DAGWORKS

+

dbt + Hamilton: Enabling you to maintain complex Python within dbt models

August 2023 MDSFest'23
Stefan Krawczyk - DAGWorks Inc.

whoami

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

12+ years in ML & Data platforms

STITCH FIX

IDIBON

IBM

VICTORIA UNIVERSITY OF WELLINGTON
TE HERENGA WAKA

Agenda

1. dbt + Python
2. Problems
3. Hamilton
4. Hamilton + dbt
5. Summary

dbt + Python

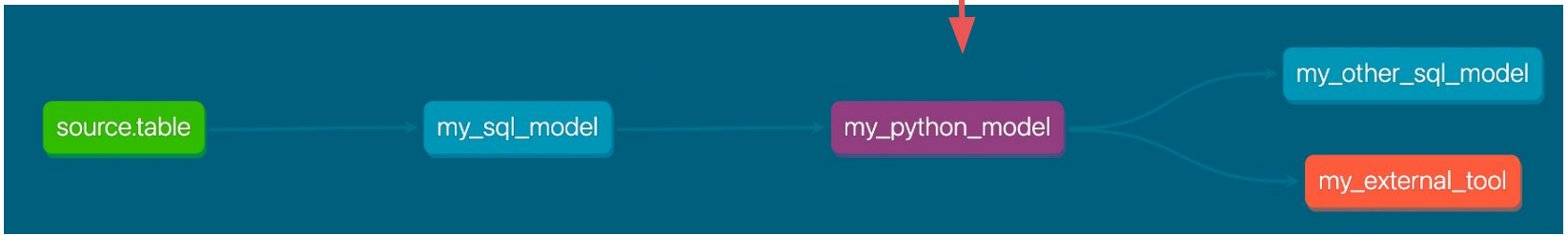
DAGWORKS

dbt + Python

Enables Python as a step, in your workflow:

1. Access upstream tables.
2. Transform.
3. Output is a dataframe.

```python
import ...

def model(dbt, session):

  my_sql_model_df = dbt.ref("my_sql_model")

  final_df = ... # stuff you can't write in SQL!

  return final_df
```

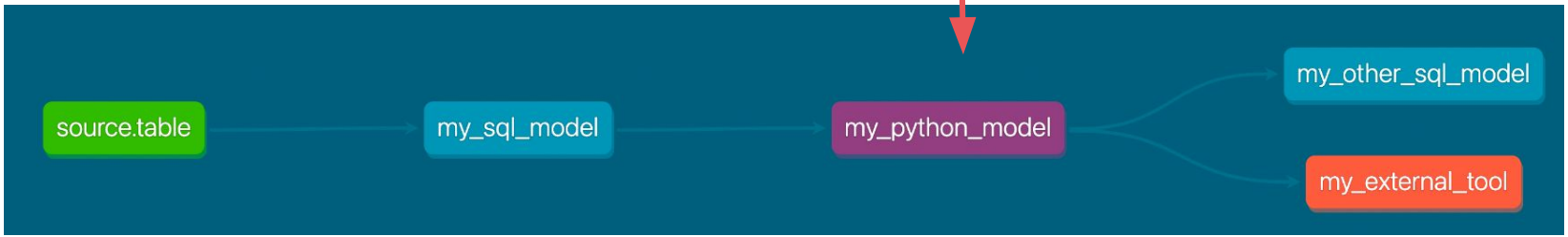


dbt + Python

Python enables:

- Machine Learning
- More complex transforms
- LLMs

```python
import ...

def model(dbt, session):

  my_sql_model_df = dbt.ref("my_sql_model")

  final_df = ... # stuff you can't write in SQL!

  return final_df
```



dbt + Python: where does Python run though?

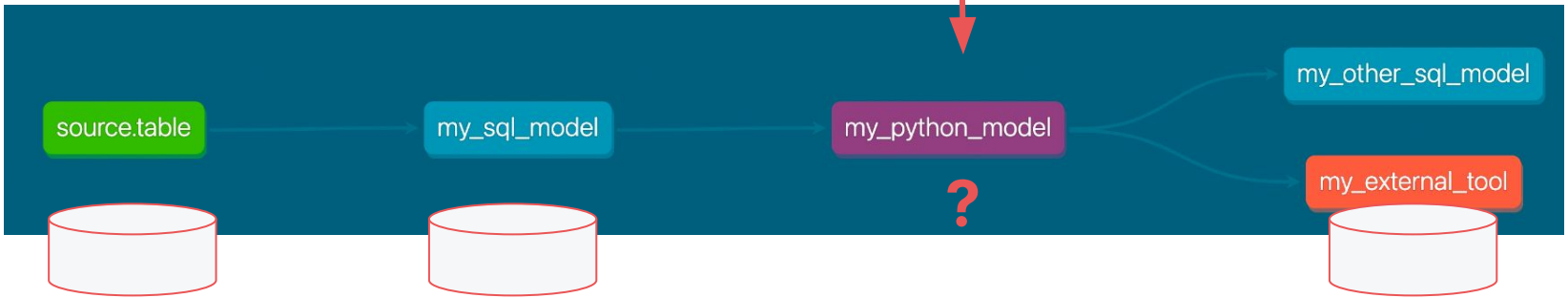
Default options:

- PySpark (GCP, Databricks)
- Snowpark (Snowflake)

With **dbt-fal** adapter:

- Where `dbt run` executes.

```python
import ...

def model(dbt, session):

  my_sql_model_df = dbt.ref("my_sql_model")

  final_df = ... # stuff you can't write in SQL!

  return final_df
```



Problems: dbt + Python

Problems: dbt + Python

Hard to manage anything substantial:

1. Can't share code between models.
   - Have to deal with packaging...
2. As code grows, you lose:
   - Lineage
   - Testability
   - Readability/documentation
3. DE + DS collaboration is harder than it needs to be!

```
def add_one(x):
  return x + 1

def model(dbt, session):
  dbt.config(materialized="table")
  temps_df = dbt.ref("temperatures")

  # warm things up just a little
  df = temps_df.withColumn(
       "degree_plus_one",
       add_one(temps_df["degree"]))
  return df
```

https://docs.getdbt.com/docs/building-a-dbt-project/building-models/python-models#defining-functions

Problems: dbt + Python

Hard to manage anything substantial:

1. Can't share code between models.
   - Have to deal with packaging...
2. As code grows, you lose:
   - Lineage
   - Testability
   - Readability/documentation
3. DE + DS collaboration is harder than it needs to be!

