DAGWORKS

# Hamilton:
# Natively bringing SWE best practice to Python data transformations

November 2023 **@ Scale By The Bay**

**Stefan Krawczyk** - DAGWorks Inc.

# whoami

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

## 12+ years in ML & Data platforms

STITCH FIX

IDIBON

HRI
Honda Research Institute US

IBM

VICTORIA UNIVERSITY OF
WELLINGTON
TE HERENGA WAKA

# Why do SWE principles matter?

# Why do SWE principles matter?

It helps scale/amplify human efforts; & humans are $$$.

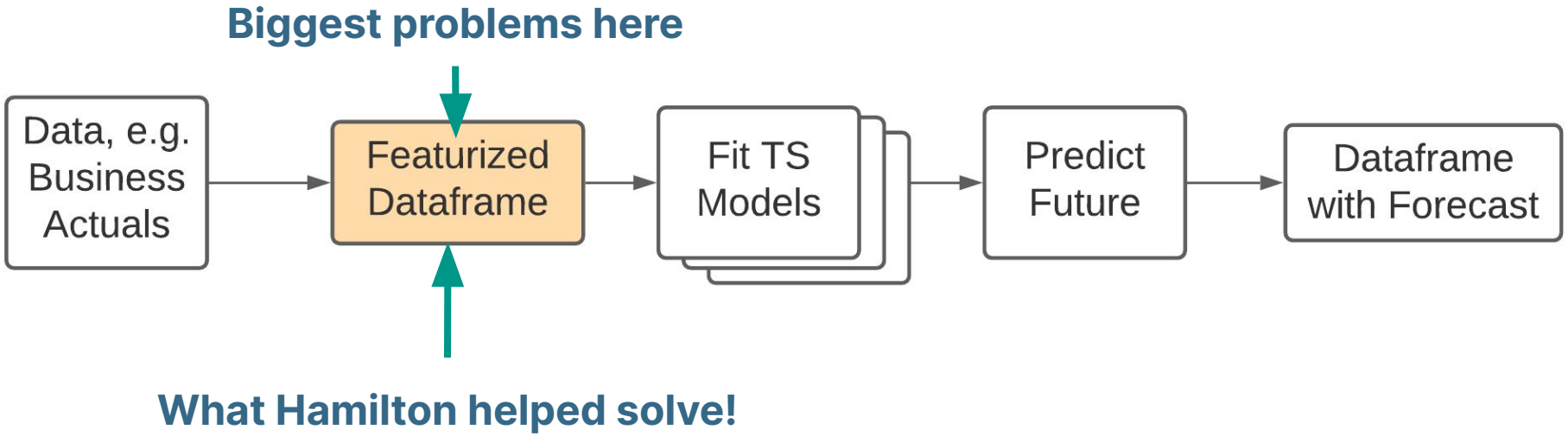
# Hamilton

Natively bringing SWE best practice to Python data transformations

1. **Motivating pain**
2. Hamilton
3. General Usage
4. Native SWE
5. Summary

# Motivating Pain

- You're a DS team that provides operational forecasts for the business.
- The business makes decisions based on your numbers.
- You need to constantly model change in the world.



# Example: Creating a dataframe (e.g. for ML training)

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

# Example: Creating a dataframe (e.g. for ML training)

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

# Example: Creating a dataframe (e.g. for ML training)

```
df = loader.load_actuals(dates) # e.g. spend, signups
if config['region'] == 'UK':
   df['holidays'] = is_uk_holiday(df['year'], df[' week'])
else:
   df['holidays'] = is_holiday(df['year'], df['week'])
```

# Example: Creating a dataframe (e.g. for ML training)

```
df = loader.load_actuals(dates) # e.g. spend, signups
if config['region'] == 'UK':
   df['holidays'] = is_uk_holiday(df['year'], df[' week'])
else:
   df['holidays'] = is_holiday(df['year'], df['week'])
df['avg_3wk_spend'] = df['spend'].rolling(3).mean()
df['acquisition_cost'] = df['spend'] / df['signups']
df['spend_shift_3weeks'] = df['spend'].shift(3)
```

# Example: Creating a dataframe (e.g. for ML training)

```
df = loader.load_actuals(dates) # e.g. spend, signups
if config['region'] == 'UK':
   df['holidays'] = is_uk_holiday(df['year'], df[' week'])
else:
   df['holidays'] = is_holiday(df['year'], df['week'])
df['avg_3wk_spend'] = df['spend'].rolling(3).mean()
df['acquisition_cost'] = df['spend'] / df['signups']
df['spend_shift_3weeks'] = df['spend'].shift(3)
df['special_feature1'] = compute_bespoke_feature(df)
df['spend_b'] = multiply_columns(df['acquisition_cost'], df['B'])
```

# Example: Creating a dataframe (e.g. for ML training)

```
df = loader.load_actuals(dates) # e.g. spend, signups
if config['region'] == 'UK':
   df['holidays'] = is_uk_holiday(df['year'], df[' week'])
else:
   df['holidays'] = is_holiday(df['year'], df['week'])
df['avg_3wk_spend'] = df['spend'].rolling(3).mean()
df['acquisition_cost'] = df['spend'] / df['signups']
df['spend_shift_3weeks'] = df['spend'].shift(3)
df['special_feature1'] = compute_bespoke_feature(df)
df['spend_b'] = multiply_columns(df['acquisition_cost'], df['B'])
save_df(df, "some_location")
```

# Example: Creating a dataframe (e.g. for ML training)

```
df = loader.load_actuals(dates) # e.g. spend, signups
if config['region'] == 'UK':
   df['holidays'] = is_uk_holiday(df['year'], df[' week'])
else:
   df['holidays'] = is_holiday(df['year'], df['week'])
df['avg_3wk_spend'] = df['spend'].rolling(3).mean()
df['acquisition_cost'] = df['spend'] / df['signups']
df['spend_shift_3weeks'] = df['spend'].shift(3)
df['special_feature1'] = compute_bespoke_feature(df)
df['spend_b'] = multiply_columns(df['acquisition_cost'], df['B'])
save_df(df, "some_location")
```

😬 Now picture the passage of time: personnel Δ, sophistication ⬆️, etc

# Problem: unit & integration testing; data quality 👎

```
df = loader.load_actuals(dates) # e.g. spend, signups
if config['region'] == 'UK':
   df['holidays'] = is_uk_holiday(df['year'], df[' week'])
else:
   df['holidays'] = is_holiday(df['year'], df['week'])
df['avg_3wk_spend'] = df['spend'].rolling(3).mean()
df['acquisition_cost'] = df['spend'] / df['signups']
df['spend_shift_3weeks'] = df['spend'].shift(3)
df['special_feature1'] = compute_bespoke_feature(df)
df['spend_b'] = multiply_columns(df['acquisition_cost'], df['B'])
save_df(df, "some_location")
```

😬 Now picture the passage of time: personnel Δ, sophistication ⬆️, etc

# Problem: code readability & documentation 🧐

```
df = loader.load_actuals(dates) # e.g. spend, signups
if config['region'] == 'UK':
   df['holidays'] = is_uk_holiday(df['year'], df[' week'])
else:
   df['holidays'] = is_holiday(df['year'], df['week'])
df['avg_3wk_spend'] = df['spend'].rolling(3).mean()
df['acquisition_cost'] = df['spend'] / df['signups']
df['spend_shift_3weeks'] = df['spend'].shift(3)
df['special_feature1'] = compute_bespoke_feature(df)
df['spend_b'] = multiply_columns(df['acquisition_cost'], df['B'])
save_df(df, "some_location")
```

?

