Move Fast and Don't Break Things

How to build a data platform that scales with your organization

DAGWORKS (YCW23) Elijah ben Izzy CTO & Co-Founder

Data Council Austin – 2024

Some Background Context

DAGWORKS

whoami

Elijah ben Izzy
Co-creator of **Hamilton**; CTO
DAGWorks Inc. (YCW23)

10+ years in ML & Data platforms

Standardizing Data, ML, and LLM pipelines

Open Core!

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

TL;DR

I want to convince you that...

1. There is a trade-off between building quickly and reliably
2. Having a good platform can make it less of a trade-off
3. Hamilton (OS) is a good conduit to do so
4. Hamilton can help you quickly move from dev → prod → dev

The Agenda

A mental model for trade-offs
Platforms to the rescue
Hamilton: a lightweight platform abstraction
Some applications
- ↳ ML Pipelines
- ↳ LLM/RAG

Zooming out

The Agenda

A mental model for trade-offs
Platforms to the rescue
Hamilton: a lightweight platform abstraction
Some applications
- ↳ **ML Pipelines**
- ↳ **LLM/RAG**

Zooming out

DAGWORKS

What describes your ML/AI code?

A dilemma…

💬 **!!!** I must quickly deliver above all else **!!!**

…

💻 + 60,000 LOC

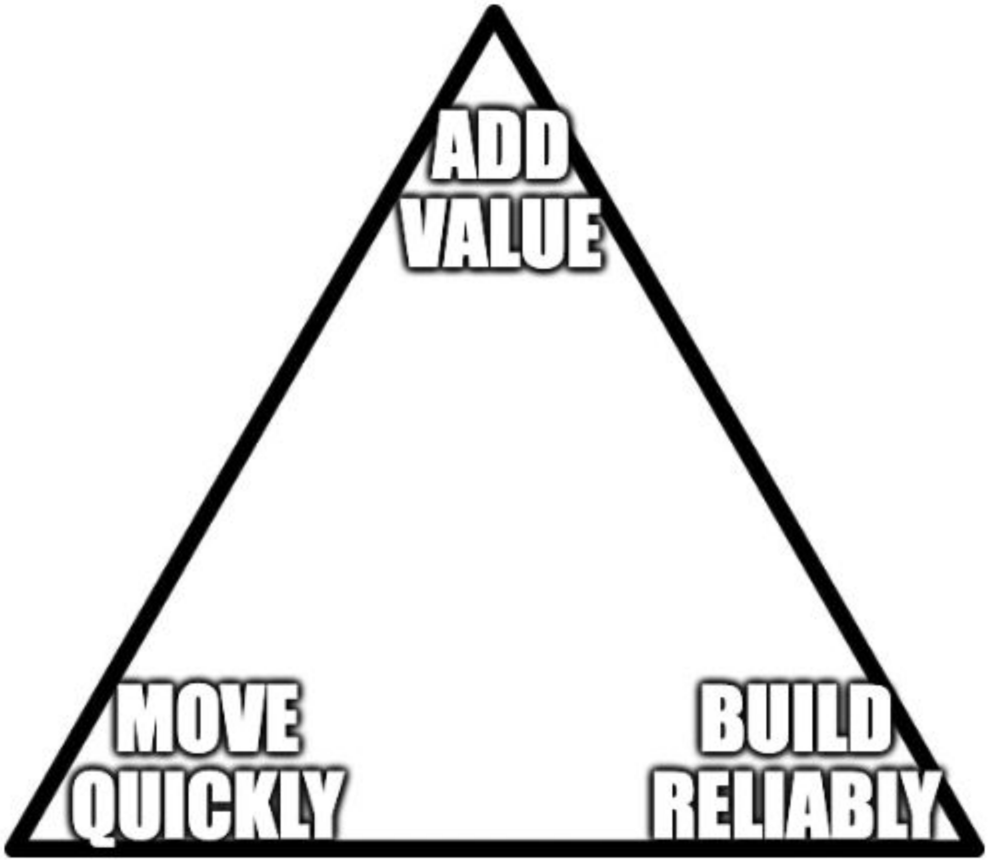
💻 + 8 prod outages

💻 + 30tb of unstructured data

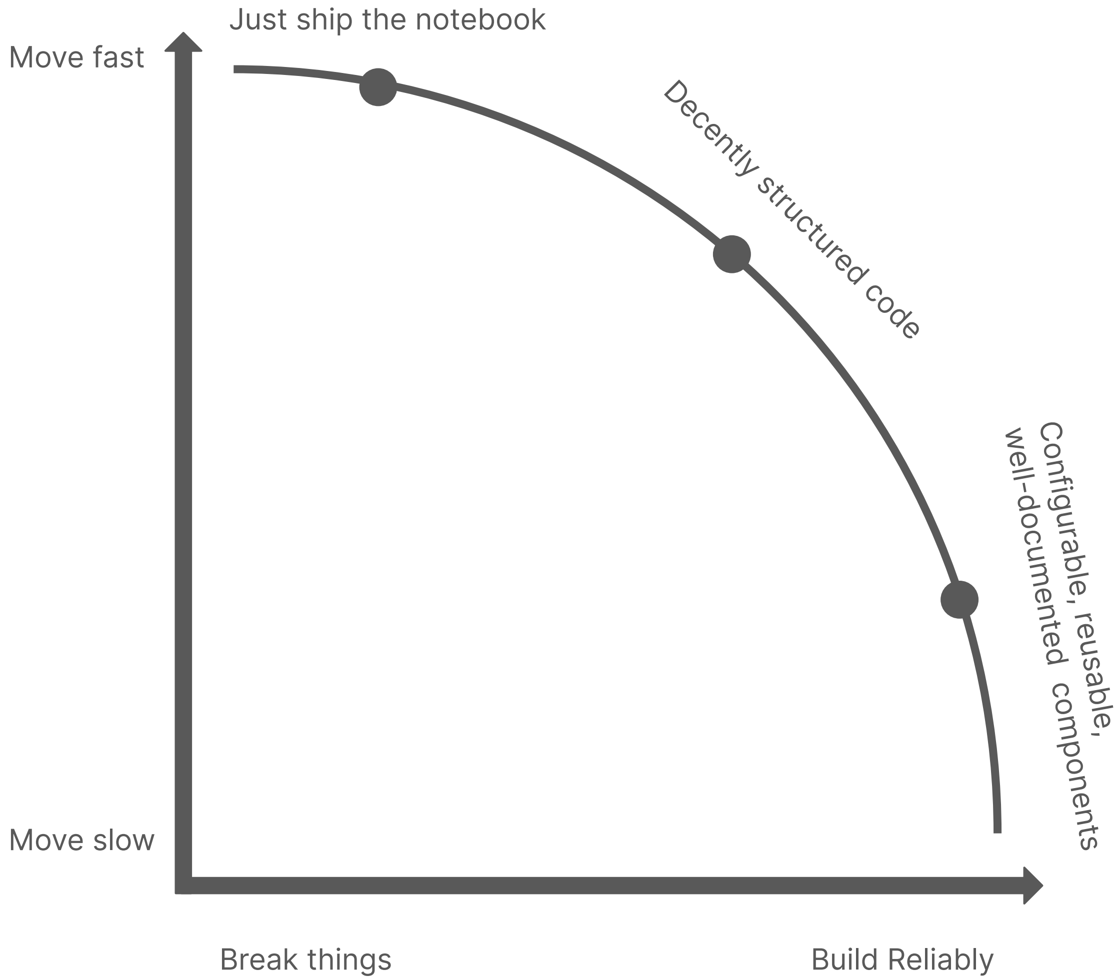
💻 + $,$$$,$$$ snowflake bill

…

💬 **!!!** Too much tech debt to do my job! **!!!**


ADD VALUE
MOVE QUICKLY
BUILD RELIABLY


MOVE QUICKLY
BUILD RELIABLY


Just ship the notebook
Move fast
Decently structured code
Configurable, reusable,
well-documented components
Move slow
Break things
Build Reliably

The Agenda

A mental model for trade-offs
Platforms to the rescue
Hamilton: a lightweight platform abstraction
Some applications
- ↳ ML Pipelines
- ↳ LLM/RAG

Zooming out

Strategic goals of a platform

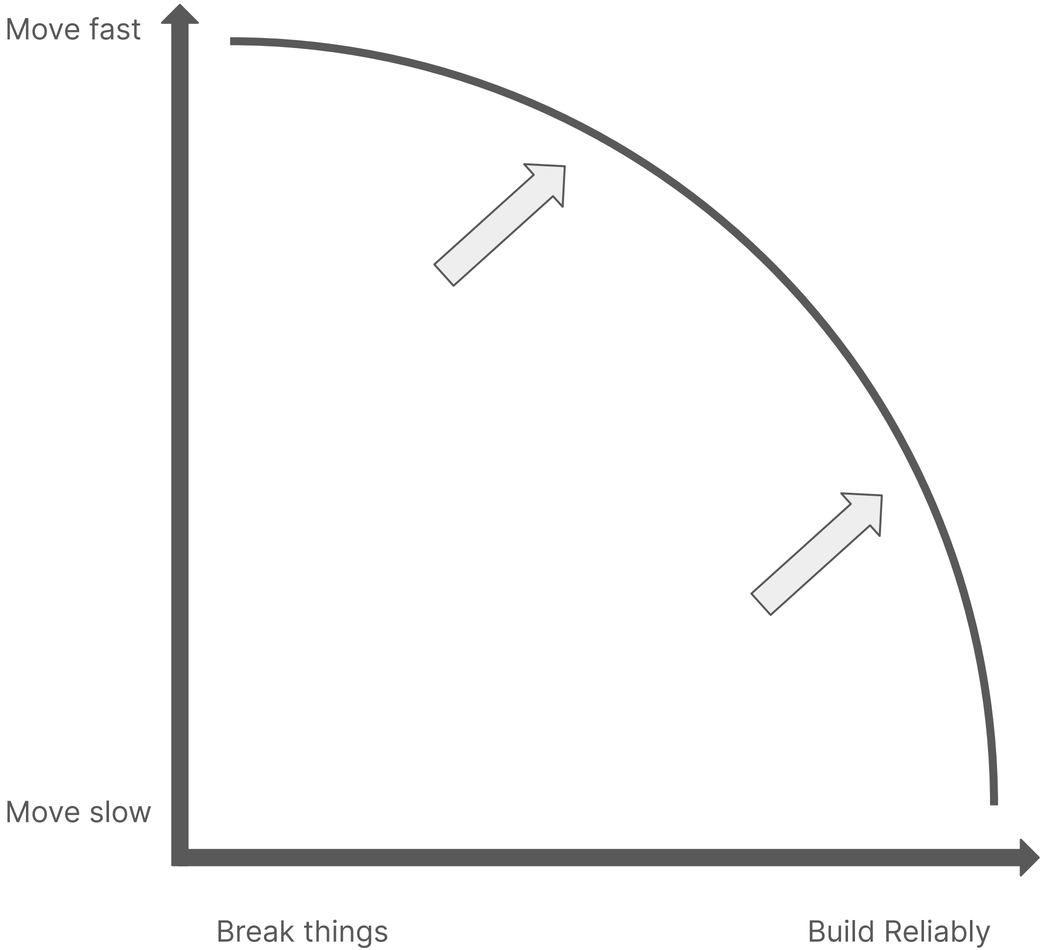
Three-fold:

1. Make it easy to operate *on* the curve
2. Change the shape of the curve
3. Make it easy to move along the curve (dev ⇔ prod)