Focus of this talk

```python
def add_one(x):
  return x + 1

def model(dbt, session):
  dbt.config(materialized="table")
  temps_df = dbt.ref("temperatures")

  # warm things up just a little
  df = temps_df.withColumn(
        "degree_plus_one",
        add_one(temps_df["degree"]))
  return df
```

https://docs.getdbt.com/docs/building-a-dbt-project/building-models/python-models#defining-functions

Hamilton

DAGWORKS

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:

Micro-orchestration is handling what happens within this step

What's a dataflow?

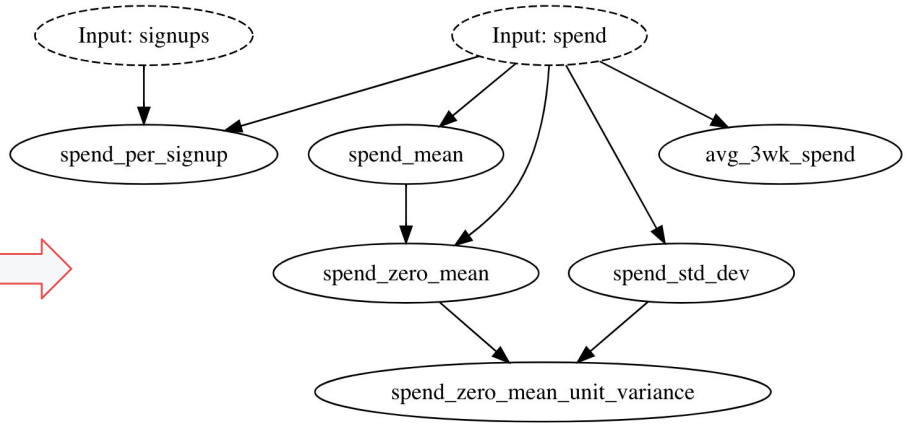
Fancy way of saying:

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['s
spend_mean = df['spend'].mean()
df['spend_zero_mean'] = df['spend']
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 Way:

Instead of

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

Outputs == Function Name **Inputs == Function Arguments**

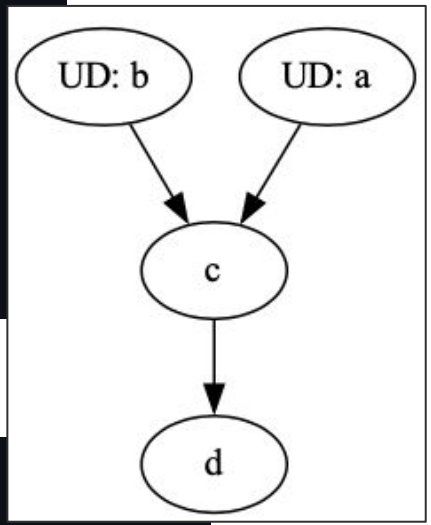
You declare

```
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
```

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)
```

Benefits: More reliable & maintainable code

```
# client_features.py

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

Testing: easier to unit & integration test.

Benefits: More reliable & maintainable code

```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: pd.Series) -> pd.Series:
   return height_zero_mean / height_std_dev
```

Testing: easier to unit & integration test.

Data quality: runtime checks via annotation*.

Benefits: More reliable & maintainable code

```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
```

Testing: easier to unit & integration test.

Data quality: runtime checks via annotation*.

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

Benefits: More reliable & maintainable code

```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
```

Testing: easier to unit & integration test.