😬 Now picture the passage of time: personnel Δ, sophistication ⬆️, etc

# Problem: difficulty in tracing lineage 🤯

```
df = loader.load_actuals(dates) # e.g. spend, signups
if config['region'] == 'UK':
   df['holidays'] = is_uk_holiday(df['year'], df[' week'])
else:
   df['holidays'] = is_holiday(df['year'], df['week'])
df['avg_3wk_spend'] = df['spend'].rolling(3).mean()
df['acquisition_cost'] = df['spend'] / df['signups']
df['spend_shift_3weeks'] = df['spend'].shift(3)
df['special_feature1'] = compute_bespoke_feature(df)
df['spend_b'] = multiply_columns(df['acquisition_cost'], df['B'])
save_df(df, "some_location")
```

😬 Now picture the passage of time: personnel Δ, sophistication ⬆️, etc

# Problem: code reuse and duplication

```
df = loader.load_actuals(dates) # e.g. spend, signups
if config['region'] == 'UK':
    df['holidays'] = is_uk_holiday(df['year'], df[' week'])
else:
    df['holidays'] = is_holiday(df['year'], df['week'])
df['avg_3wk_spend'] = df['spend'].rolling(3).mean()
df['acquisition_cost'] = df['spend'] / df['signups']
df['spend_shift_3weeks'] = df['spend'].shift(3)
df['special_feature1'] = compute_bespoke_feature(df)
df['spend_b'] = multiply_columns(df['acquisition_cost'], df['B'])
save_df(df, "some_location")
```

😬 Now picture the passage of time: personnel Δ, sophistication ⬆️, etc

# Problem: onboarding 📈 & debugging 📈

```
df = loader.load_actuals(dates) # e.g. spend, signups
if config['region'] == 'UK':
   df['holidays'] = is_uk_holiday(df['year'], df[' week'])
else:
   df['holidays'] = is_holiday(df['year'], df['week'])
df['avg_3wk_spend'] = df['spend'].rolling(3).mean()
df['acquisition_cost'] = df['spend'] / df['signups']
df['spend_shift_3weeks'] = df['spend'].shift(3)
df['special_feature1'] = compute_bespoke_feature(df)
df['spend_b'] = multiply_columns(df['acquisition_cost'], df['B'])
save_df(df, "some_location")
```

😬 Now picture the passage of time: personnel Δ, sophistication ⬆️, etc

# Question for you!

1. Are any of these pains familiar to you? If so, which ones?
2. Would you want to inherit code like this?
3. [Independently] would you be in anguish if you suddenly had to inherit your colleagues code?

✋ Raise hand | ~~Unmute~~ !

# Hamilton

Natively bringing SWE best practice to Python data transformations

1. ~~Motivating pain~~
2. **Hamilton**
3. General Usage
4. Native SWE
5. Summary

# 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?

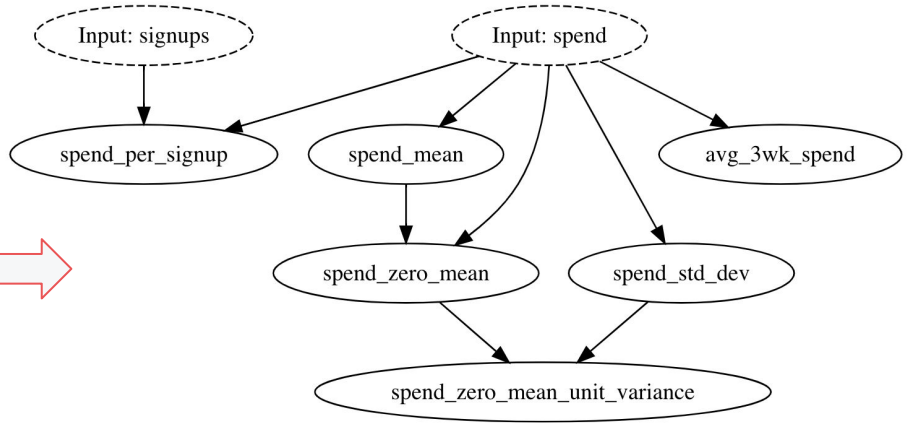
**Fancy way of saying:**

## 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 Paradigm:

Instead of

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

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

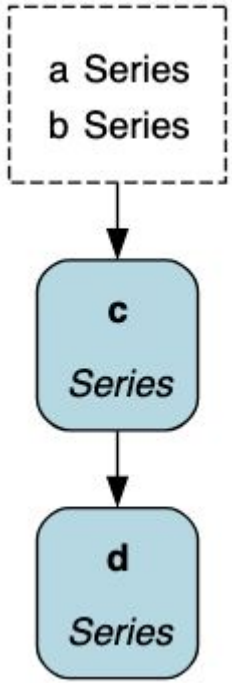
You declare

```
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)
```

# Things to mention, but I won't cover:

We also have decorators that you add to functions that...

- `@tag` `# attach metadata`
- `@parameterize` `# curry + repeat a function`
- `@extract_columns` `# one dataframe -> multiple series`
- `@extract_outputs` `# one dict -> multiple outputs`
- `@check_output` `# data validation; very lightweight`
- `@config.when` `# conditional transforms`
- `@subdag` `# parameterize parts of your DAG`

& more... Hamilton code is **portable** & runs **& scales** anywhere python runs.

# Hamilton @ Stitch Fix

Running in production since 2019

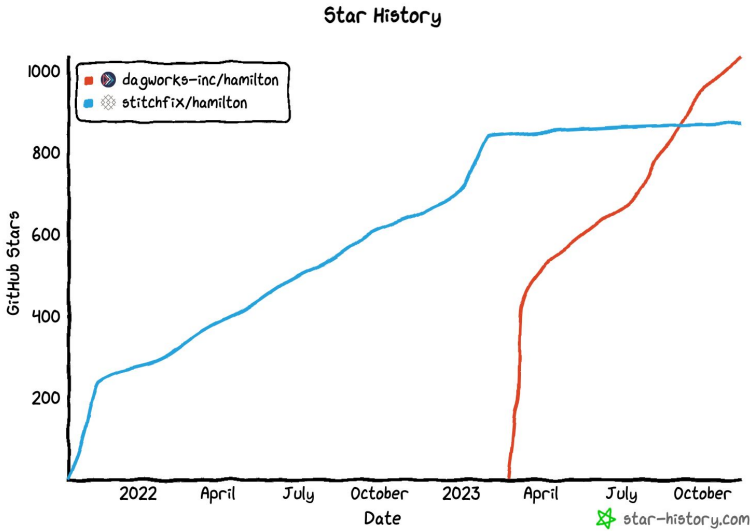
One team manages **4000+** feature definitions

Data science teams ❤ it

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

# Some Hamilton stats

**~1.7K Unique Stargazers**
**250+ slack members**
**125K+ 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

Natively bringing SWE best practice to Python data transformations

1. ~~Motivating pain~~
2. ~~Hamilton~~
3. **General Usage**
4. Native SWE
5. Summary

# When ***should I*** consider Hamilton?

If you can draw a flowchart (DAG), you can put it into Hamilton:

- Feature engineering (origin, especially time-series)
- Tired of managing scripts that do transformations...
- Describing E2E ML Pipelines + MLOps integrations
- Web request flows
- LLM Workflows! (e.g. replace langchain)

