Feature Engineering with Hamilton:

Portability & Lineage

Stefan Krawczyk
CEO/Co-founder
DAGWORKS

TL;DR:

Hamilton is a ***paradigm*** that can help you:

1. Write features to run in **multiple contexts**.
2. Understand how features (& models) relate with ***lineage.***
3. Keep your code organized/clean.

DAGWORKS

At DAGWorks we're making ML pipelines easy to manage.

Nobody should be afraid to inherit your code.

>>> I'm not selling you anything in this talk! <<<

Hamilton is Open Source!!

I created it while at Stitch Fix: created 2019, OS'ed late 2021.

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

Get started in <15 minutes!

Try it out: https://www.tryhamilton.dev

Documentation: https://hamilton.readthedocs.io

Github: https://github.com/dagworks-inc/hamilton https://www.tryhamilton.dev

Hamilton

Self-documenting, readable, and extensible dataflows.

Learn (5 mins) Github 1.5K+ ⭐

- ☑ Write always unit testable code
- ☑ Add runtime data validation easily
- ☑ Produce readable and maintainable code
- ☑ Visualize lineage (click the run button to see)
- ☑ Run anywhere python runs: in airflow, jupyter, fastapi, etc...
- ☑ Intuitive to use, easy to learn

Try Hamilton right here in your browser 👇

```python
# functions.py - declare and link your transformations as functions....
import pandas as pd

def a(input: pd.Series) -> pd.Series:
    return input % 7

def b(a: pd.Series) -> pd.Series:
    return a * 2

def c(a: pd.Series, b: pd.Series) -> pd.Series:
    return a * 3 + b * 2

def d(c: pd.Series) -> pd.Series:
    return c ** 3
```

```python
# And run them!
import functions
from hamilton import driver
dr = driver.Driver({}, functions)
result = dr.execute(
    ['a', 'b', 'c', 'd'],
    inputs={'input': pd.Series([1, 2, 3, 4, 5])}
)
print(result)
dr.display_all_functions("graph.dot", {})
```

▶ Run me!

The Agenda

Problems with feature engineering
The solution: *Hamilton*
Portability:
- ↳ **Batch**
- ↳ **Streaming / Real-time**

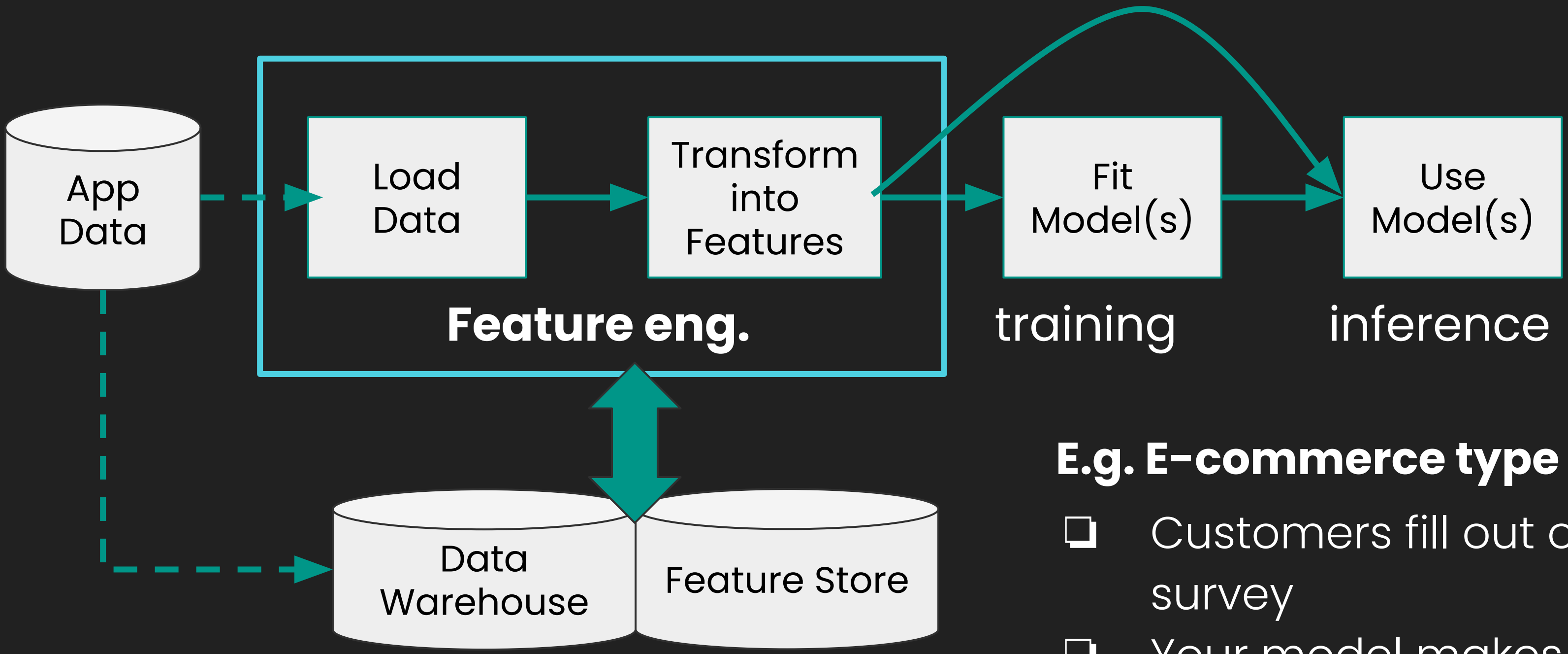
Lineage as Code
Summary & additional benefits of Hamilton
OS progress/updates

The Agenda

Problems with feature engineering
The solution: *Hamilton*
Portability:
- ↳ Batch
- ↳ Streaming / Real-time