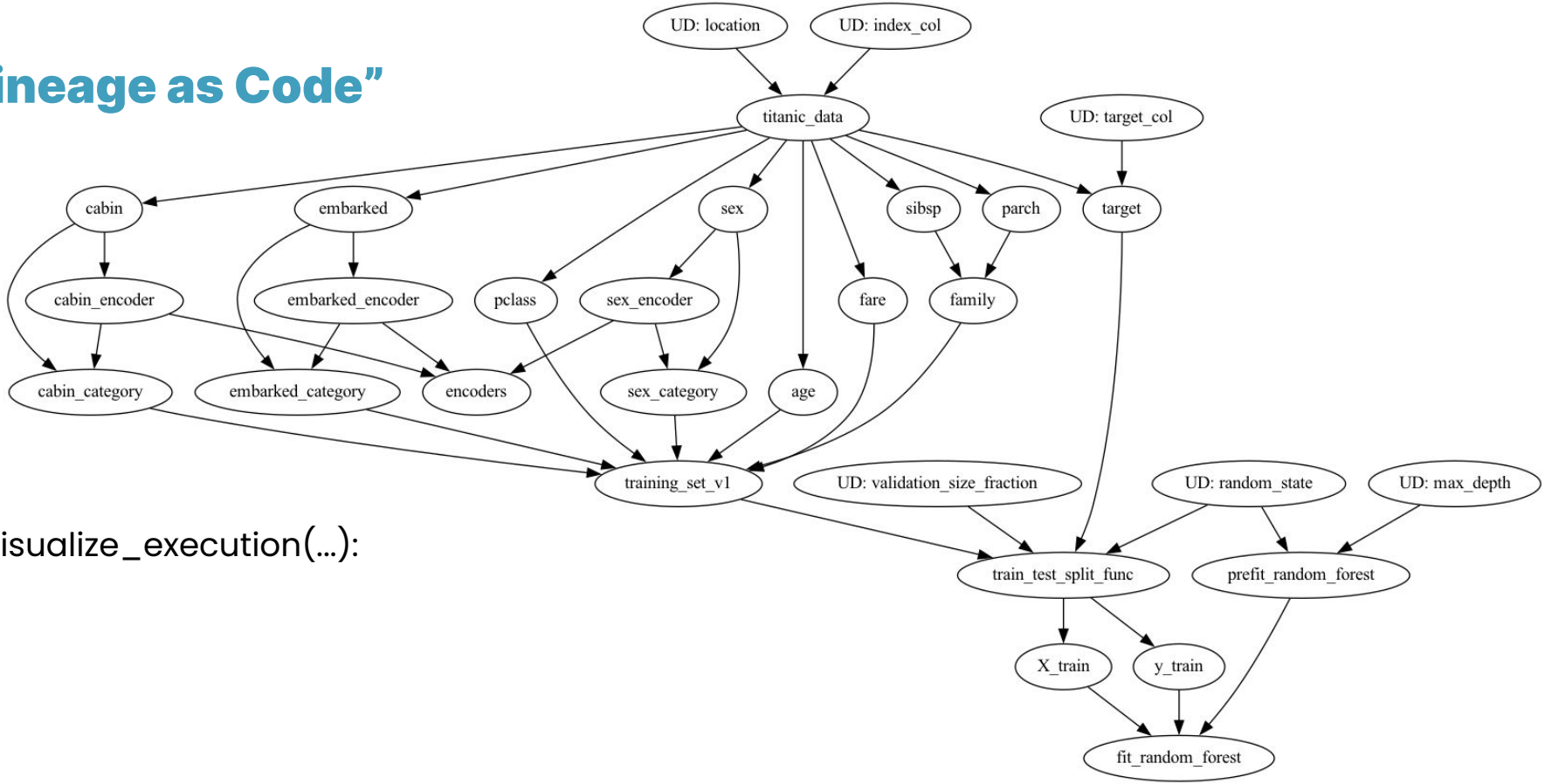
Data quality: runtime checks via annotation*.

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

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

Visualization is first class

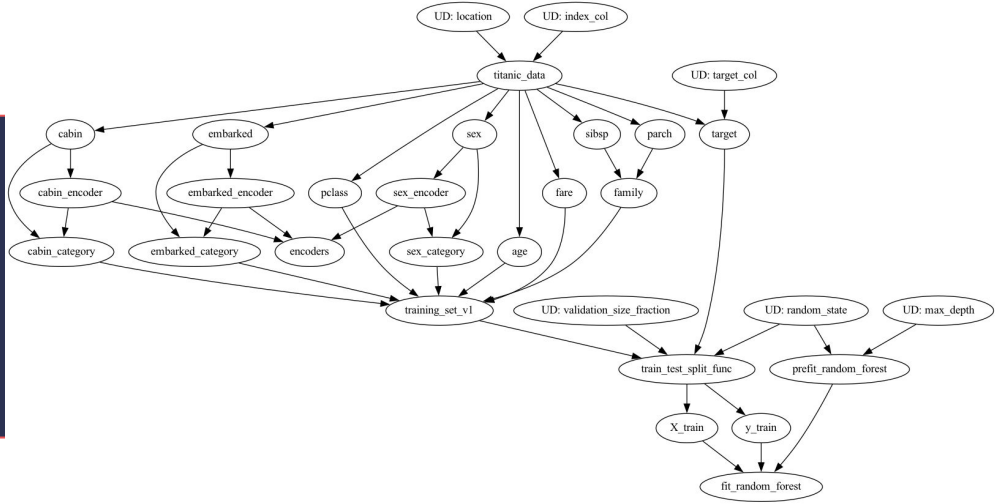
“Lineage as Code”



dr.visualize_execution(...):

Benefits: Faster iteration & collaboration

```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
```



Reusable: Functions are in modules. Reusable from day 1.

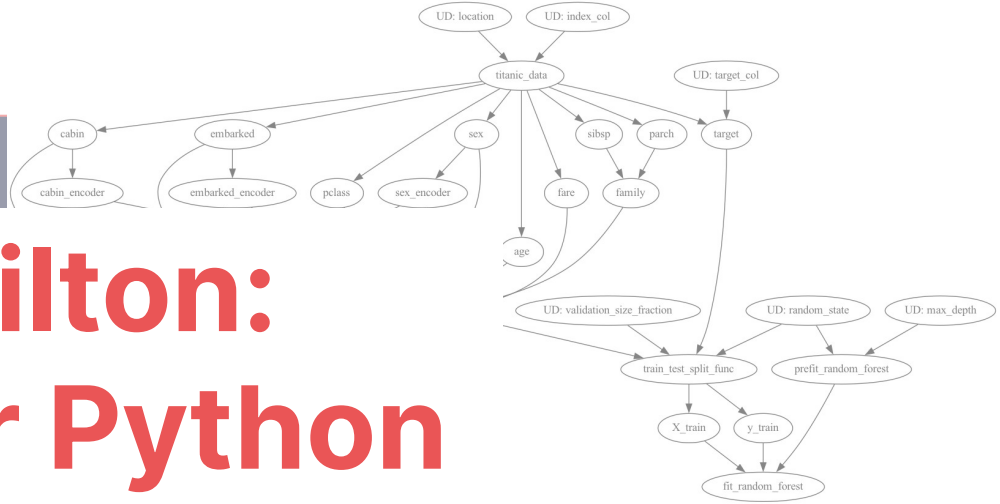
Modular: Can define different versions/implementations surgically.