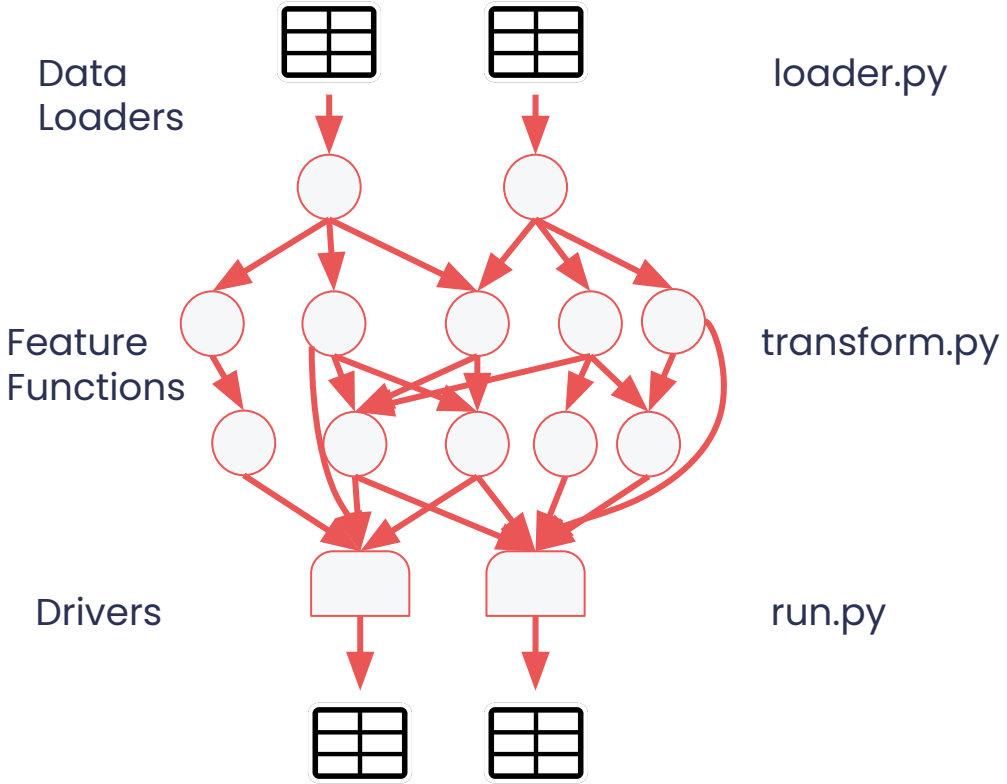
Code & software best practices enthusiasts:

- Hamilton ⬇️ Code Complexity

# Example Hamilton use case: Feature Engineering

Code that needs to be written:

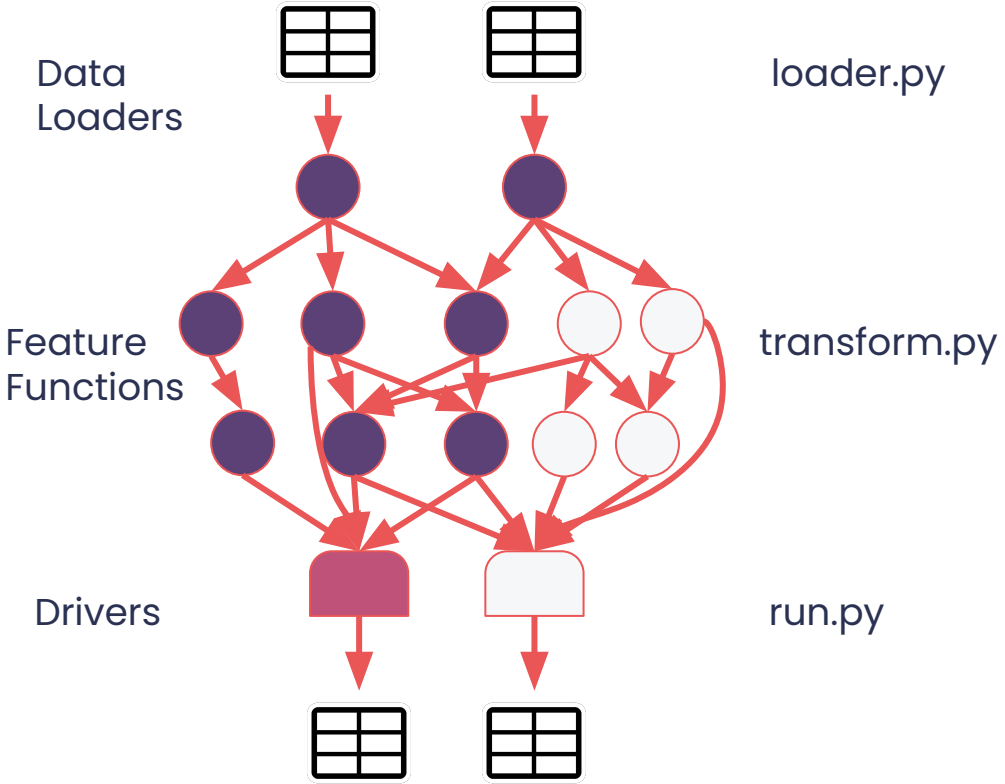
1. Functions to load data
2. Feature functions
3. Drivers materialize data



# Example Hamilton use case: Feature Engineering

Code that needs to be written:

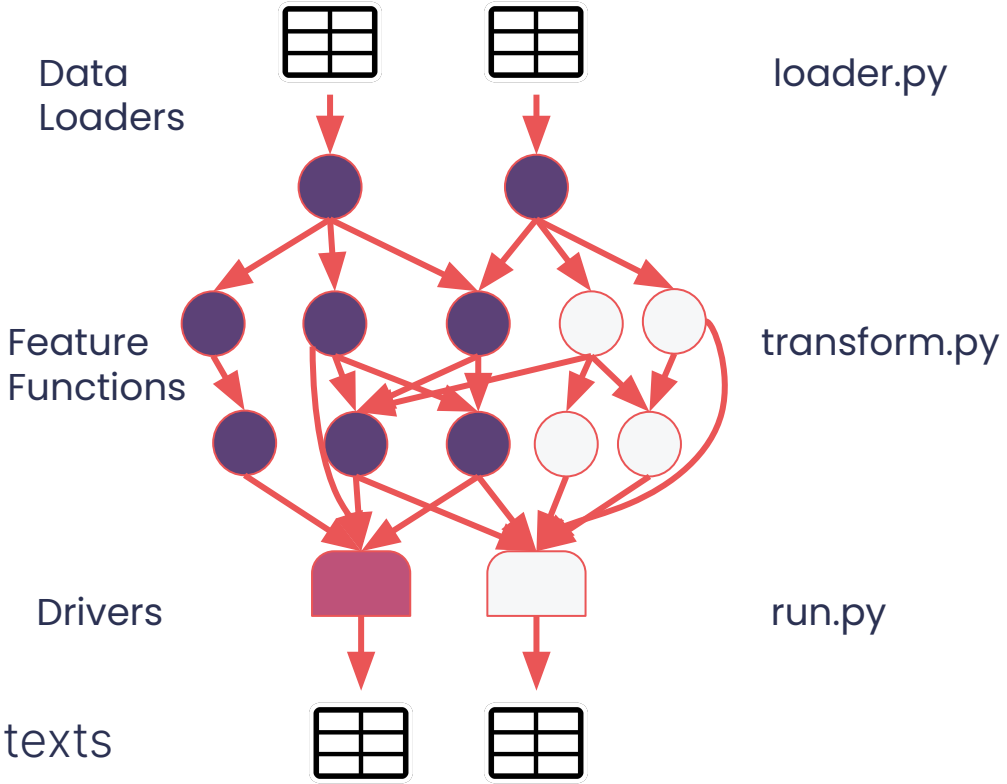
1. Functions to load data
2. Feature functions
3. Drivers materialize data



# Example Hamilton use case: Feature Engineering

Code that needs to be written:

1. Functions to load data
2. Feature functions
3. Drivers materialize data




**Code base implications:**

- Natural structure emerges
- Logic modules vs execution contexts

# Hamilton

Natively bringing SWE best practice to Python data transformations

1. ~~Motivating pain~~
2. ~~Hamilton~~
3. ~~General Usage~~
4. **Native SWE**
5. Summary

# Native SWE

5 common ideals
that Hamilton guides
you towards

- Testing & Documentation
- KISS
- YAGNI
- DRY
- SOLID

# Native SWE

5 common ideals
that Hamilton guides
you towards

- **Testing & Documentation**
- KISS – will skip
- YAGNI – will skip
- DRY – will skip
- **SOLID***

# General: Testing & Documentation

# General: Testing & Documentation

```
# client_features.py

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

**Testing**: easier to unit & integration test.