Strategic goals of a platform

Three-fold:

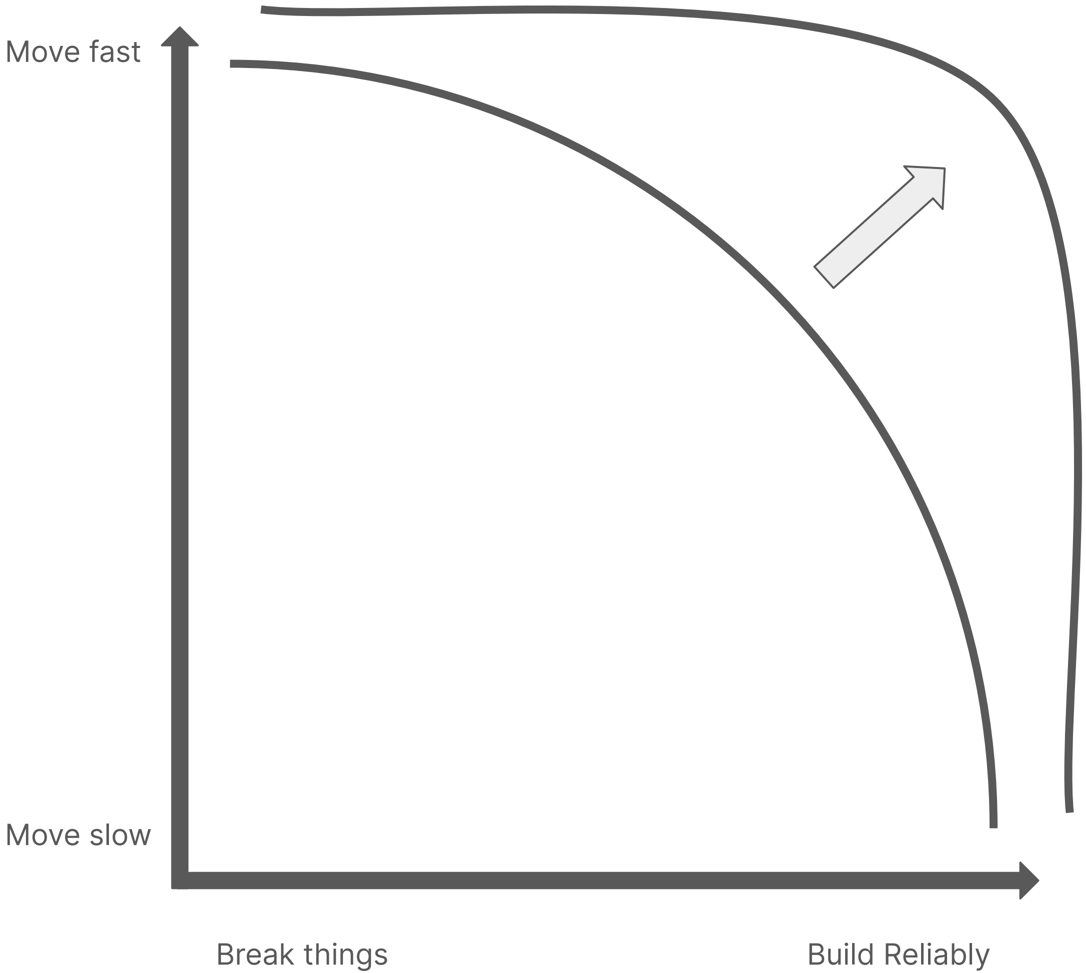
1. Make it easy to operate *on* the curve
2. Change the shape of the curve
3. Make it easy to move along the curve (dev ⇔ prod)


Move fast
Move slow
Break things
Build Reliably

Strategic goals of a platform

Three-fold:

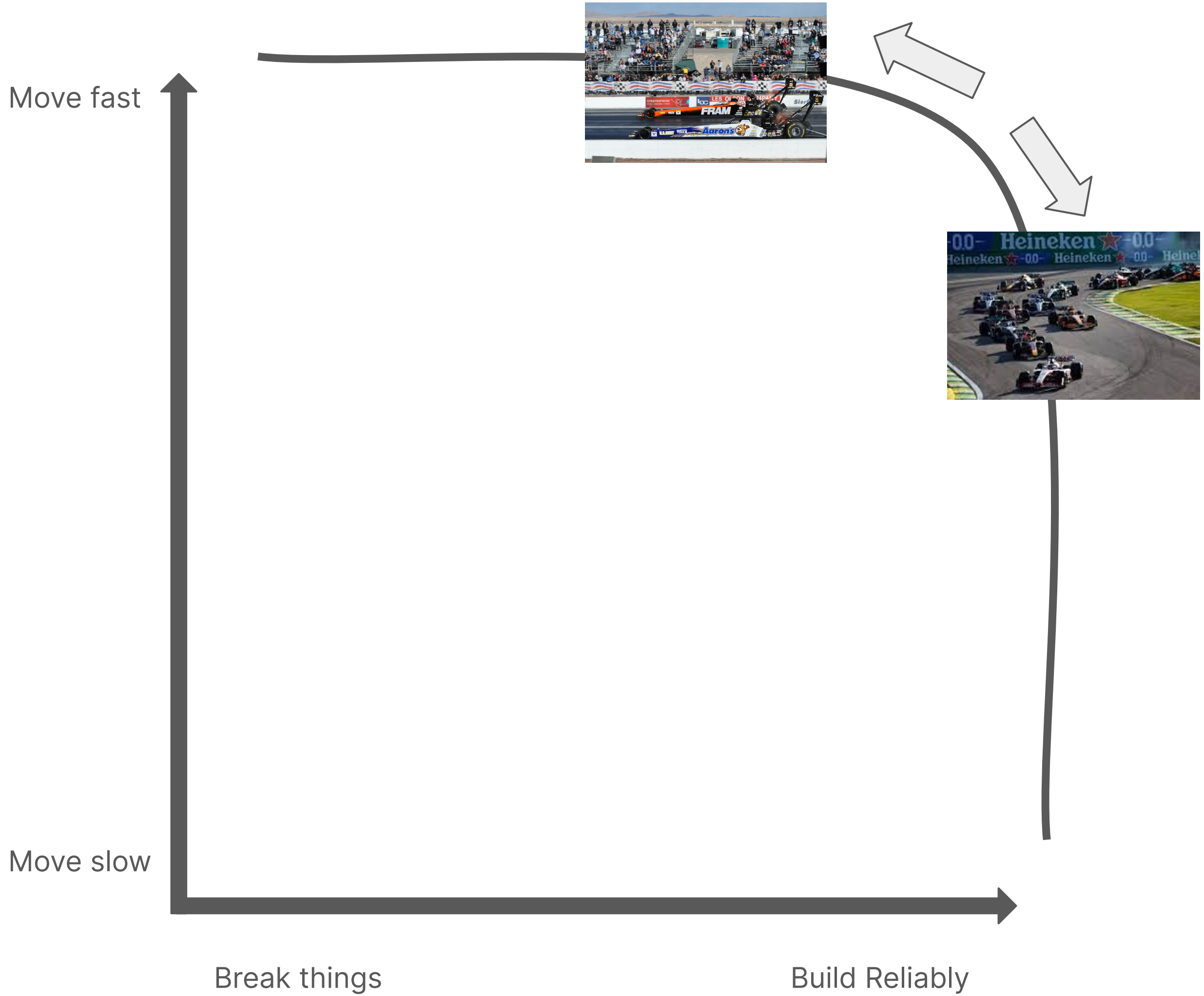
1. Make it easy to operate *on* the curve
2. Change the shape of the curve
3. Make it easy to move along the curve (dev ⇔ prod)


