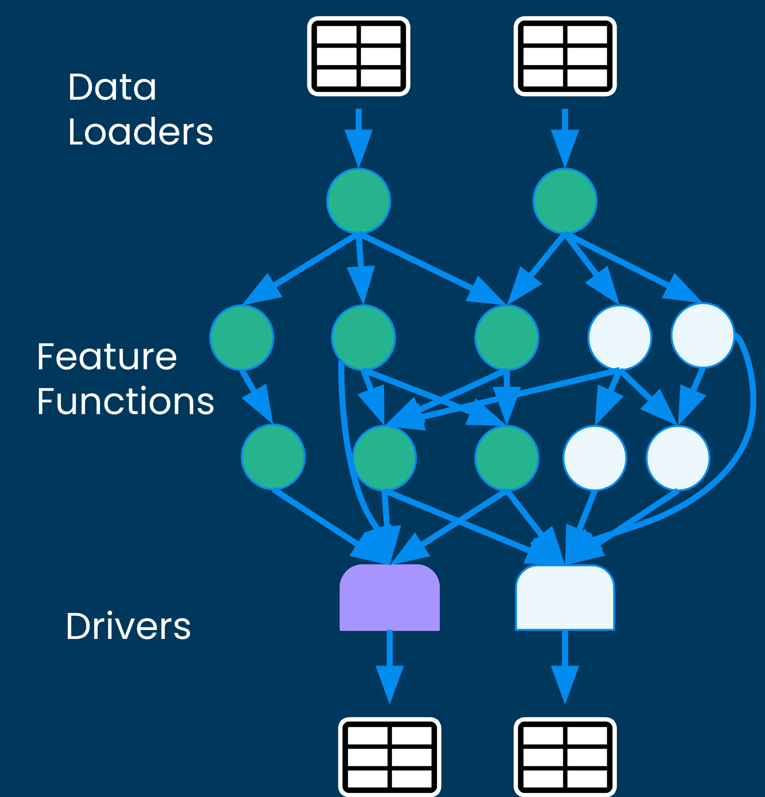
1. Functions to load data
    a. normalize/create common index to join on
2. Feature functions
    a. Optional: model functions.
3. Drivers materialize data
    a. DAG is walked for only what's needed.



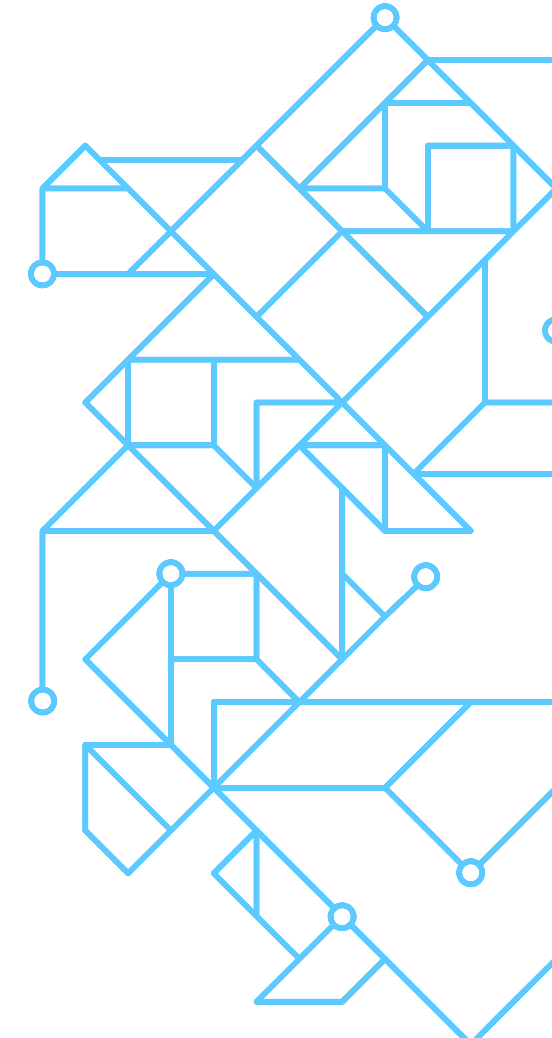
# Modeling featurization

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.



# Scalable Feature Engineering: Hamilton + Ray



# Problems that thwart scaling

> 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

} **Ray** helps here!