```
# test_client_features.py

def test_height_zero_mean_unit_variance():
    actual = height_zero_mean_unit_variance(pd.Series([1,2,3]), 2.0)
    expected = pd.Series([0.5,1.0, 1.5])
    assert actual == expected
```

# General: Testing & Documentation

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

**Testing**: easier to unit & integration test.

**Data Quality Tests:** runtime checks via annotation*; Pandera supported.

# General: Testing & Documentation

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

**Testing**: easier to unit & integration test.

**Data Quality Tests:** runtime checks via annotation*; Pandera supported.

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

# General: Testing & Documentation

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

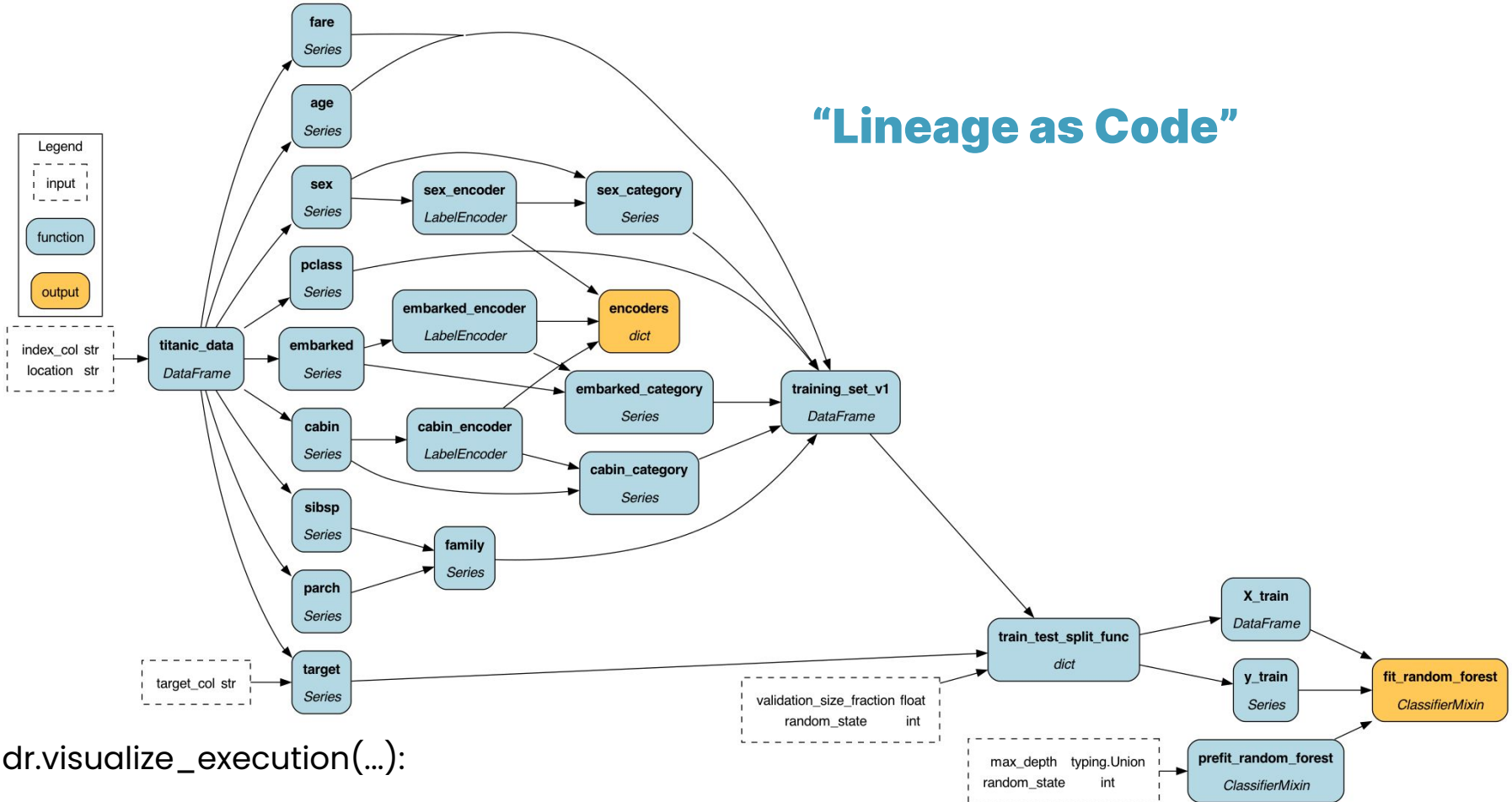
**Testing**: easier to unit & integration test.

**Data Quality Tests:** runtime checks via annotation*; Pandera supported.

**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(...):

# Comparing to the code from earlier:

```
df = loader.load_actuals(dates) # e.g. spend, signups
if config['region'] == 'UK':
   df['holidays'] = is_uk_holiday(df['year'], df[' week'])
else:
   df['holidays'] = is_holiday(df['year'], df['week'])
df['avg_3wk_spend'] = df['spend'].rolling(3).mean()
df['acquisition_cost'] = df['spend'] / df['signups']
df['spend_shift_3weeks'] = df['spend'].shift(3)
df['special_feature1'] = compute_bespoke_feature(df)
df['spend_b'] = multiply_columns(df['acquisition_cost'], df['B'])
save_df(df, "some_location")
```

👎 : testing, documentation, visualization...

# Native SWE

5 common ideals
that Hamilton guides
you towards

- ~~Testing & Documentation~~
- ~~KISS~~
- ~~YAGNI~~
- ~~DRY~~
- **SOLID***

# 🤔 Think about your codebase. How does your code compare?

# SOLID Principles:
# OO origin. Follow → 💰

## This is my take on it applied to dataflows.

# SOLID Principles:
# Single Responsibility Principle

code should do one thing

# Code should do one thing

## Functions: single task; “named piece of business logic”

```
def embedding(query: str) -> List[float]:
  response = openai.Embedding.create(input=query, model="HARDCODED")
  return response["data"][0]["embedding"]
```

## Driver: no logic; just handling context of what & where

```
dr = driver.Driver(config, module1, module2)
outputs = ["spend", "signups", ...]
result = dr.execute(outputs,inputs=input_data)
```

→ Net result: execution “decoupled” from “logic”.

→ Easier to test, maintain, and change.

# SOLID Principles:
# Open Closed Principle

**easy to extend, hard to break;**
**→ prevent issues with code evolution**

# Easy to extend