Portable: Runs anywhere python runs. e.g., dbt models, etc:

RAY dask APACHE Spark FastAPI Flask jupyter

Benefits: Faster iteration & collaboration

```
# client_features.py
@tag(owner='Data-Science', pii='Fal
@check_output(data_type=np.float64,
def height_zero_mean_unit_variance(

    """Zero mean unit variance value
    return height_zero_mean / height
```



Hamilton: ~dbt for Python

Reusable: Functions are in modules. Reusable from day 1.

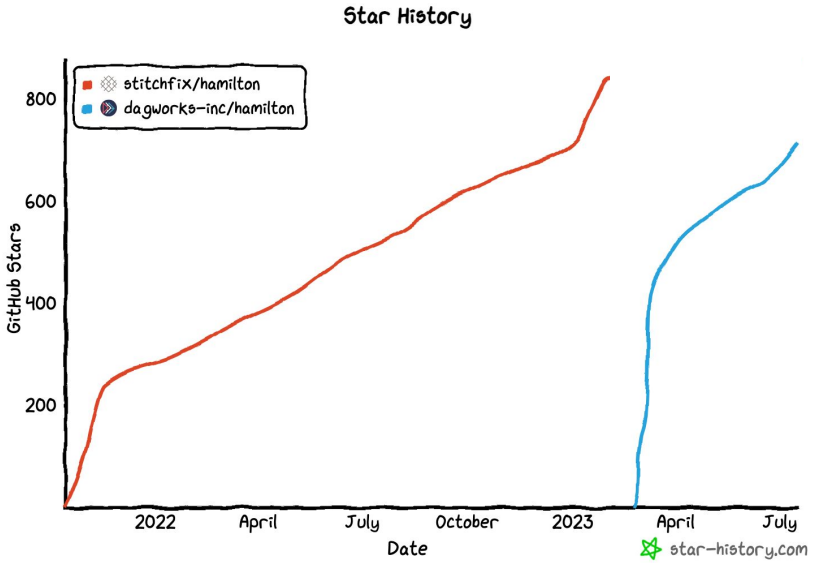
Modular: Can define different versions/implementations surgically.

Portable: Runs anywhere python runs. e.g., dbt models, etc:

RAY dask APACHE Spark FastAPI Flask jupyter

Some Hamilton stats

~1.5K Unique Stargazers
195+ slack members
93K+ downloads



Note: dbt took 3.5 years to get to 600 stars

Hamilton is used by many, including:

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

Hamilton + dbt

DAGWORKS

Hamilton + dbt: mechanics

1. Create python model
   - Use pandas
   - Use pyspark/pandas-on-spark
   - Snowpark DF*
2. Run Hamilton code to create dataframe.

```python
from hamilton import driver
import transforms, model_fitting

def model(dbt, session):

  my_sql_model_df = dbt.ref("my_sql_model")

  dr = driver.Driver({}, transforms, model_fitting)

  final_df = dr.execute( # inputs can be whatever
     ["my_final_df], inputs={...})

  return final_df
```

Hamilton + dbt: mechanics

1. Create python model
   - Use pandas
   - Use pyspark/pandas-on-spark
   - Snowpark DF*
2. Run Hamilton code to create dataframe.

Requires:

- installing sf-hamilton
- packaging your python modules

```python
from hamilton import driver
import transforms, model_fitting

def model(dbt, session):

  my_sql_model_df = dbt.ref("my_sql_model")

  dr = driver.Driver({}, transforms, model_fitting)

  final_df = dr.execute( # inputs can be whatever
     ["my_final_df], inputs={...})

  return final_df
```

This depends on your set up as to what to do; will briefly mention ways.