Lineage as Code
Summary & additional benefits of Hamilton
OS progress/updates

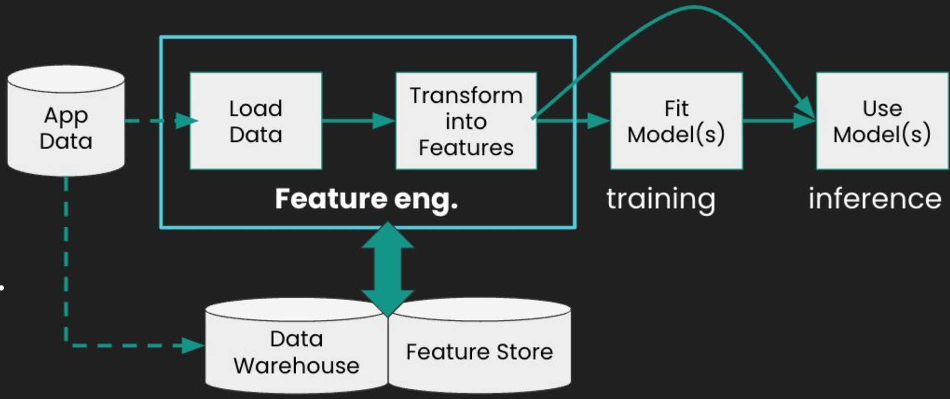
Feature Engineering high level



E.g. E-commerce type scenario

- ❑ Customers fill out onboarding survey
- ❑ Your model makes predictions based on **f(**survey**)**

Problems with Feature Engineering



Challenges:

1. **SLAs & business context**:
    a. Batch vs stream vs real-time.
2. **Training != Inference**:
    a. E.g. aggregations, stores to pull data from.
3. **Observability / Understanding**:
    a. Teams x infra x (Data -> features -> model) connections is non-trivial.

TL;DR: Portability: it's hard to write a feature once

Lineage: it's hard to understand how it all connects

Current Approaches

Context-specific execution ⟷ **Feature DSL to unify**

Challenges:

- Multiple implementations
- Implementations x versions
- Do they match?
- **Cumbersome to manage**

Challenges:

- Single implementation
- Opinionated
- DSL limits expressiveness and use
- **Requires platform team to manage**

Current Approaches

Context-specific execution ⟷ **Feature DSL to unify**

Challenges:

- Multiple implementations
- Implementations x versions
- Do they match?
- **Cumbersome to manage**

Challenges:

- Single implementation
- Opinionated
- DSL limits expressiveness and use
- **Requires platform team to manage**

Q: Is there a solution in the middle?

The Agenda

Problems with feature engineering
The solution: *Hamilton*
Portability:
- ↳ Batch
- ↳ Streaming / Real-time

Lineage as Code
Summary & additional benefits of Hamilton
OS progress/updates

What is Hamilton?

paradigm for defining dataflows
(e.g. feature eng.)

SWE best practices:

✅ testing ✅ documentation
✅ modularity/reuse
✅ data quality ✅ lineage

What is Hamilton?

paradigm for defining dataflows
(e.g. feature eng.)

SWE best practices: ✅ testing ✅ documentation
✅ modularity/reuse
✅ data quality ✅ lineage

Hamilton genesis: the “A-ha” Moment

Problem: Debugging features.

Idea 1:

What if every feature corresponded to **exactly one** python fn?

Idea 2:

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

*In Hamilton, the feature (artifact) is determined by the **name of the function**. Dependencies for computation 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)

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
```

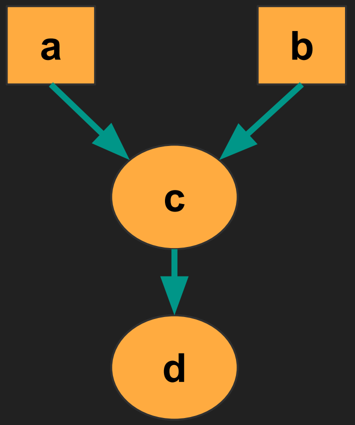
*Hamilton supports *all* python objects, not just dfs/series!*

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
```

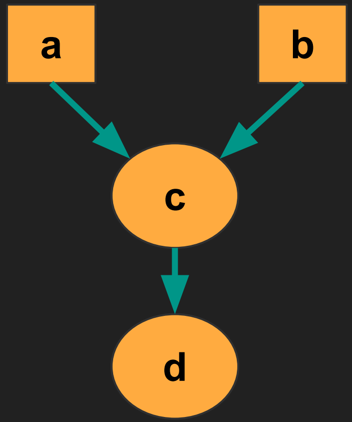


Driver says what/when to execute

```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)
```