```
def embedding(query: str) -> List[float]:
  response = openai.Embedding.create(input=query, model="HARDCODED")
  return response["data"][0]["embedding"]


def nn_ids(
 embedding: List[float], vectordb_client: Client, top_k: int) -> List[int]:
  results = vectordb_client.search(embedding=embedding, top_k=top_k)
  return results
```

# Easy to extend

```
def embedding(query: str) -> List[float]:
  response = openai.Embedding.create(input=query, model="HARDCODED")
  return response["data"][0]["embedding"]


def nn_ids(
 embedding: List[float], vectordb_client: Client, top_k: int) -> List[int]:
  results = vectordb_client.search(embedding=embedding, top_k=top_k)
  return results
```

```
-------------------------------------------------------------------------
def nn_text(nn_ids: List[int], some_other_args: dict) -> Dict[int, str]:
  results = {nn_id: _get_text(nn_id) for nn_id in nn_ids}
  ...
  return results
```

# Hard to break

**Hard to break existing logic; or if you do, it's clear why.**

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

**Things Hamilton checks:**

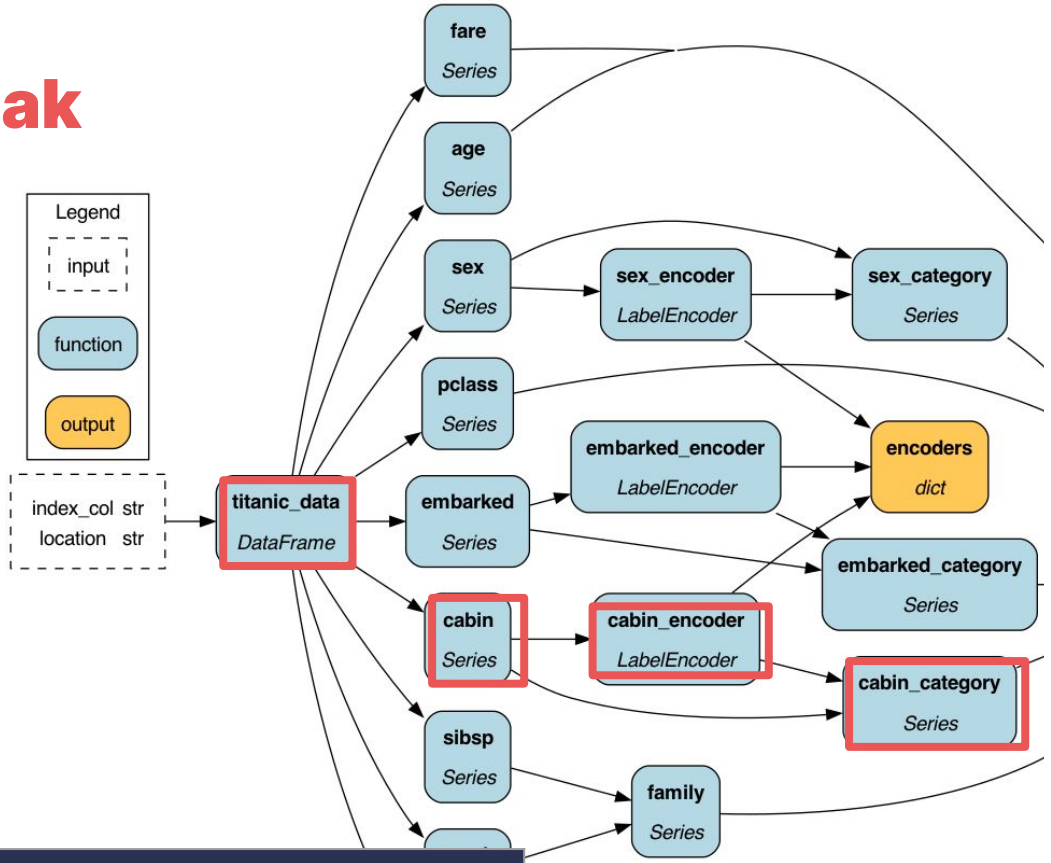
- Type annotations match
- You have the right inputs for the outputs you want
- Can add runtime data quality checks via `@check_output`
  - e.g. with Pandera

Also, straightforward to add unit tests & integration tests.

# Hard to break

Can easily integration test a distinct path

E.g. add to your CI system.




```
dr = driver.Driver(config, modules…,)
result = dr.execute(
    ["cabin_category"],inputs=...)
```

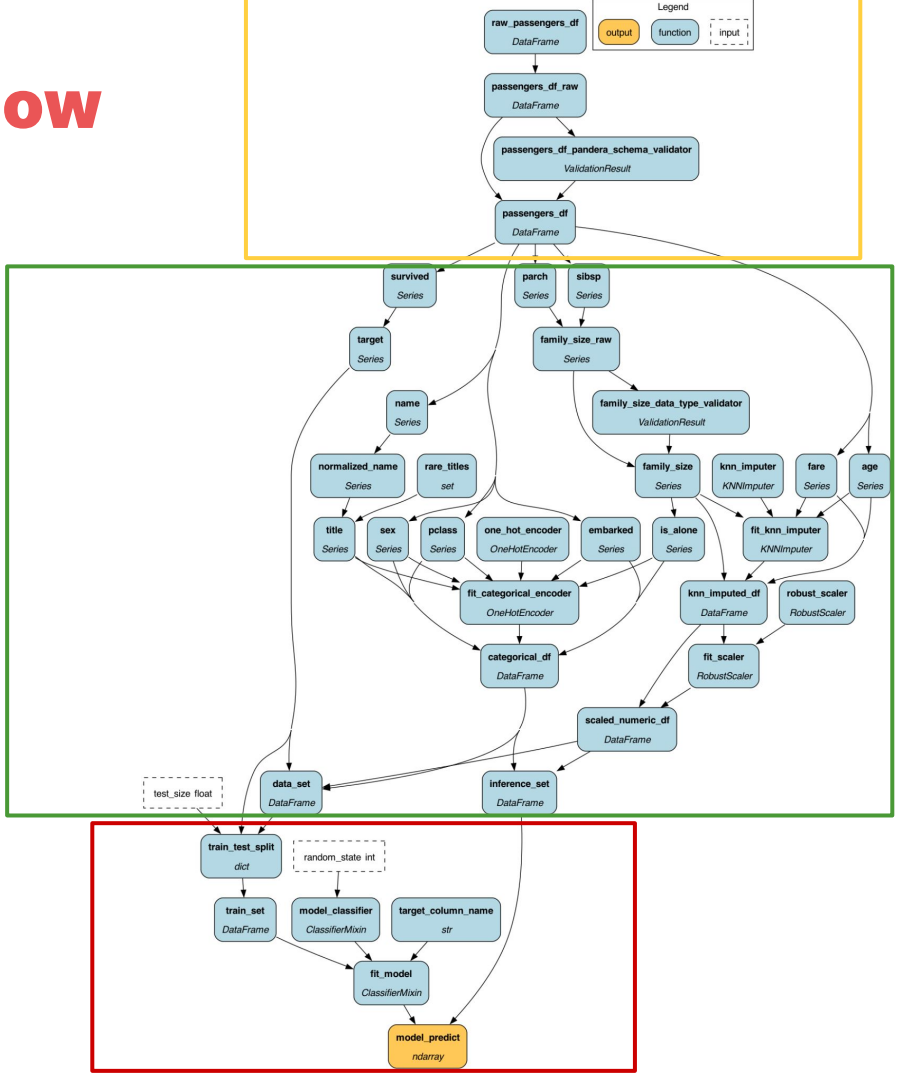
# SOLID Principles:
# Liskov Substitution Principle

## easy to swap parts of the flow
## without altering “correctness”

# Easy to swap parts of your flow

E.g. dev concerns vs prod concerns..