Move fast
Move slow
Break things
Build Reliably

Strategic goals of a platform

Three-fold:

1. Make it easy to operate *on* the curve
2. Change the shape of the curve
3. Make it easy to move along the curve (dev ⇔ prod)


Move fast
Move slow
Break things
Build Reliably
FRAM
Heineken

The Agenda

A mental model for trade-offs
Platforms to the rescue
Hamilton: a lightweight platform abstraction
Some applications
- ↳ ML Pipelines
- ↳ LLM/RAG

Zooming out

DAGWORKS

All open source!

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

Get started in <15 minutes!

Documentation

https://hamilton.dagworks.io/

Try it out

https://www.tryhamilton.dev/

Try Hamilton Get Started Reference Documentation DAGWorks Inc. GitHub

https://www.tryhamilton.dev

Hamilton

Wrangle Pandas codebases into shape.

Learn (5 mins) Github 890+

- [x] Write always unit testable code
- [x] Add runtime data validation easily
- [x] Produce readable and maintainable code
- [x] Visualize lineage (click the run button to see)
- [x] Run anywhere python runs: in airflow, jupyter, fastapi, etc...
- [x] Skip the CS degree to use it

Try Hamilton right here in your browser 👇

```python
# 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!

DAGWORKS

STITCH FIX

BRITISH CYCLING

Joby AVIATION

TRANSFIX

IBM®

ascena RETAIL GROUP INC.

AXIOM CLOUD

amazon web services

coinbase

naro™

Equipment Share

veriff

HABITAT ENERGY

Government Digital Service

Pacific Northwest NATIONAL LABORATORY

LexisNexis® RISK SOLUTIONS

Opendoor

TRADEWELL TECHNOLOGIES

Hamilton: the “a-ha” Moment

Idea What if every asset corresponded to **exactly one** python fn?

And… what if the way that function was written tells you everything you needed to know?

In Hamilton, the artifact (asset) is determined by the ***name of the function****.*
The dependencies are determined by ***the parameters****.*

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
```

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

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

```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
```

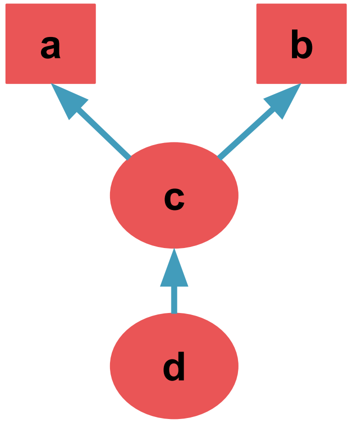
Driver says what/when to execute

```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)
```