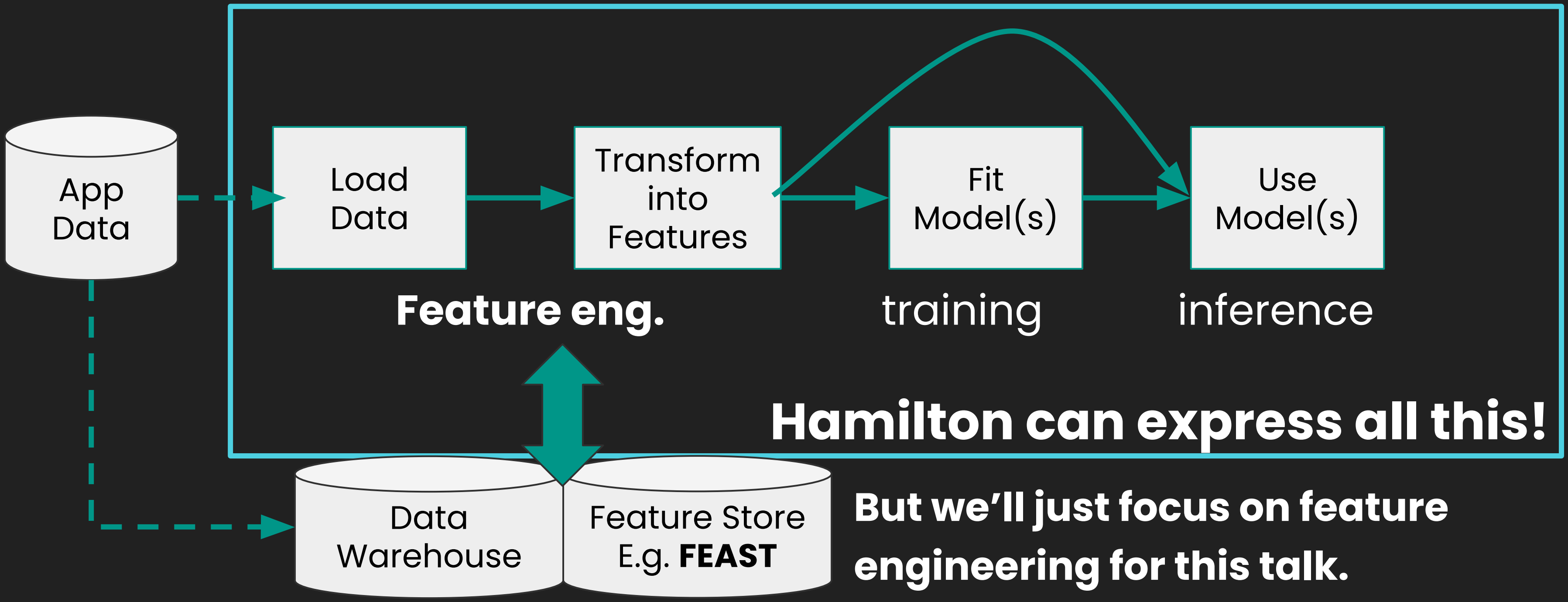
Hamilton: Extending functionality

Decorators:

Syntactic sugar, and add extra expressiveness:

- ❑ `@extract_columns # one dataframe -> multiple series`
- ❑ `@parameterize # curry + repeat a function`
- ❑ `@config.when # conditional - replaces ifs`
- ❑ `@check_output # runtime data validation`
- ❑ `@tag # attach metadata to transforms`
- ❑ `@subdag # recursively utilize groups of nodes`
- ❑ *`@... # and more`*

Hamilton: Feature & Model pipelines



But we'll just focus on feature engineering for this talk.

The Agenda

Problems with feature engineering
The solution: *Hamilton*
Portability:
- ↳ Batch
- ↳ Streaming / Real-time

Lineage as Code
Summary & additional benefits of Hamilton
OS progress/updates

Portability:

How to think about feature functions with Hamilton:

	Batch	Streaming	Online
Map functions	Write once, run everywhere!		
Aggregations	Batch aggregation	Look up / windowed agg.	Look up fixed value
Joins	Batch join	Key-Value lookup	Key-Value lookup

Majority of features are map based!

Portability:

How to think about feature functions with Hamilton:

	Batch	Streaming	Online
Map functions	Write once, run everywhere!		
Aggregations	Batch aggregation	Look up / windowed agg.	Look up fixed value
Joins	Batch join	Key-Value lookup	Key-Value lookup

You choose: store, compute on the fly, update regularly, etc…!
Reimplement only what you need!

Portability:

Let’s write some code; here’s our e-commerce scenario:

- ❑ Simple map operations
  - ❑ raw survey data -> [budget, gender, age]
  - ❑ *derived* features [is_high_roller, is_male, is_female]
- ❑ Joins
  - ❑ time_since_last_login = **f**(client_id, login_data)
- ❑ Aggregations
  - ❑ normalized_age = **g**(mean(age), stddev(age))

The Agenda

Problems with feature engineering
The solution: *Hamilton*
Portability:
- ↳ **Batch**
- ↳ Streaming / Real-time

Lineage as Code
Summary & additional benefits of Hamilton
OS progress/updates

Batch feature engineering

Task

- ❏ Compute features for batch training (& inference)

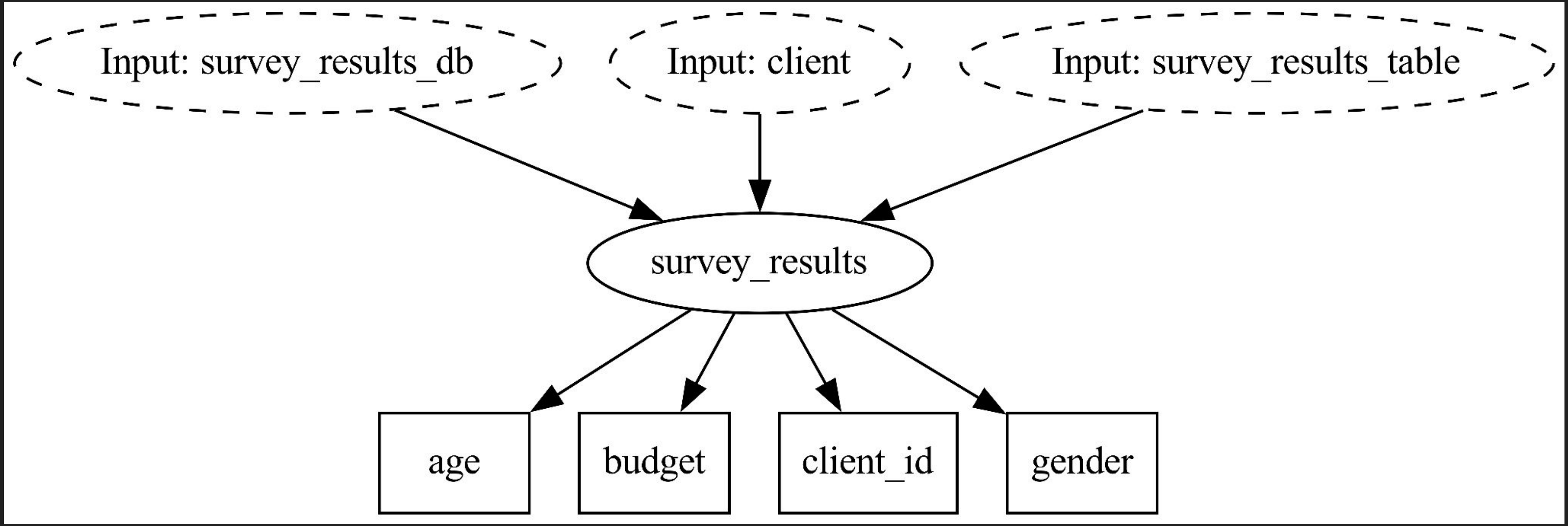
Context

- ❏ DB table with raw survey results
- ❏ DB table with client login data
- ❏ Data is reasonable size *[Hamilton can scale too]*