Hamilton + dbt: DE & DS collaboration

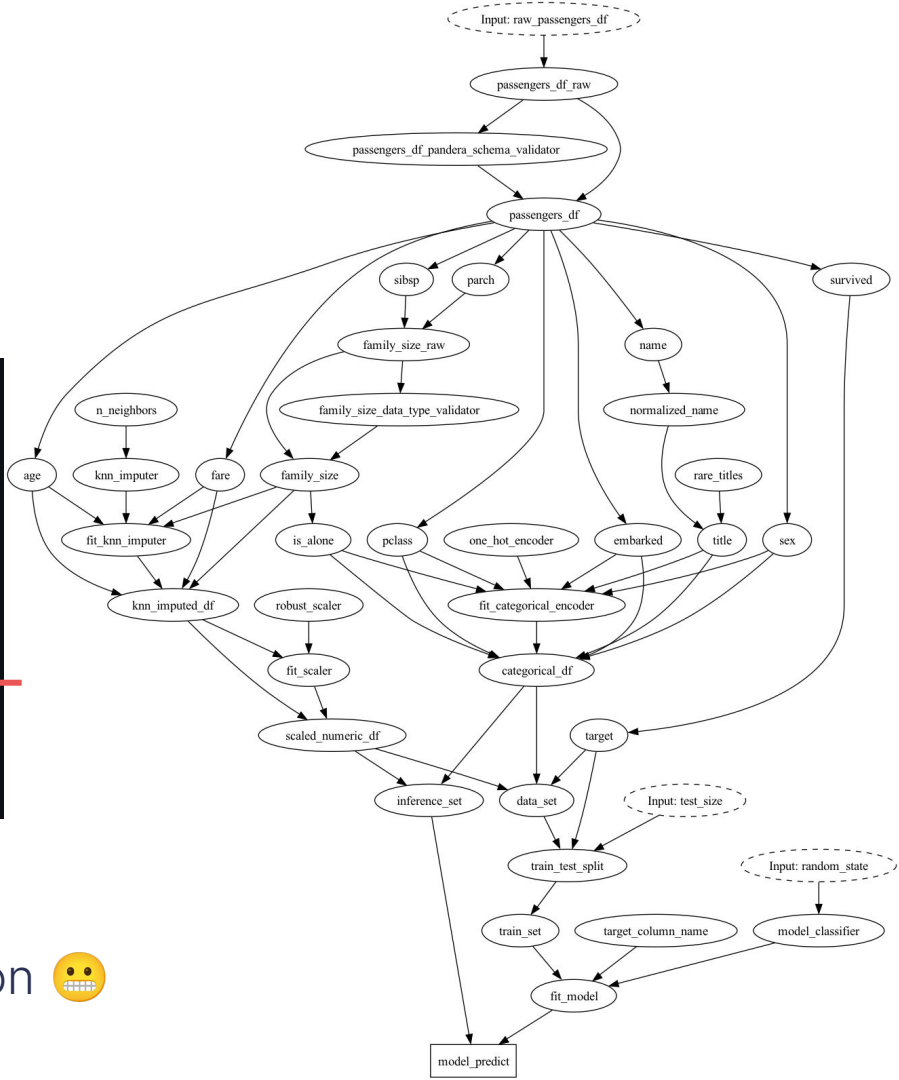
1. Hamilton forces DS code into python modules.
2. DBT model code is straightforward.
3. Contract becomes:
   - Expected schema in.
   - Dataframe out*.
4. DS develops outside of dbt; passes/implements model when ready.
5. DS can test independently, while DE can test independently.

```python
from hamilton import driver
import transforms, model_fitting

def model(dbt, session):

  my_sql_model_df = dbt.ref("my_sql_model")

  dr = driver.Driver({}, transforms, model_fitting)

  final_df = dr.execute( # inputs can be whatever
     ["my_final_df], inputs={...})

  return final_df
```

Example: doing ML on the Titanic Dataset

Normally this would be 🤯:

```
import ...

def model(dbt, session):

  my_sql_model_df = dbt.ref("my_sql_model")

  final_df = ...

  return final_df
```



> No lineage view like →

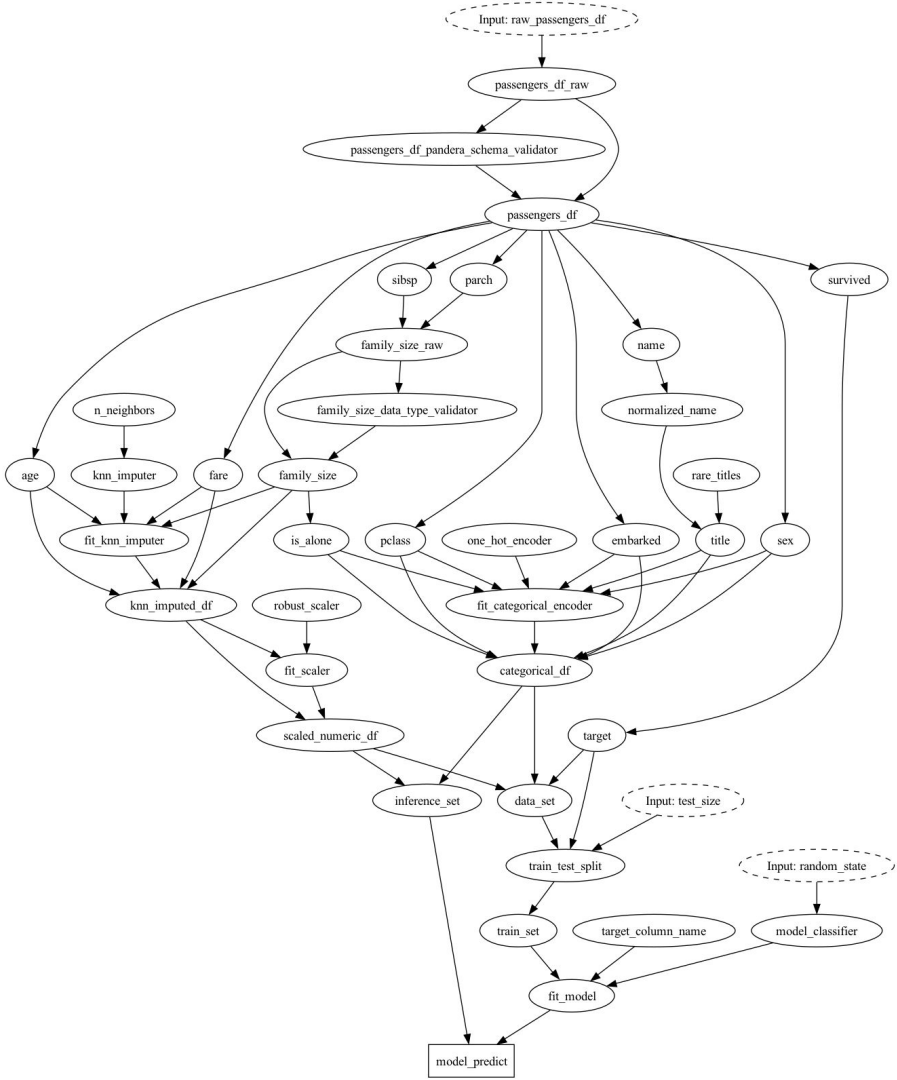
> SDLC: Testing, debugging & collaboration 😬

Example: doing ML on the Titanic Dataset

Find this in the Hamilton repo under dbt; note we use the dbt-fal adapter.