DAGWORKS

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)
```

DAGWORKS

Hamilton: extensions

Q: Doesn't Hamilton make your code more verbose?

A: Yes, but that's not always a bad thing. When it is, we have decorators!

- ❑ `@tag # attach metadata`
- ❑ `@parameterize # curry + repeat a function`
- ❑ `@extract_columns # one dataframe -> multiple series`
- ❑ `@check_output # data validation`
- ❑ `@config.when # conditional transforms`
- ❑ `@subdag # recursively utilize groups of nodes`
- ❑ *`@... # new ones all the time`*

Move (quickly) along the curve

Testing – Everything ⊆ python functions ⇒ easy testing ⇒ faster dev

Scaling – driver handles *where* to run, port subcomponents as needed

- **Parallelism** – simple constructs to abstract away parallelism
- **Delegation** – run on any executor with hamilton constructs

Caching – single line *extension* enables fingerprinting

Data quality – everything is decoupled – code is stabler + more flexible

Integrations – customize on your own infrastructure

Testing – Everything ⊆ python functions ⇒ easy testing ⇒ faster dev.

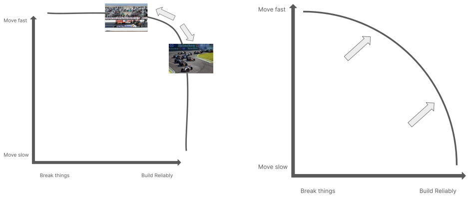
Scaling – driver handles *where* to run, port subcomponents as needed

- **Parallelism** – simple constructs to abstract away parallelism
- **Delegation** – run on any executor with hamilton constructs

Caching – single line *extension* enables fingerprinting

Data quality – everything is decoupled – code is stabler + more flexible

Integrations – customize execution



DAGWORKS

Unit testing – functions ⇒ easy testing

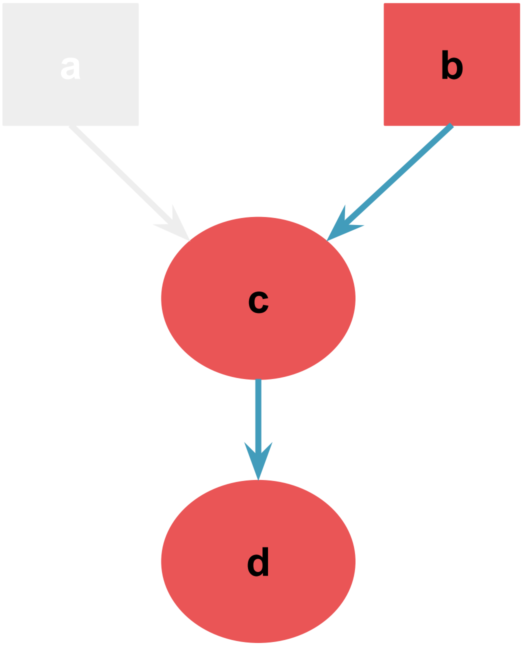
```python
# client_features.py

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

```python
# 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
```

DAGWORKS

Integration Testing

- Run one portion of DAG
- Inject sample data
- Or run on prod data (yolo)



DAGWORKS

Testing – functions are easy to unit tests. Paths are easy to integrate.

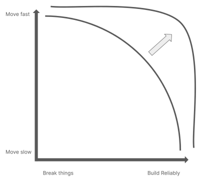
Scaling – driver handles *where* to run, port subcomponents as needed

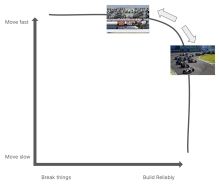
- **Parallelism** – simple constructs to abstract away parallelism
- **Delegation** – run on any executor with hamilton constructs

Caching – single line *extension* enables fingerprinting

Data quality – everything is decoupled – code is stabler + more flexible

Integrations – customize execution





DAGWORKS

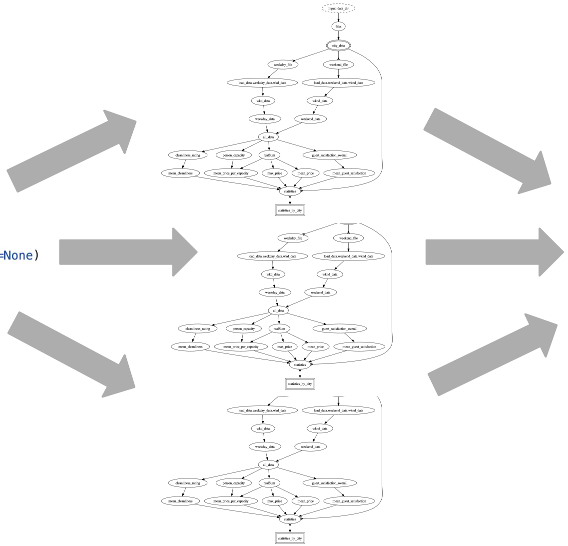
Scaling: Parallelism/Map-Reduce

Map: Declare fn output as **`Parallelizable[...]`**

Reduce: Declare fn input as **`Collect[...]`**

Delegate to custom/built-in executor

```python
def city_data(files: List[str]) -> Parallelizable[CityData]:
    """Gathers a list of per-city data for processing/analyzing"""

    cities = dict()
    for file_name in files:
        city = os.path.basename(file_name).split("_")[0]
        is_weekend = file_name.endswith("weekends.csv")
        if city not in cities:
            cities[city] = CityData(city=city, weekend_file=None, weekday_file=None)
        if is_weekend:
            cities[city].weekend_file = file_name
        else:
            cities[city].weekday_file = file_name
    for city in cities.values():
        yield city
```



```python
def statistics_by_city(
  statistics: Collect[dict]
) -> pd.DataFrame:
    """Joins all data together"""
    return pd.DataFrame.from_records(statistics).set_index("city")
```

Testing – functions are easy to unit tests. Paths are easy to integrate.

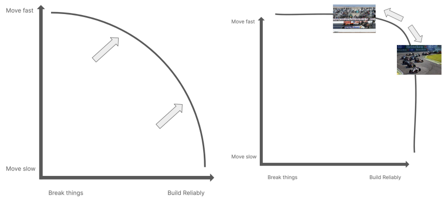
Scaling – driver handles *where* to run, port subcomponents as needed

- **Parallelism** – simple constructs to abstract away parallelism
- **Delegation** – run on any executor with hamilton constructs

Caching – single line *extension* enables fingerprinting

Data quality – everything is decoupled – code is stabler + more flexible

Integrations – customize on your own infrastructure



DAGWORKS

Caching

Fingerprinting (cache inputs, data, code) with one-line change

- dev speedup (local)
- remote/prod – come chat

```
dr = (
    driver.Builder()
    .with_modules(functions)
    .with_adapters(h_diskcache.DiskCacheAdapter())
    .build()
)
```

Testing – functions are easy to unit tests. Paths are easy to integrate.