Data Loading

```python
@extract_columns('budget', 'age', 'gender','client_id')
def survey_results(
        client: connection.Client,
        survey_results_table: str,
        survey_results_db: str) -> pd.DataFrame:
    """Connects to DB and returns table, from which we expose 4 columns."""
    return pd.read_sql(f"SELECT * FROM {survey_results_db}.{survey_results_table}",
            con=client)
```

Data Loading



Map functions

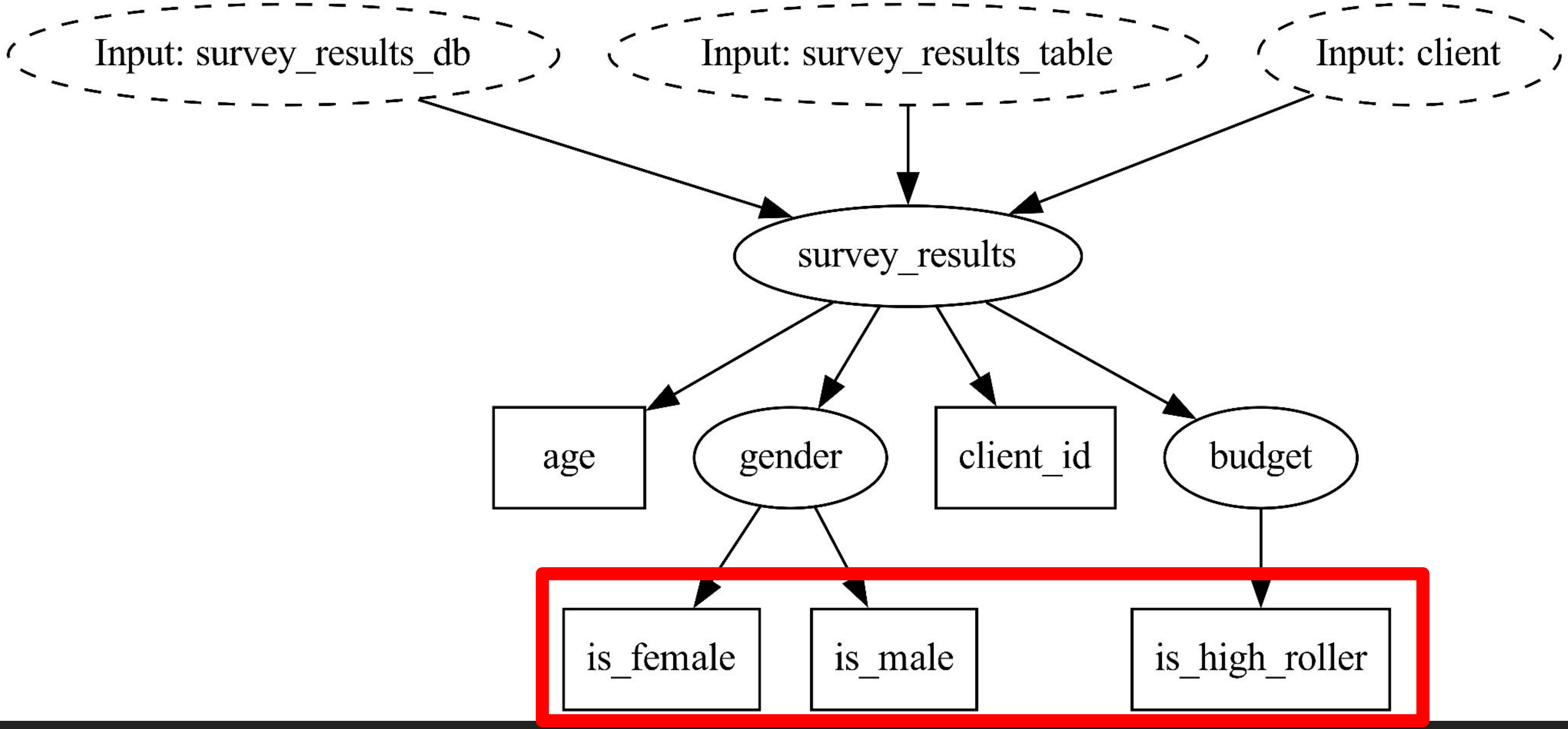
Derived features

```python
def is_male(gender: pd.Series) -> pd.Series:
    return gender == 'male'

def is_female(gender: pd.Series) -> pd.Series:
    return gender == 'female'

def is_high_roller(budget: pd.Series) -> pd.Series:
    return budget > 1000
```