Premise:

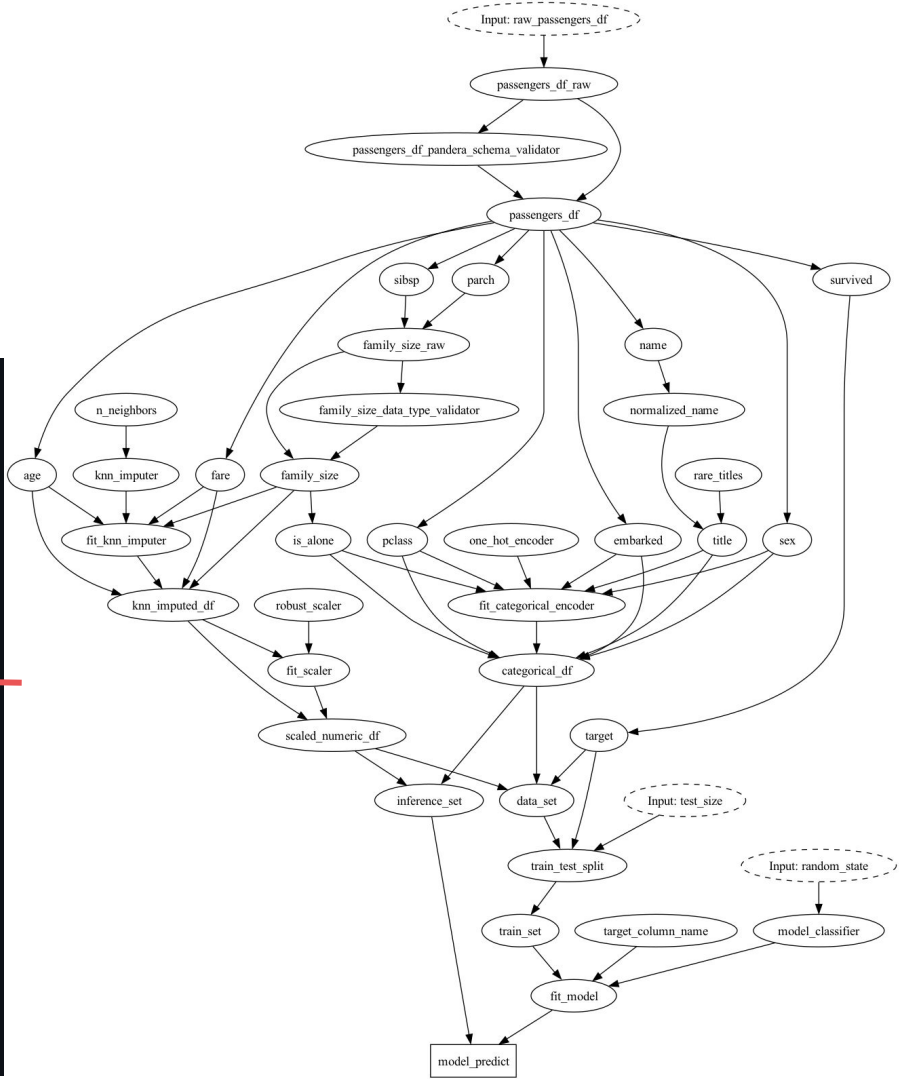
- Base table could be any source table in your warehouse.
- “Realistic” ML pipeline
- Output is a column of predictions



Example: doing ML on the Titanic Dataset

With Hamilton 😎

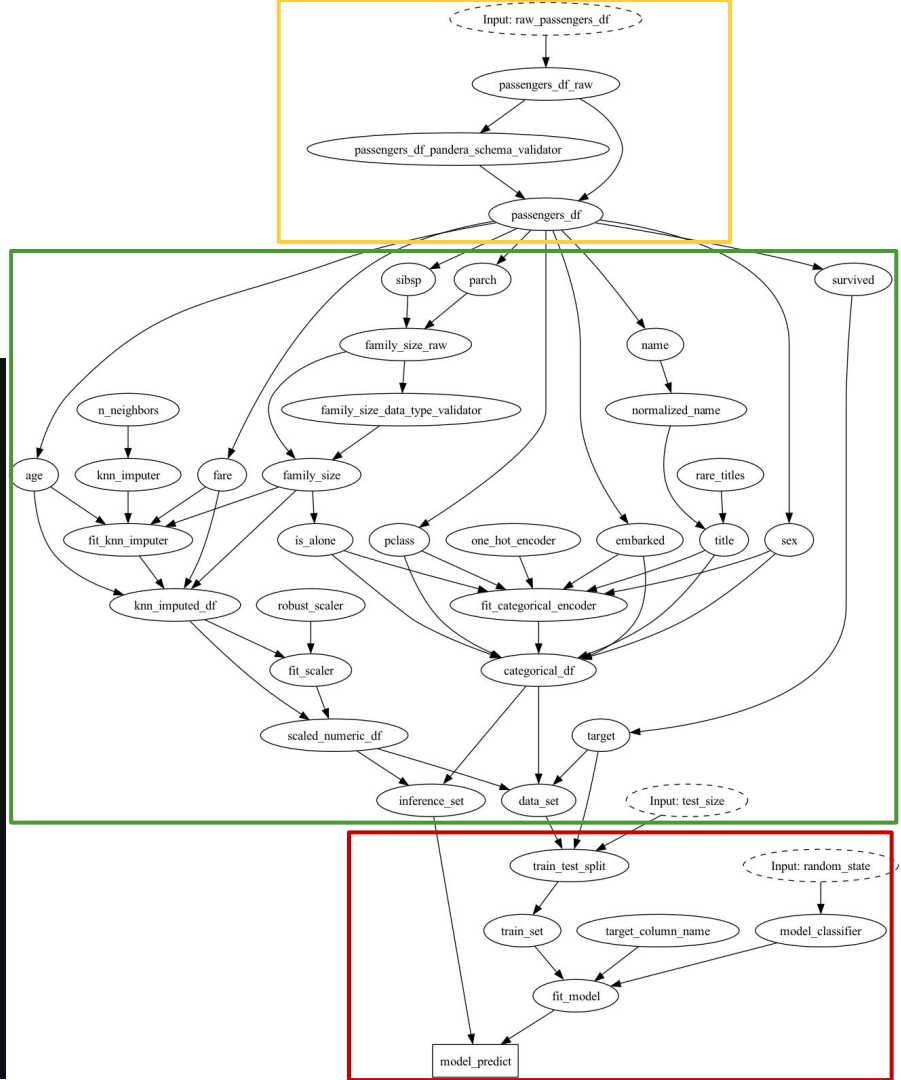
```python
import data_loader, feature_transforms, model_pipeline

def model(dbt, session):
  raw_passengers_df = dbt.ref("raw_passengers")
  # DAG for training/inferring on titanic data
  titanic_dag = driver.Driver(config,
    data_loader, feature_transforms, model_pipeline,
    adapter=base.DefaultAdapter(),
  )
  # execute & get output
  result = titanic_dag.execute(["model_predict"],
      inputs={"raw_passengers_df": raw_passengers_df}
  )
  # Take the "predictions" result,
  # which is an np array
  predictions = result["model_predict"]
  # Return a dataframe!
  return pd.DataFrame(predictions,
                      columns=["prediction"])
```