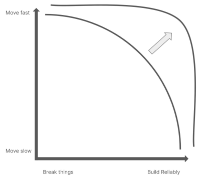
Scaling – driver handles *where* to run, port subcomponents as needed

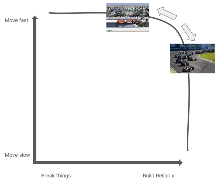
- **Parallelism** – simple constructs to abstract away parallelism
- **Delegation** – run on any executor with hamilton constructs

Caching – single line *extension* enables fingerprinting

Data quality – everything is decoupled – code is stabler + more flexible

Integrations – customize execution





DAGWORKS

Data quality – functions ⇒ data assets ⇒ data quality

```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 – functions are easy to unit tests. Paths are easy to integrate.

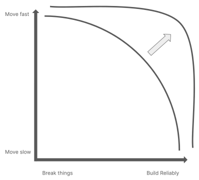
Scaling – driver handles *where* to run, port subcomponents as needed

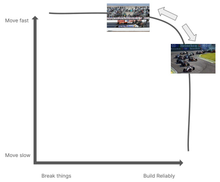
- **Parallelism** – simple constructs to abstract away parallelism
- **Delegation** – run on any executor with hamilton constructs

Caching – single line *extension* enables fingerprinting

Data quality – everything is decoupled – code is stabler + more flexible

Integrations – customize execution





DAGWORKS

Integrations – lifecycle methods

```python
class PDBDebugger(NodeExecutionHook, NodeExecutionMethod):
    def run_to_execute_node(
            self,
            *,
            node_callable: Callable,
            node_kwargs: Dict[str, Any],
            **future_kwargs: Any) -> Any:
        """This is run to execute the node callable.

      All this does is run the node_callable with the node_kwargs in a PDB debugger.

        :param node_callable: Function that the node runs
        :param node_kwargs:  Keyword arguments passed to the node
        :param future_kwargs:  Reserved for future backwards compatibility.
        :return: The result of running the node
        """
        return pdb.runcall(node_callable, **node_kwargs)
```

```python
dr = (
    Builder()
    .with_modules(..)
    .with_adapters(PDBDebugger())
    .build()
)
```

DAGWORKS

Integrations – materializers

```python
@dataclasses.dataclass
class MLFLowSaver(DataSaver):
    """Our MLFlow Materializer"""
    experiment_name: str
    run_name: str
    artifact_path: str
    model_type: str = "sklearn"

    @classmethod
    def applicable_types(cls) -> Collection[Type]:
        return [dict]

    @classmethod
    def name(cls) -> str:
        return "mlflow"

    def save_data(self, data: Any) -> Dict[str, Any]:
        # Initiate the MLflow run context
        with mlflow.start_run(run_name=self.run_name) as run:
           # Log the parameters used for the model fit
           mlflow.log_params(data["params"])
           # Log the error metrics that were calculated
           mlflow.log_metrics(data["metrics"])

           # Log an instance of the trained model for later use
           ml_logger = getattr(mlflow, self.model_type)
           model_info = ml_logger.log_model(
               sk_model=data["trained_model"],
               input_example=data["input_example"],
               artifact_path=self.artifact_path
           )
        return model_info # return some metadata
```

```python
dr = (
  # declare the driver and what the pipeline is to be built from
  driver.Builder()
    .with_modules(data_loading, featurization, model_pipeline)
    .build()
)

materializer_results, results = dr.materialize(
    to.sklearn_mlflow(
      id="mlflow_sink",
      dependencies=["trained_model_and_metrics"],
      experiment_name="exp_name",
      run_name="run_foo",
      artifact_path="save/to/path"
    ),
    inputs={"location": ..., "target": ..., ...}
)
```

DAGWORKS

The Agenda

A mental model for trade-offs
Platforms to the rescue
Hamilton: a lightweight platform abstraction
Some applications
- ↳ ML Pipelines
- ↳ LLM/RAG

Zooming out

The Agenda

A mental model for trade-offs
Platforms to the rescue
Hamilton: a lightweight platform abstraction
Some applications
- ↳ ML Pipelines
- ↳ LLM/RAG

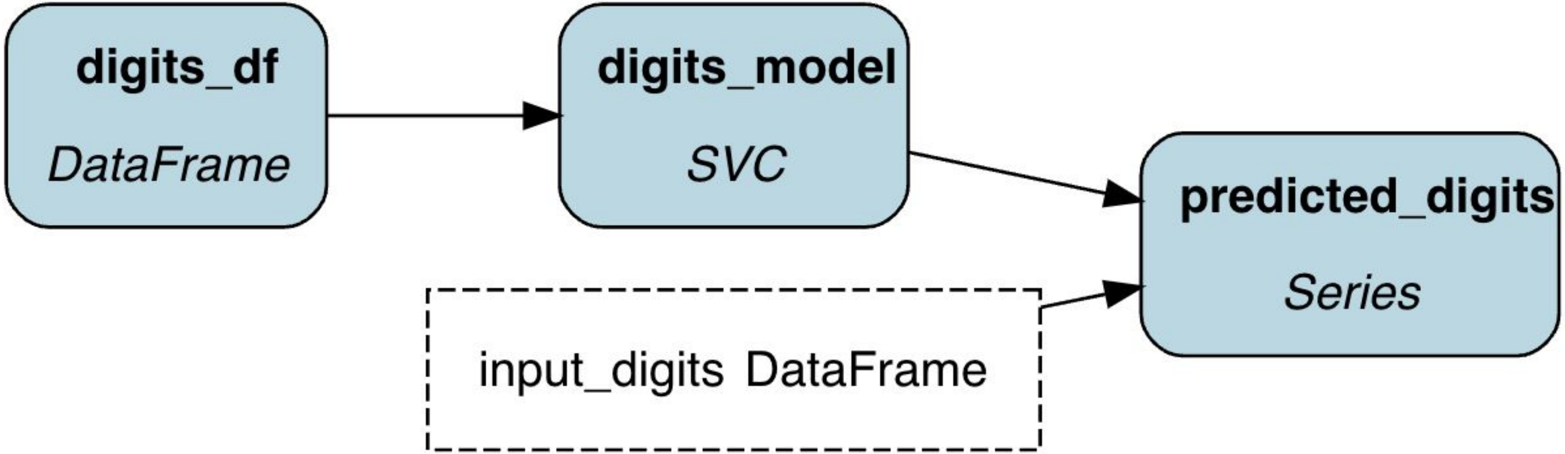
Zooming out

Hamilton: ML pipelines