Map functions



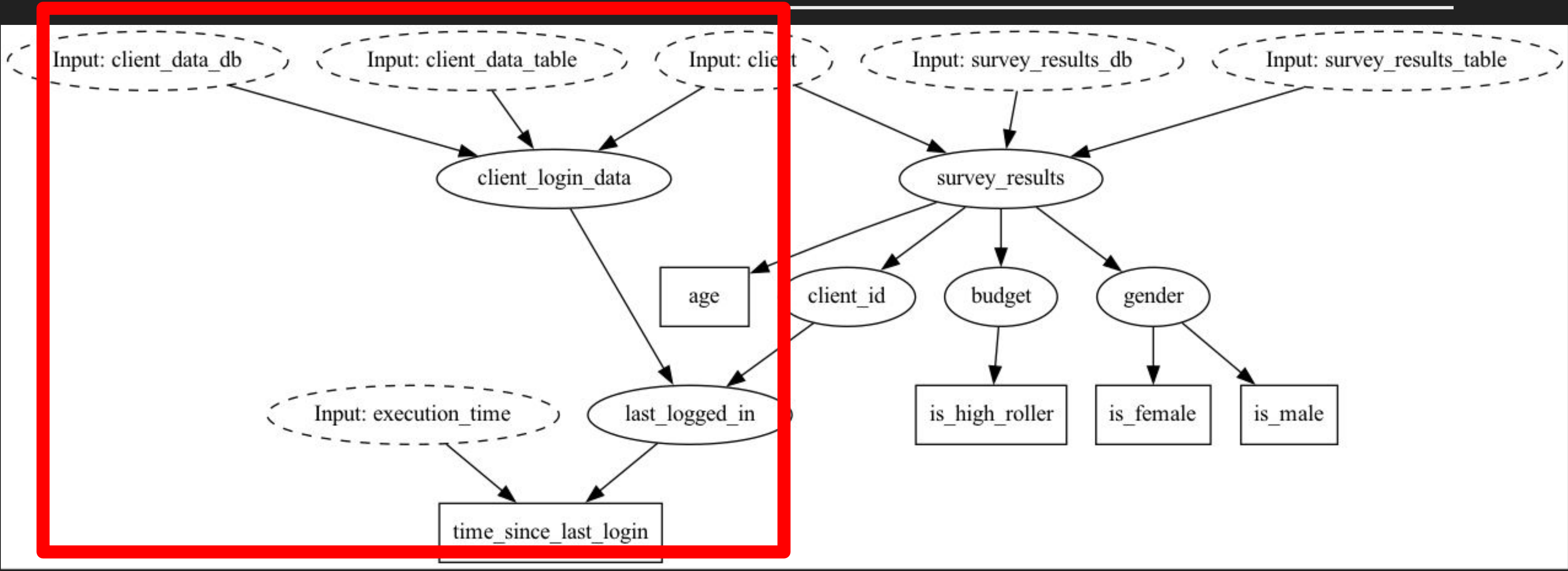
Joins

```python
def client_login_data(
        client: connection.Client,
        client_data_table: str,
        client_data_db: str) -> pd.DataFrame:
    return pd.read_sql(f"SELECT * from {client_data_db}.{client_data_table}", con=client)

def last_logged_in(client_id: pd.Series,
                   client_login_data: pd.DataFrame) -> pd.Series:
    return pd.merge(client_id, client_login_data,
                    left_on='client_id')['last_logged_in']

def time_since_last_login(execution_time: datetime.datetime,
                          last_logged_in: pd.Series) -> pd.Series:
    return execution_time - last_logged_in
```

Joins


Input: client_data_db
Input: client_data_table
Input: clie t
Input: survey_results_db
Input: survey_results_table
client_login_data
survey_results
age
client_id
budget
gender
Input: execution_time
last_logged_in
is_high_roller
is_female
is_male
time_since_last_login

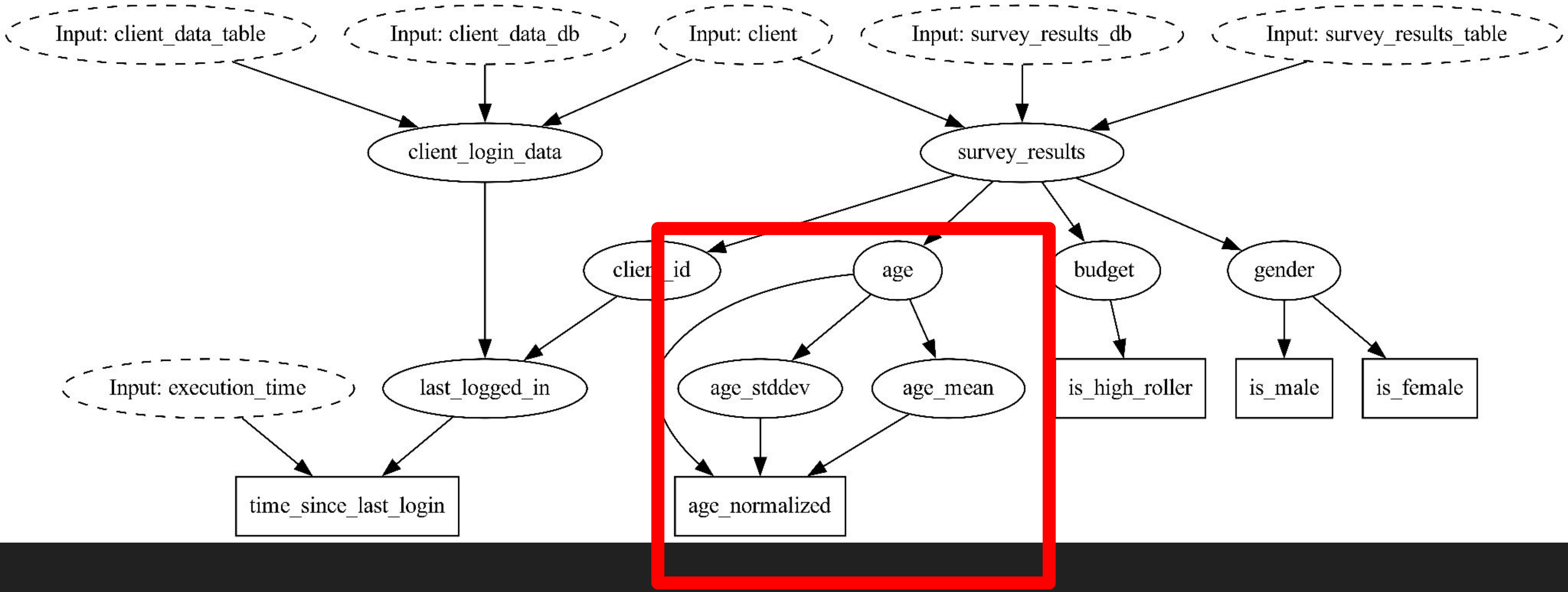
Aggregations

```python
def age_mean(age: pd.Series) -> float:
    return age.mean()

def age_stddev(age: pd.Series) -> float:
    return age.std()

def age_normalized(
        age: pd.Series,
        age_mean: float,
        age_stddev: float) -> pd.Series:
    return (age - age_mean)/age_stddev
```