```
import data_loader, feature_transforms, model_pipeline

# 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}
  )
```




## Options to swap:

- swap where this code runs
- module swap
- @config.when

# Easy to swap parts of your dataflow

```python
def embedding(query: str) -> List[float]:
  response = openai.Embedding.create(input=query, model="HARDCODED")
  return response["data"][0]["embedding"]


def nn_ids(
 embedding: List[float], vectordb_client: Client, top_k: int) -> List[int]:
  results = vectordb_client.search(embedding=embedding, top_k=top_k)
  return results
```

# Easy to swap parts of your dataflow

## Can use @config to modify; swapping functions is straightforward.

```python
@config.when(provider="openai")
def embedding__openai(query: str) -> List[float]:
  response = openai.Embedding.create(input=query, model="HARDCODED")
  return response["data"][0]["embedding"]


@config.when(provider="anthropic")
def embedding__anthropic(query: str, anthropic_param: str) -> List[float]:
  response = antropic_api.get_embedding(input=query, param=anthropic_param)
  return response["data"]["embedding"]


def nn_ids(
 embedding: List[float], vectordb_client: Client, top_k: int) -> List[int]:
  results = vectordb_client.search(embedding=embedding, top_k=top_k)
  return results
```

# SOLID Principles:
# Interface Segregation Principle

**I should only depend on what I need;**
**→ clearer code/execution**

# Depend/Run only what you need

```python
# client_features.py

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

1. Functions only depend on what they need.

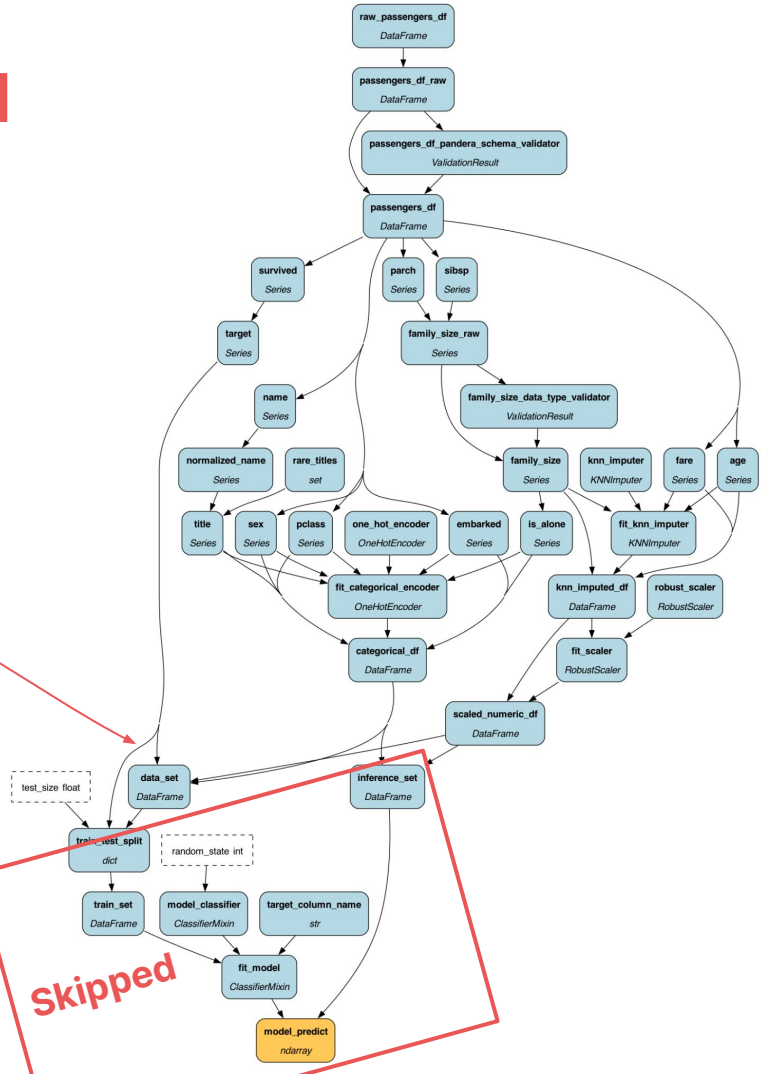
# Depend/Run only what you need

```
# client_features.py

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


def some_other_function(height_zero_mean_unit_variance: pd.Series,
                        foo_bar_input: float) -> pd.Series:
    return … # your logic here
```

1. Functions only depend on what they need.
2. Don’t need it? Don’t run it.

# Depend/Run only what you need

```
import data_loader, feature_transforms, model_pipeline

# DAG for training/inferring on titanic data
titanic_dag = driver.Driver(config,
  data_loader, feature_transforms, model_pipeline,
  adapter=base.DefaultAdapter(),
)
# execute & get output just data_set
result = titanic_dag.execute(["data_set"],
      inputs={"raw_passengers_df": raw_passengers_df}
)
```




1. Functions only depend on what they need.
2. Don’t need it? Don’t run it.

# SOLID Principles:
# Dependency Inversion Principle

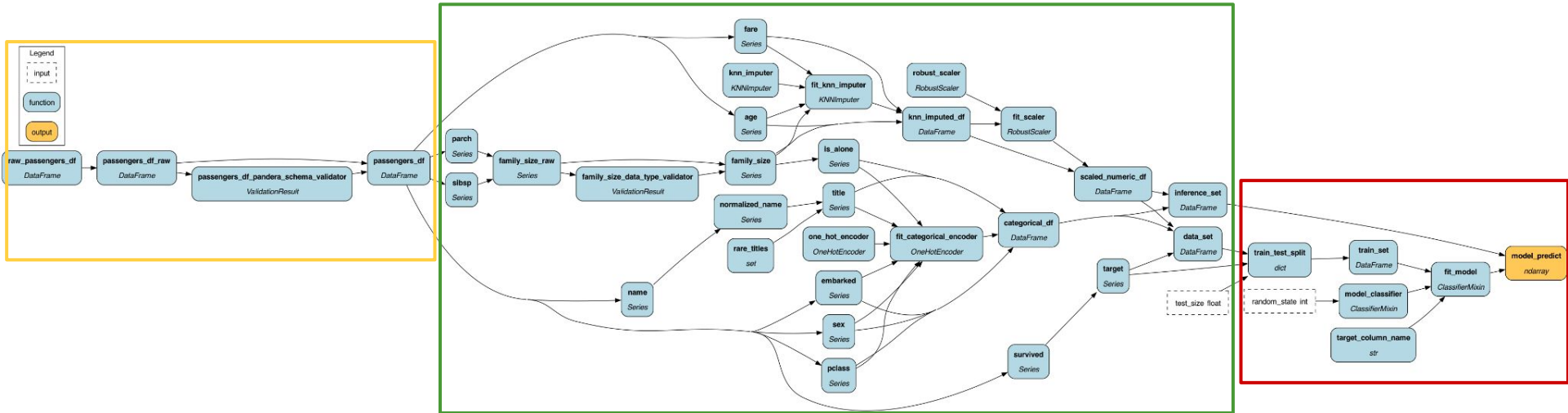
**avoid tight coupling between different parts of your code via abstractions;**
**→ ensure flexibility for change & refactor**

# Avoid tight coupling of code

**Hamilton does this by definition.**

Abstraction:

- Name of functions & parameters + type annotations.
- Functions & modules as independent iterable units.