# How Hamilton helps with Human/Team Scaling

| Highly coupled code | Decouples “functions” from use (driver code). |
| --- | --- |

# How Hamilton helps with Human/Team Scaling

| | |
|---|---|
| Highly coupled code | Decouples “functions” from use (driver code). |
| In ability to reuse/understand work | Functions are curated into modules.<br><br>Everything is unit testable.<br><br>Documentation is natural.<br><br>Forced to align on naming. |

# How Hamilton helps with Human/Team Scaling

| | |
|---|---|
| Highly coupled code | Decouples “functions” from use (driver code). |
| In ability to reuse/understand work | Functions are curated into modules.<br>Everything is unit testable.<br>Documentation is natural.<br>Forced to align on naming. |
| Broken/unhealthy production pipelines | Debugging is straightforward.<br>Easy to version features via git/packaging.<br>Runtime data quality checks. |

RAY SUMMIT 22

# 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:

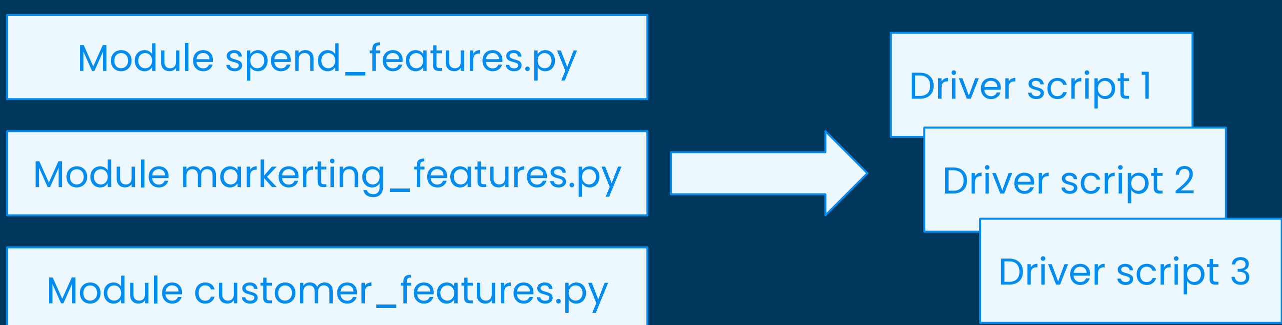
- Unit testing ✅ always possible
- Documentation ✅ tags, visualization, function doc
- Modularity/reuse ✅ module curation & drivers
- Central feature definition store ✅ naming, curation, versioning
- Data quality ✅ runtime checks

RAY SUMMIT 22

# Scaling Humans/Teams

Code base implications:

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



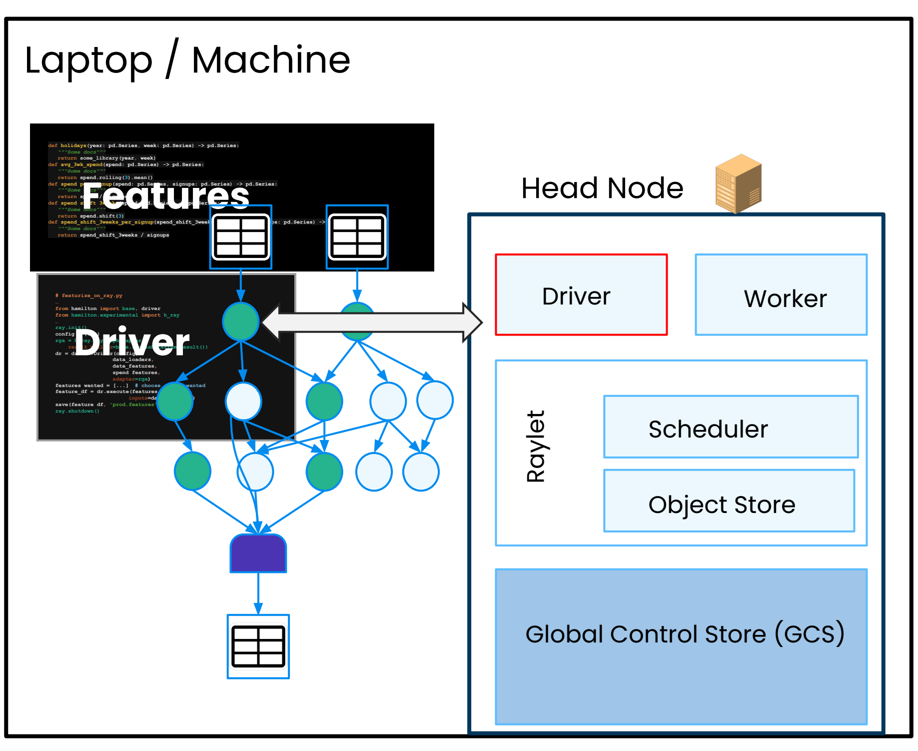

> Code reuse from day one!
> Low maintenance to support many driver scripts

# Scaling Compute/Data with Ray

- Ray enables you to scale beyond your laptop
  - Single process -> Multiprocessing -> Cluster
- Ray building blocks:
  - Built on Ray Core.
  - Ray workflows also supported.
- Switching to run on Ray requires:

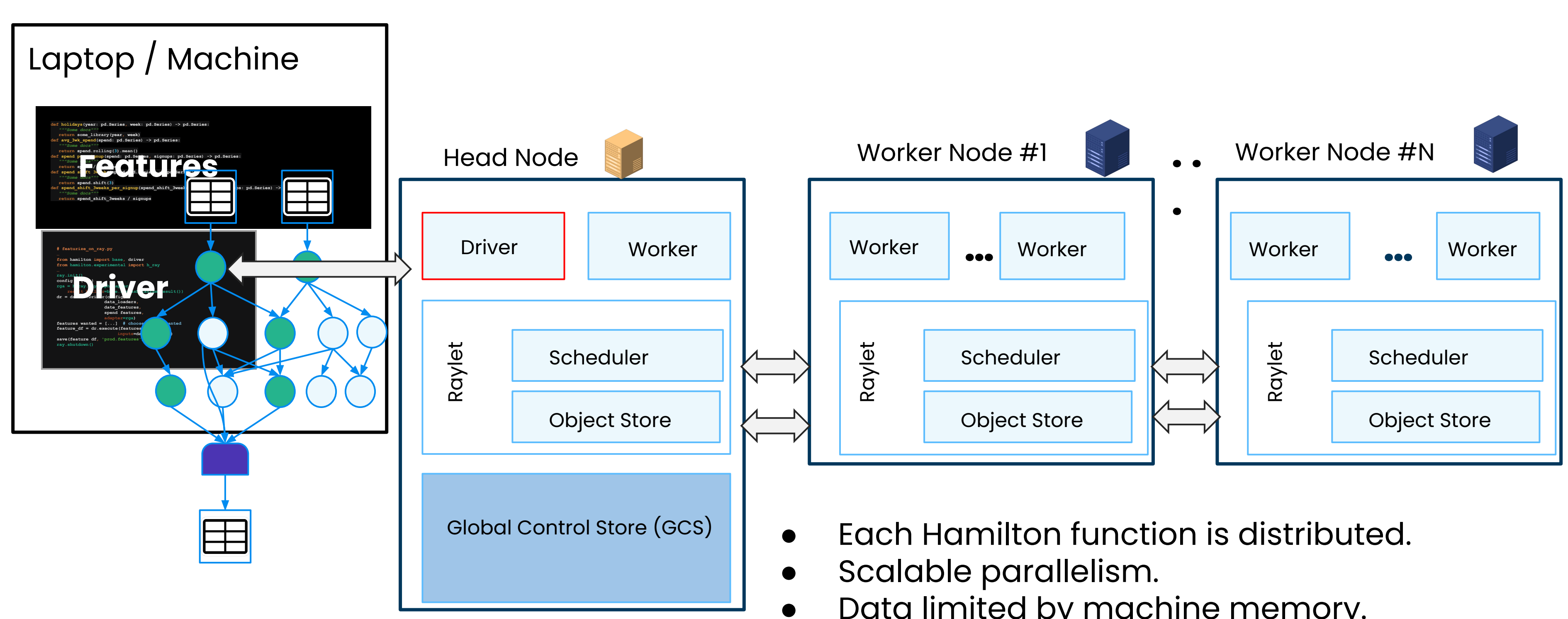
  › **<u>Only</u> changing driver.py code**

# Architecture Hamilton + Ray: Local




- Parallelism limited by cores on machine.
- Limited by memory on machine.

# Architecture Hamilton + Ray: Cluster




- Each Hamilton function is distributed.
- Scalable parallelism.
- Data limited by machine memory.

# 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()
```

RAY SUMMIT

# Hamilton + Ray: 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: 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

Ray Local/Cluster