Aggregations

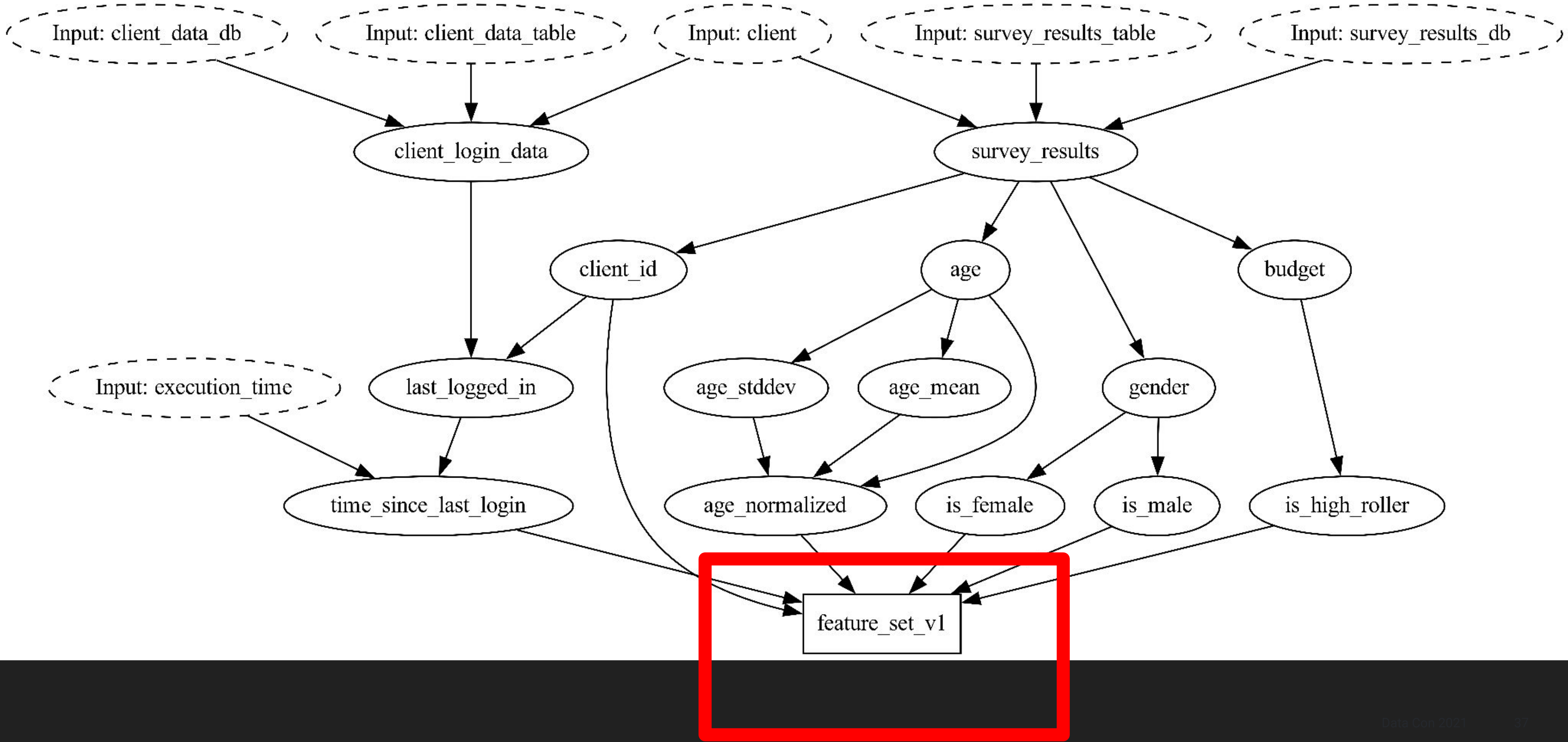


Data Set Creation

```python
def feature_set_v1(
    age_normalized: pd.Series,
    is_high_roller: pd.Series,
    is_male: pd.Series,
    is_female: pd.Series,
    time_since_last_login: pd.Series) -> pd.DataFrame:
    """V1 of our feature set."""
    return pd.DataFrame(...)
```

Note: you could also request this same feature set be created via the “driver”.

Batch feature engineering for training & inference



Driver

```python
#etl.py
from project import load_data, map_features, join_features, agg_features, data_sets
model = …  # instantiate a model
target = … # pull target data …
# create the DAG
dr = driver.Driver({}, load_data, map_features, join_features, agg_features, data_sets)

inputs = {
    "survey_results_table" : ...,
    "survey_results_db" : ...,
    "execution_time" : datetime.datetime.now(),
    "client_data_table" : ...,
    "client_data_db": ...,
}
df = dr.execute(['feature_set_v1'], inputs=inputs)
model = model.fit(df, target) # or model.predict(df) …
```

The Agenda

Problems with feature engineering
The solution: *Hamilton*
Portability:
- ↳ **Batch**
- ↳ **Streaming / Real-time**

Lineage as Code
Summary & additional benefits of Hamilton
OS progress/updates

Streaming / Real-time Features

Task

- ❏ Compute features for inference (or push to feature store)

Context

- ❏ Survey event comes in on a stream/request
- ❏ Have service to give client login data
- ❏ Have stored aggregations from training

Changes required

- ❏ Swap out nodes that load data
- ❏ Aggregation doesn’t make sense - use values from training

E.g. for streaming context (real-time similar)

@config.when swap out features you need to change:

```python
@extract_columns('budget', 'age', 'gender', 'client_id')
@config.when(mode='streaming')
def survey_results__streaming(survey_records: list[dict]) -> pd.DataFrame:
    return pd.DataFrame.from_records(survey_records)


@config.when(mode='streaming')
def last_logged_in__streaming(client_id: pd.Series, client: connection.Client) ->
pd.Series:
    return pd.Series(client.query(ids=client_id.values()))
```

Note: our batch features should have a similar `@config.when` annotation

E.g. for streaming context (real-time similar)