# Hamilton

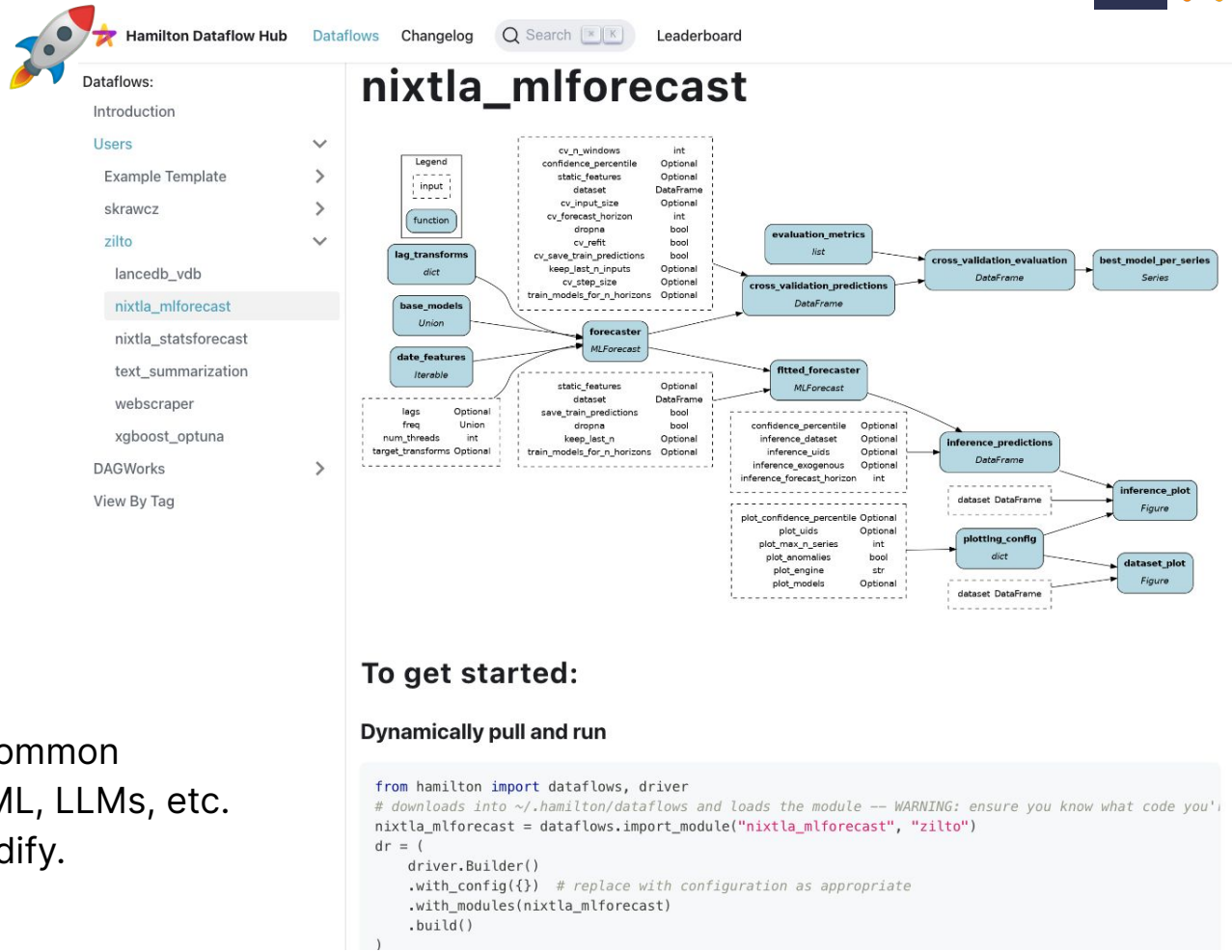
Natively bringing SWE best practice to Python data transformations

1. ~~Motivating pain~~
2. ~~Hamilton~~
3. ~~General Usage~~
4. ~~Native SWE~~
5. **Summary**

# TL;DR: Summary - Hamilton + Native SWE

1. Hamilton is a micro-framework for expressing dataflows in Python.
    - Write code that people aren't terrified of inheriting!
2. The Hamilton paradigm: ⬆ SWE Best Practices ⬆ value of your work
    - Naturally testable & documentation friendly code.
    - Naturally reusable and modular code so you can move faster.
    - It's hard to do bad things when adding/removing/adjusting dataflows.
    - Standardized way to iterate and add to a code base.
3. We can design ways that guide users towards good SWE practices.
    - Hard to beat the modern python function with type annotations as an interface!

# Soft launch sharing dataflows:




[hub.dagworks.io](hub.dagworks.io)

Goal: build a repository of common transformations, e.g. data, ML, LLMs, etc. that you can then easily modify.

# What I'm building on top of Hamilton

**With a one-line code change you get:**

- **Versioning** (code)
- **Lineage** (code & artifacts)
- **Catalog** (code & artifacts)
- **Observability** (code & data)


**Sign up for free @ www.dagworks.io**

# Fin. Thanks for listening!

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

**Questions**?

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

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

📚 Documentation: hamilton.dagworks.io

🎓 Self-paced tutorial https://github.com/DAGWorks-Inc/hamilton-tutorials/tree/main/2023-10-09

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

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

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

# Native SWE

5 Common Ideals

- ~~Testing & Documentation~~
- **KISS**
- **YAGNI**
- DRY
- SOLID

# KISS (keep it simple, stupid)

# KISS (keep it simple, stupid)

```
data['hzmuv'] = data['height_zero_mean'] / height_std_dev
```

**vs**

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

**No object-oriented code**: don't need to learn much to write a function.

**Testing story:** can change with confidence.

**Complexity is contained**: function, including naming, defines the boundaries

# What we're comparing against:

```
df = loader.load_actuals(dates) # e.g. spend, signups
if config['region'] == 'UK':
   df['holidays'] = is_uk_holiday(df['year'], df[' week'])
else:
   df['holidays'] = is_holiday(df['year'], df['week'])
df['avg_3wk_spend'] = df['spend'].rolling(3).mean()
df['acquisition_cost'] = df['spend'] / df['signups']
df['spend_shift_3weeks'] = df['spend'].shift(3)
df['special_feature1'] = compute_bespoke_feature(df)
df['spend_b'] = multiply_columns(df['acquisition_cost'], df['B'])
save_df(df, "some_location")
```

# YAGNI (You Aren’t Gonna Need It)

*“Premature optimization is the root of all evil” - Donald Knuth*

# YAGNI (You Aren't Gonna Need It)

```python
def height_zero_mean_unit_variance(height_zero_mean: pd.Series,
                                   height_std_dev: float) -> pd.Series:
    """Zero mean unit variance value of height"""
    return height_zero_mean / height_std_dev
```