```
# Delegate to Ray
…
ray.remote(
    node.callable).remote(**kwargs)
```

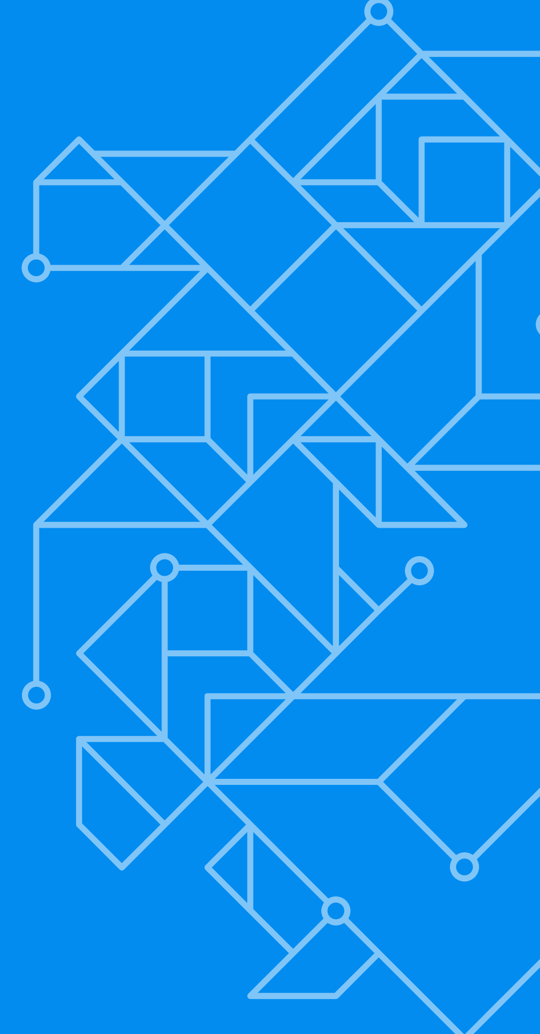
# Hamilton + Ray: Caveats

Currently you need to think about:

1. Serialization:
   a. everything needs to be pickle protocol v5 compatible.
2. Memory:
   a. memory aware scheduling isn't exposed.
      i. Need to figure out UX to expose this.
3. Python dependencies:
   a. Cluster has what you need installed
   b. Or, you specify them via `ray.init()`.
4. Looking to graduate Ray from experimental status

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

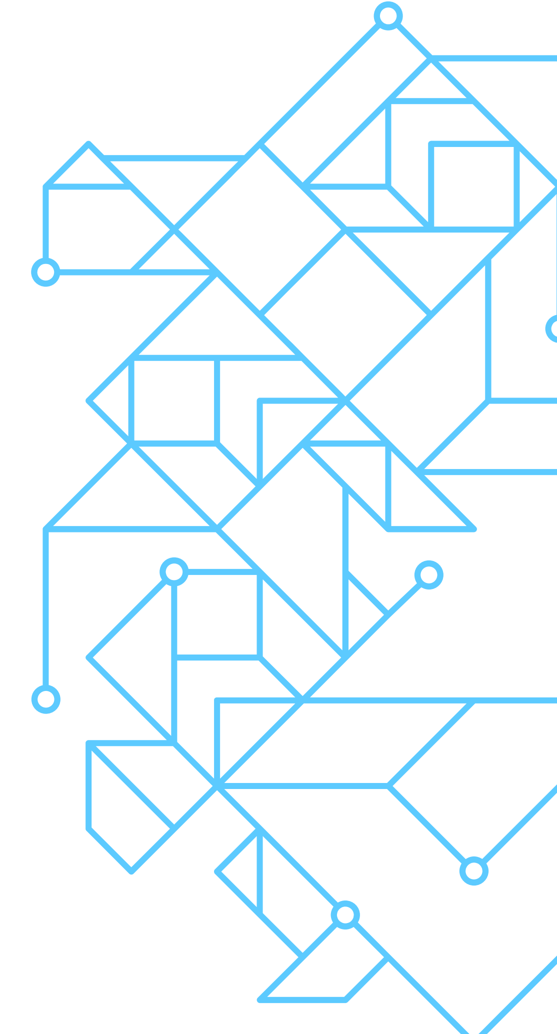
# Demo



# Demo Primer

1. Give a feel for what code might look like.
2. Show scaling to Ray.
3. Show visualization.
4. Show feature iteration cycle.

# Summary:
# Hamilton + Ray



# Summary: Hamilton + Ray

```
def feature_c(feature_a: pd.Series, feature_b: pd.Series) -> pd.Series:
    """Explanation of feature_c"""
    return a + b
```

- 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 code base
  - it'll make yours better too!
- The Hamilton paradigm scales humans/teams through software engineering best practices.
- **Hamilton + Ray** enables one to:

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

# Give Hamilton a Try!
# We'd love your Feedback

- `pip install sf-hamilton`

⭐ on [github](https://github.com/stitchfix/hamilton) (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)
(https://join.slack.com/t/hamilton-opensource/shared_invite/zt-1bjs72asx-wcUTgH7q7QX1igiQ5bbdcg)

# Thank you.

Questions?

https://twitter.com/stefkrawczyk

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

https://github.com/stitchfix/hamilton