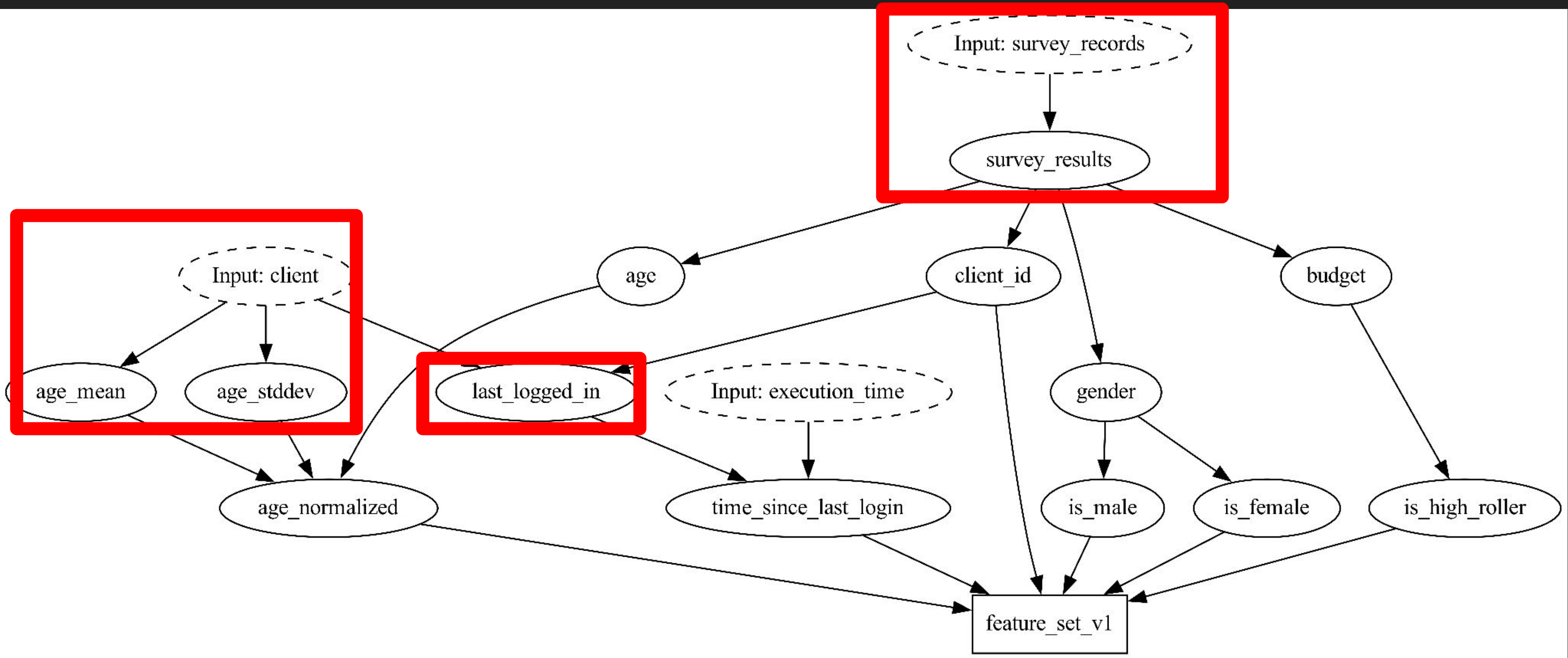
@config.when swap out features you need to change:

```python
@config.when(mode='streaming')
def age_mean__streaming(client: connection.Client) -> float:
    return client.query('age_mean')


@config.when(mode='streaming')
def age_stddev__streaming(client: connection.Client) -> float:
    return client.query('age_stddev')
```

Note: our batch features should have a similar `@config.when` annotation

Tying it together...



Streaming Driver Code

```python
# processor.py
from project import load_data, map_features, join_features, agg_features, data_set

config = {'mode' : 'streaming'}
dr = driver.Driver(config,
                   load_data, map_features, join_features,agg_features, data_set)
model = load_model(...)

def process_records(records: list[dict]) -> list[float]:
    inputs = {
        "records" : records,
        "execution_time" : datetime.datetime.now(),
        "client" : some_client(),
    }
   df = dr.execute(['feature_set_v1'], inputs=inputs)
   return model.predict(df).values
```

Real-time Driver Code

```python
# app.py
from project import load_data, map_features, join_features, agg_features, data_set
app = ... # webservice app
model_obj = ... # load model somehow
config = {'mode' : 'real-time'}
dr = driver.AsyncDriver(config,
                        load_data, map_features, join_features,agg_features, data_set)

@app.post("/predict")
async def predict(record: PredictRequest) -> float:
   inputs = {
        "records" : [record.to_dict()],
        "execution_time" : datetime.datetime.now(),
        "client" : some_async_client(),
   }
   df = await dr.execute(['feature_set_v1'], inputs=inputs)
   return model.predict(df).values
```

The Agenda

Problems with feature engineering
The solution: *Hamilton*
Portability:
- ↳ **Batch**
- ↳ **Streaming / Real-time**

Lineage as Code
Summary & additional benefits of Hamilton
OS progress/updates

Lineage

Lineage definition:

“historical record or traceability of data as it is transformed”

Why it’s important/useful:

- ❏ GDPR / compliance
- ❏ Collaboration:
    - ❏ Debugging
    - ❏ Onboarding/offboarding
- ❏ Reducing outages / MTTR