```python
# simple_pipeline.py
from sklearn import datasets
from sklearn import svm

import pandas as pd

def digits_df() -> pd.DataFrame:
    """Load the digits dataset."""
    digits = datasets.load_digits()
    _digits_df = pd.DataFrame(digits.data)
    _digits_df["target"] = digits.target
    return _digits_df

def digits_model(digits_df: pd.DataFrame) -> svm.SVC:
    """Train a model on the digits dataset."""
    clf = svm.SVC(gamma=0.001, C=100.)
    _digits_model = clf.fit(
          digits_df.drop('target', axis=1),
          digits_df.target
    )
    return _digits_model

def predicted_digits(
      digits_model: svm.SVC,
      input_digits: pd.DataFrame) -> pd.Series:
    """Predict the digits."""
    return pd.Series(digits_model.predict(input_digits))
```

Hamilton: ML pipelines

```python
# simple_pipeline.py
from sklearn import datasets
from sklearn import svm

import pandas as pd
```




```python
    """Predict the digits."""
    return pd.Series(digits_model.predict(input_digits))
```

Hamilton: ML pipelines

MLEs/DS:

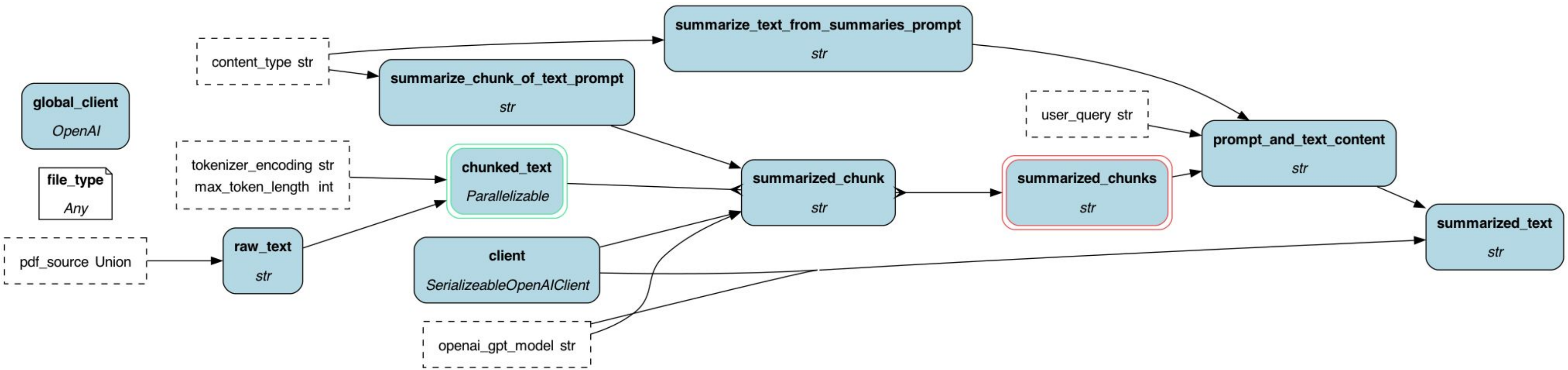
- Write ML pipelines as Hamilton steps
- Easily add data quality
- Scale up by delegating to spark/ray/dask
- Separate code structure from task structure
  - Test locally on sample data
  - Run remotely on full data
- Integrate model registry, other providers

The Agenda

A mental model for trade-offs
Platforms to the rescue
Hamilton: a lightweight platform abstraction
Some applications
- ↳ ML Pipelines
- ↳ LLM/RAG

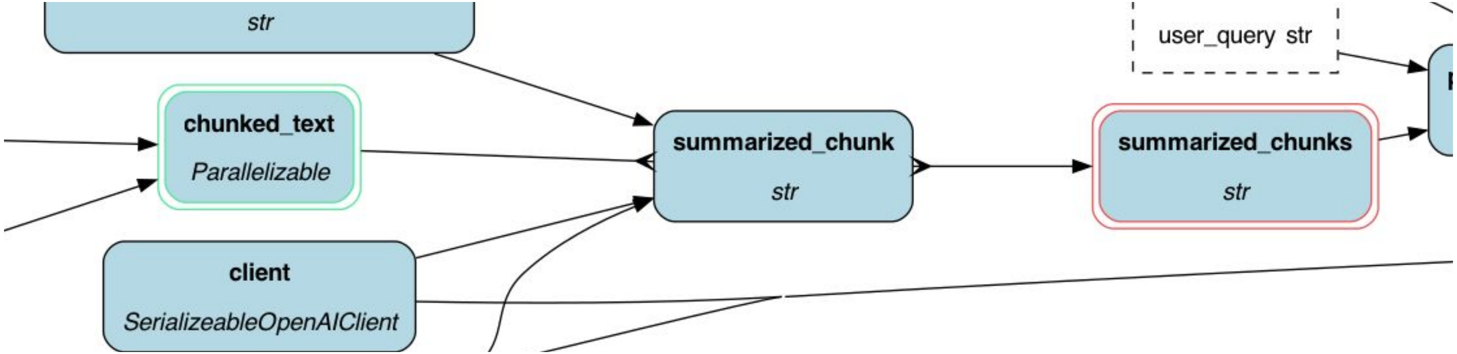
Zooming out

Hamilton: LLM/RAG pipelines

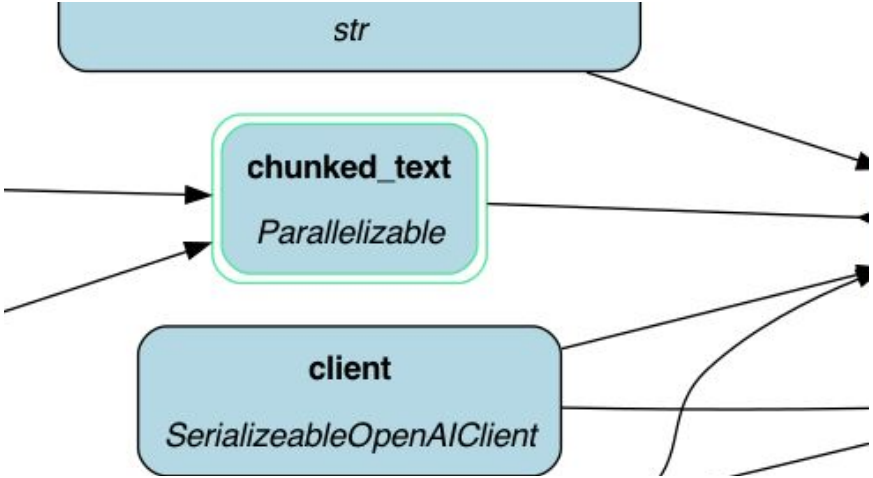


DAGWORKS

Hamilton: LLM/RAG pipelines

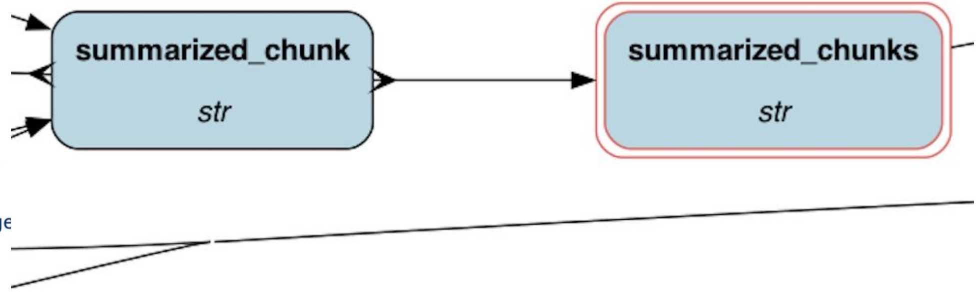