Example: doing ML on the Titanic Dataset

With Hamilton 😎: Modularity

```python
import data_loader, feature_transforms, model_pipeline

def model(dbt, session):
  raw_passengers_df = dbt.ref("raw_passengers")
  # DAG for training/inferring on titanic data
  titanic_dag = driver.Driver(config,
    data_loader, feature_transforms, model_pipeline,
    adapter=base.DefaultAdapter(),
  )
  # execute & get output
  result = titanic_dag.execute(["model_predict"],
      inputs={"raw_passengers_df": raw_passengers_df}
  )
  # Take the "predictions" result,
  # which is an np array
  predictions = result["model_predict"]
  # Return a dataframe!
  return pd.DataFrame(predictions,
                      columns=["prediction"])
```



Example: doing ML on the Titanic Dataset

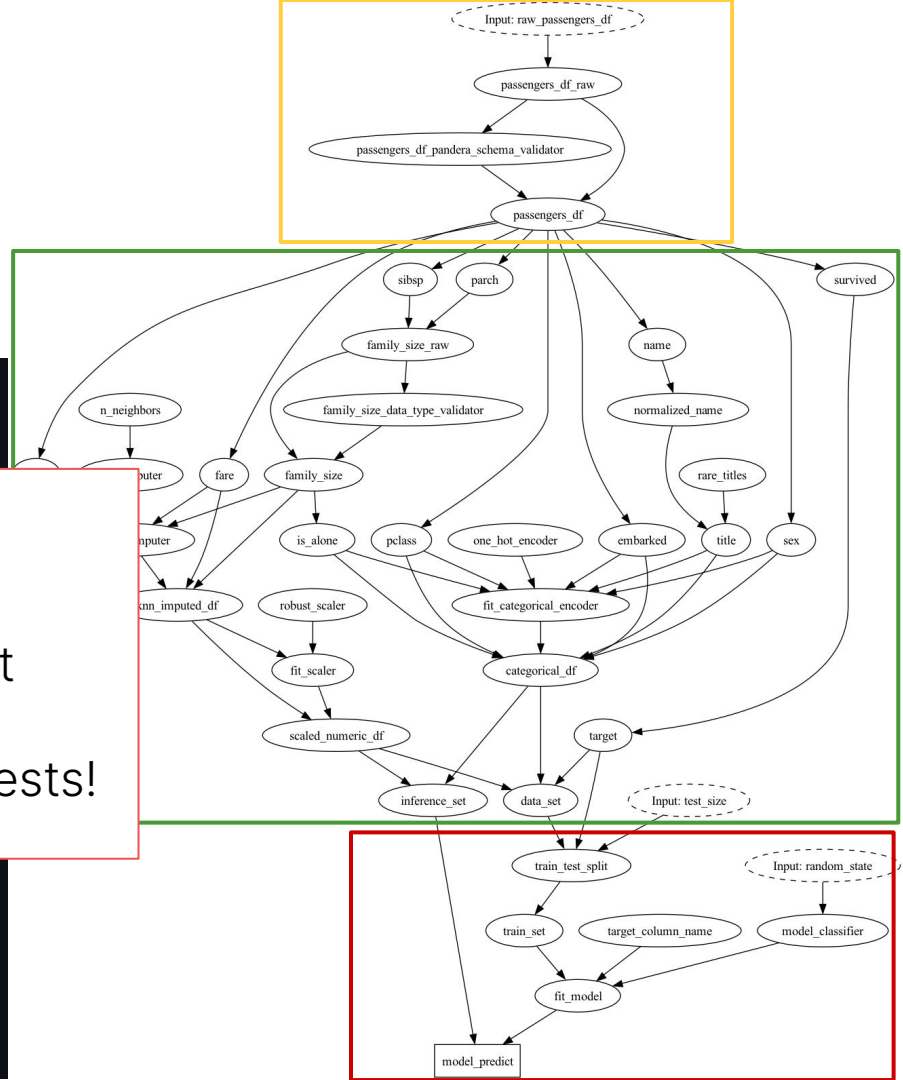
With Hamilton 😎: Testing

```python
import data_loader, feature_transforms, model_pipeline

def model(dbt, session):
  raw_passengers_df = dbt.ref("ra
  # DAG for training/inferring on
  titanic_dag = driver.Driver(con
    data_loader, feature_transfor
    adapter=base.DefaultAdapter()
  )
  # execute & get output
  result = titanic_dag.execute(["
      inputs={"raw_passengers_df"
  )
  # Take the "predictions" result
  # which is an np array
  predictions = result["model_predict"]
  # Return a dataframe!
  return pd.DataFrame(predictions,
                      columns=["prediction"])
```