Lineage

Challenges

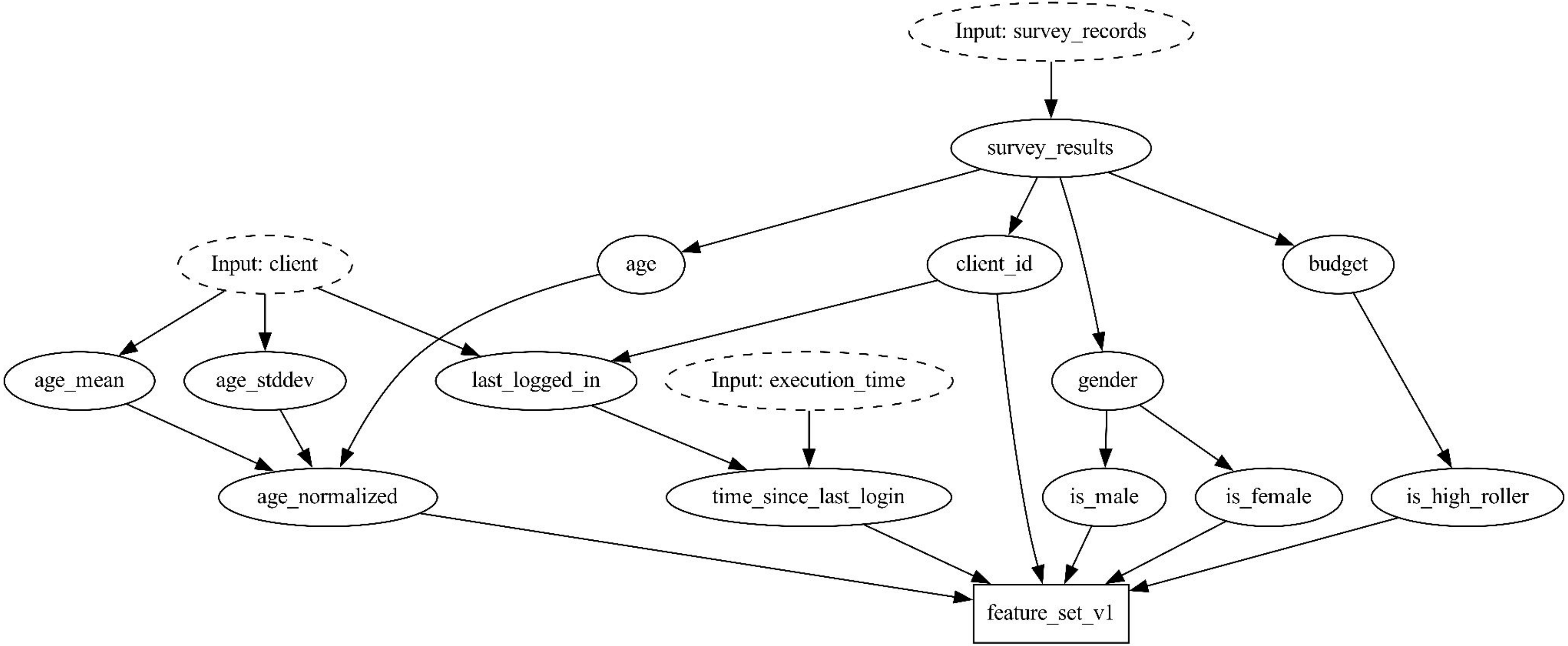
- ❑ Most people generally don't have feature lineage.
- ❑ Requires extra systems & engineering effort.

Current solutions

- ❑ Open lineage + data hub.
- ❑ Manual documentation.

but then there's Hamilton: Lineage as Code

dr.visualize_execution(…)



Lineage as Code

What you get with Hamilton

- ❑ Code defines how things connect → lineage
- ❑ Couple with git == lightweight lineage
- ❑ Couple with @tag == can ask questions of the DAG

Changes required

- ❑ None, apart from adding @tag to functions

Lineage as Code

What you can do with Hamilton

- E.g. Annotate with:
  - PII, team, source, extra info, etc..

```python
@tag(
    PII="true",
    source="prod.surveys",
    owner="data-engineering",
    importance="production",
    info="https://internal.wikipage
)
def my_func(...)
```

Questions you can answer

- Who owns this feature?
- How is feature X computed?
- Where is age used?
- What sources did I train on?

```python
dr.visualize_execution(["X"], ...)
```

```python
nodes = dr.what_is_downstream_of("age")
```

```python
nodes = dr.what_is_upstream_of("model")
sources = [n for n in nodes if nodes.tags.get("source")...]
```

The Agenda

Problems with feature engineering
The solution: *Hamilton*
Portability:

- ↳ Batch
- ↳ Streaming / Real-time

Lineage as Code
Summary & additional benefits of Hamilton
OS progress/updates

Summary: write Hamilton functions

```
# 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
```

And you get...

- Portability/modularity/reuse
- Lineage as code
- Unit & Integration testing
- Documentation
- Data quality
- Feature definition catalog

✅ module curation & decoupled drivers; extensibility & decorators
✅ know how code & data relate
✅ always possible, straightforward
✅ tags, lineage, function doc
✅ runtime checks
✅ naming, curation, versioning

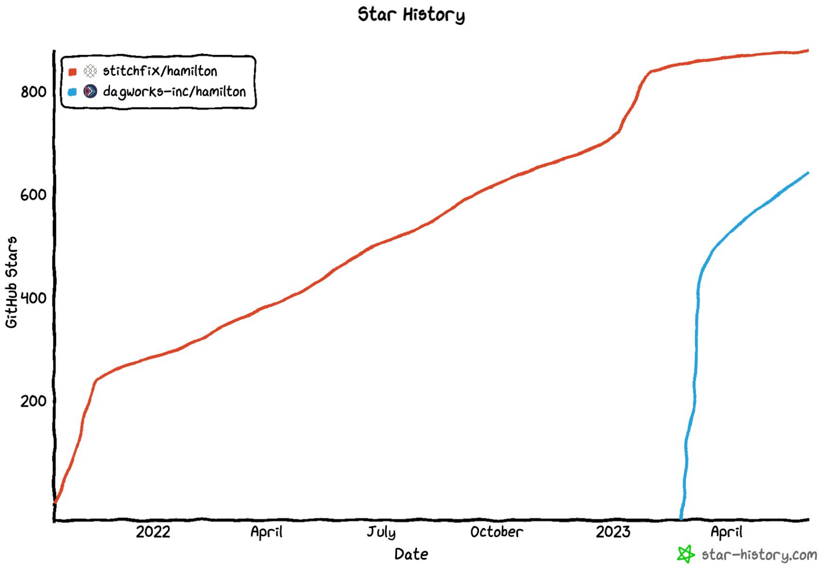
The Agenda

Problems with feature engineering
The solution: *Hamilton*
Portability:
- ↳ Batch
- ↳ Streaming / Real-time

Lineage as Code
Summary & additional benefits of Hamilton
OS progress/updates

OS Progress

~1.4K+ Unique Stargazers
150+ slack members
72K+ downloads



OS used by lots of companies like:

OS Roadmap

A few things we're thinking about:

- ❑ Hamilton compile -> orchestration system
  - ❑ E.g. Hamilton -> Airflow
- ❑ Generator support for mini-batch processing large datasets
- ❑ Extending pyspark integration beyond map functions.
- ❑ Connectors to common MLOps tools
- ❑ <Your idea here!>

Give Hamilton a Try! We'd Love Your Feedback.

www.tryhamilton.dev

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

⭐ on github (https://github.com/dagworks-inc/hamilton)

☑️ create & vote on issues on github

📣 join us on on Slack

Kösz!

Questions?

- https://twitter.com/stefkrawczyk
- https://www.linkedin.com/in/skrawczyk
- https://github.com/dagworks-inc/hamilton
- stefan@dagworks.io