```python
def chunked_text(
    raw_text: str, tokenizer_encoding: str = "cl100k_base", max_token_length: int = 1500
) -> Parallelizable[str]:
    """Chunks the pdf text into smaller chunks of size max_token_length.
    :param raw_text: the Series of individual pdf texts to chunk.
    :param max_token_length: the maximum length of tokens in each chunk.
    :param tokenizer_encoding: the encoding to use for the tokenizer.
    :return: Series of chunked pdf text. Each element is a list of chunks.
    """
    tokenizer = tiktoken.get_encoding(tokenizer_encoding)
    _encoded_chunks = _create_chunks(raw_text, max_token_length, tokenizer)
    # _decoded_chunks = [tokenizer.decode(chunk) for chunk in _encoded_chunks]
    # return _decoded_chunks
    for chunk in _encoded_chunks:
        decoded_chunk = tokenizer.decode(chunk)
        yield decoded_chunk
```



DAGWORKS

```python
@retry(wait=wait_random_exponential(min=1, max=40), stop=stop_after_attempt(3))
def summarized_chunk(
    chunked_text: str,
    summarize_chunk_of_text_prompt: str,
    openai_gpt_model: str,
    client: SerializeableOpenAIClient,
) -> str:
    """This helper function applies a prompt to some input content. In this case it returns a summarized chunk of text.

    :param chunked_text: a list of chunks of text for an article.
    :param summarize_chunk_of_text_prompt:  the prompt to use to summarize each chunk of text.
    :param openai_gpt_model: the openai gpt model to use.
    :return: the response from the openai API.
    """
    prompt = summarize_chunk_of_text_prompt + chunked_text
    response = client.client.chat.completions.create(
        model=openai_gpt_model, messages=[{"role": "user", "content": prompt}], temperature=0
    )
    return response.choices[0].message.content


def summarized_chunks(summarized_chunk: Collect[str]) -> str:
    """Joins the chunks from the parallel chunking process into a single chunk.
    :param summarized_chunk: the openai gpt model to use.
    :return: a single string of each chunk of text summarized, concatenated toge
    """
    return "".join(summarized_chunk)
```



DAGWORKS

Hamilton: LLM/RAG pipelines

AI engineers:

- Write simple Hamilton pipeline
  - Chunk
  - Summarize
  - Query
- Leverage `Parallelizable` for dynamic execution
  - Test locally on small dataset/synchronous
  - Abstract infrastructure away, scale up as needed
- Hook into eval frameworks

The Agenda

A mental model for trade-offs
Platforms to the rescue
Hamilton: a lightweight platform abstraction
Some applications
- ↳ ML Pipelines
- ↳ LLM/RAG

Zooming out

Wrapping up

High-level:

- There will always be a trade-off
- A platform (+abstraction) can make it less painful
  - Platforms inject/guide best practices
- Hamilton has lots of hooks to enable you to scale!
  - Good software practice for *free (testing, docs, etc...)*
  - Move along the curve
  - Startup → enterprise w/minimal code change

A new approach to data

Asset-based

- Think *what* you want, not *how* to compute it.

Declarative

- Don’t split between *how* it works and *where* its called
- The same code can model both

Portable

- Dataflows should run anywhere. Batch, online, etc...
- You should never be afraid to migrate.

DAGWORKS

Thank you!

Questions?

- https://twitter.com/elijahbenizzy
- https://www.linkedin.com/in/elijahbenizzy/
- https://github.com/dagworks-inc/hamilton
- elijah@dagworks.io
- linktr.ee/elijahbenizzy

DAGWORKS