**Hard to over engineer**: functions force simplicity.

**Declarative structure:** easy to modify when needed.

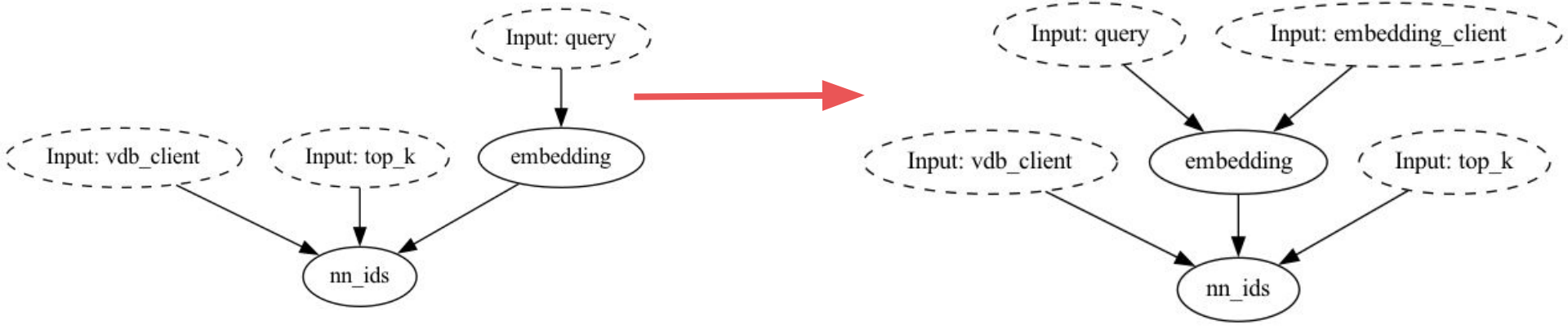
# YAGNI (You Aren't Gonna Need It)

## E.g. easy to refactor when needed:

```python
def embedding(query: str) -> List[float]:
  response = openai.Embedding.create(input=query, model="HARDCODED")
  return response["data"][0]["embedding"]
```



```python
def embedding(query: str, embedding_client: object) -> List[float]:
  return embedding_client.get_embedding(query)
```



# What we're comparing against:

```
df = loader.load_actuals(dates) # e.g. spend, signups
if config['region'] == 'UK':
   df['holidays'] = is_uk_holiday(df['year'], df[' week'])
else:
   df['holidays'] = is_holiday(df['year'], df['week'])
df['avg_3wk_spend'] = df['spend'].rolling(3).mean()
df['acquisition_cost'] = df['spend'] / df['signups']
df['spend_shift_3weeks'] = df['spend'].shift(3)
df['special_feature1'] = compute_bespoke_feature(df)
df['spend_b'] = multiply_columns(df['acquisition_cost'], df['B'])
save_df(df, "some_location")
```

# Native SWE

5 Common Ideals

- ~~Testing & Documentation~~
- ~~KISS~~
- ~~YAGNI~~
- **DRY**
- **SOLID**

# DRY (don’t repeat yourself)

# DRY (don't repeat yourself)

```
data['avg_3wk_spend'] = data['spend'].rolling(3).mean()
data['spend_per_signup'] = data['spend']/data['signups']
spend_mean = data['spend'].mean()
data['spend_zero_mean'] = data['spend'] - spend_mean
spend_std_dev = data['spend'].std()
data['szmuv'] = data['spend_zero_mean']/spend_std_dev
```

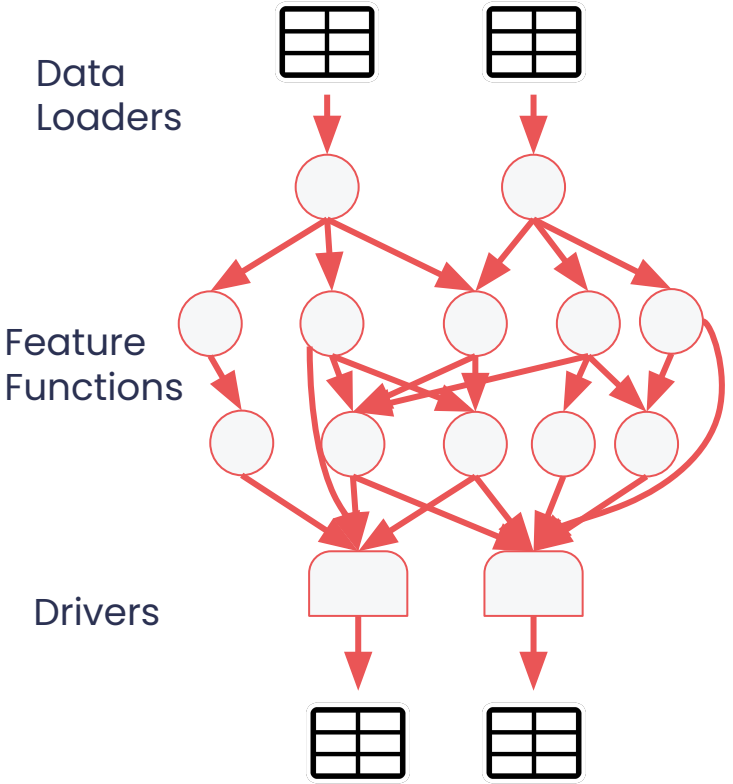
VS

```
def spend_zero_mean(spend: pd.Series, spend_mean: float) -> pd.Series:
  """Shows function that takes a scalar. In this case to zero mean spend."""
  return spend - spend_mean

def spend_std_dev(spend: pd.Series) -> float:
  """Function that computes the standard deviation of the spend column."""
  return spend.std()

def spend_zero_mean_unit_variance(spend_zero_mean: pd.Series, spend_std_dev: float)
-> pd.Series:
  """Function showing one way to make spend have zero mean and unit variance."""
  return spend_zero_mean / spend_std_dev
```

# Example Hamilton use case: Feature Engineering

Code that needs to be written:

1. Functions to load data
    a. normalize/create common index to join on
2. Feature functions
    a. Unit test these easily!
    b. Optional: model functions.
3. Drivers materialize data
    a. DAG is walked for only what's needed.
    b. E.g. place this code in wherever you run your python.



# When should I not consider Hamilton?

You **can't draw** a flowchart (DAG)...

Or if you have code that depends on inspecting the value output of a transform, e.g.

```
output_1 = transform_1(a, b)
if output_1 < 0.5:
  output_2 = transform_2(output_1)
else:
  output_2 = transform_3(output_1
```

If it's minor, you can break this up into separate DAGs ... otherwise not a fit.

*[though we can build this capability in...]*

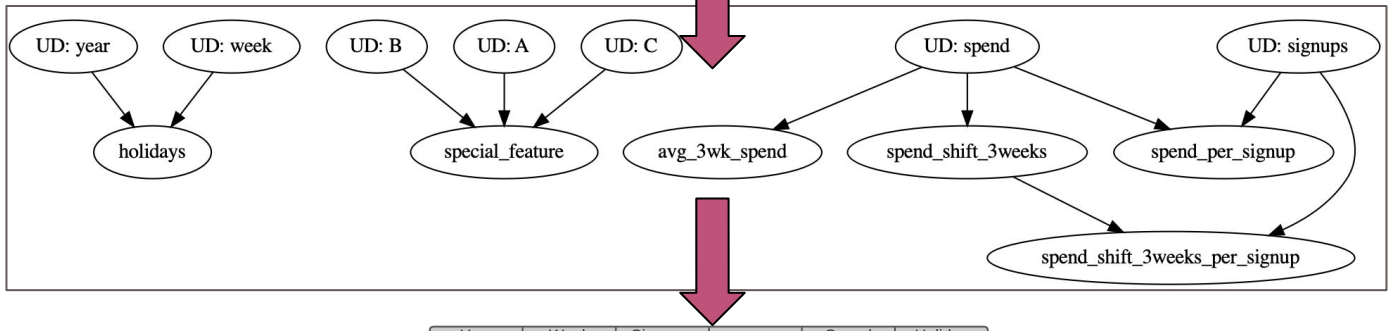
# Example Hamilton use case: Feature Engineering

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:




run.py

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 |