Outside of dbt:

- unit test
- integration test

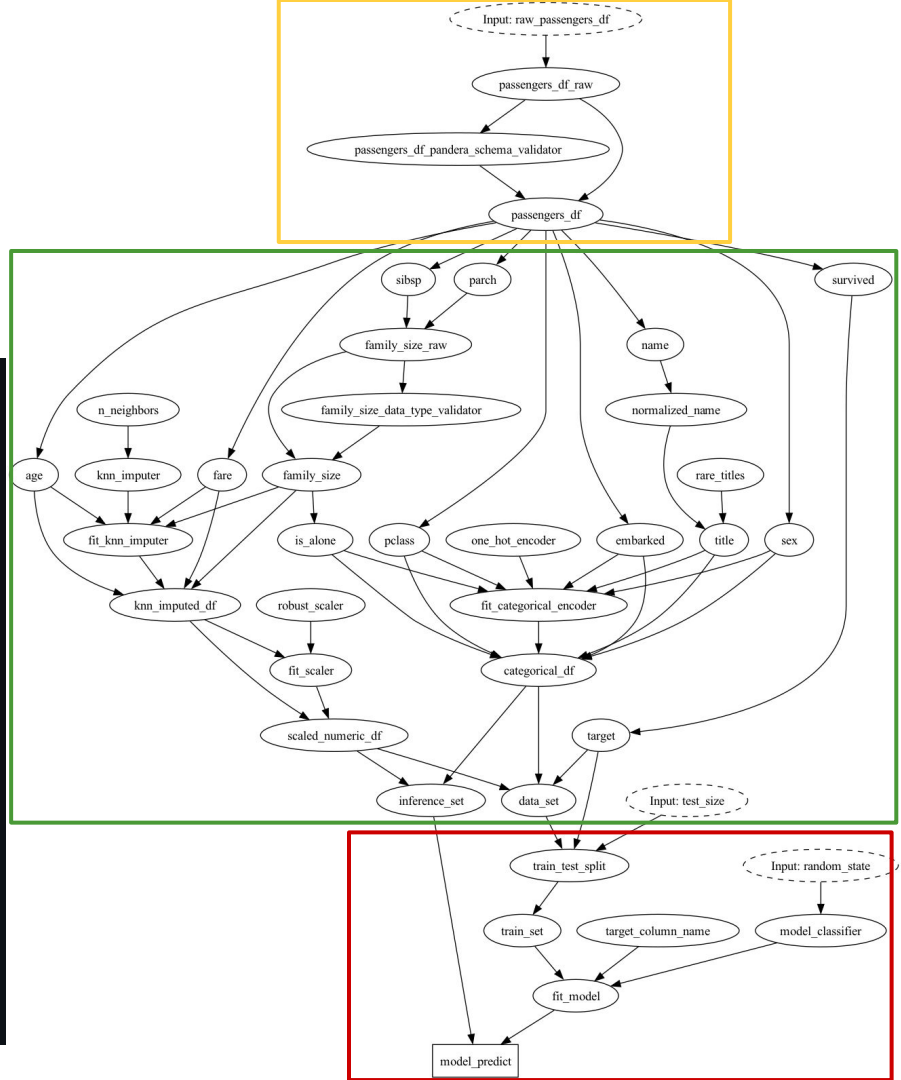
Can also add dbt tests!



Example: doing ML on the Titanic Dataset

With Hamilton 😎: Debugging

```python
import data_loader, feature_transforms, model_pipeline

def model(dbt, session): # can easily create "debug models"
  raw_passengers_df = dbt.ref("raw_passengers")
  # DAG for training/inferring on titanic data
  titanic_dag = driver.Driver(config,
    data_loader, feature_transforms, model_pipeline,
    adapter=base.DefaultAdapter(),
  )
  # execute & get output
  result = titanic_dag.execute(
      ["data_set", "pclass"],
      inputs={"raw_passengers_df": raw_passengers_df}
  )
  # can pull out intermediate nodes easily.
  data_set = result["data_set"]
  print(result["plcass"])
  return data_set
```



Example: doing ML on the Titanic Dataset

Cynic:

- **But how do you package those DS python modules?**

You need a process; this is par for the course with python.

GCP/Databricks - Pyspark:

- Have to install code, or provide it as part of the job submission.

Snowflake - Snowpark:

- Have to “stage” the code.

Run where dbt runs via dbt-fal:

- Can collocate code, or have it installed.

Summary

DAGWORKS

TL;DR: Summary

Hamilton, aka *dbt for Python*, helps you:

1. Scale the Python code within your Python model.
2. Provides a great lineage, testing, and documentation story.
3. Provides a clear interface & process for your DE & DS to collaborate.

Ask: we're looking for people who want to pilot Snowpark dataframe support.

Pro-tip: checkout dbt-fal adapter for more Python flexibility with dbt.

Fin. Thanks for listening!

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

Questions?

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

📚 documentation: hamilton.dagworks.io

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

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

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

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