



DAGWORKS



Bridging Classic ML Pipelines with the World of LLMs



Stefan Krawczyk & Elijah ben Izzy
PyData Global 2023



who are we

Stefan Krawczyk (CEO)

Elijah ben Izzy (CTO)

DAGWorks Inc. (YCW23)

Creators of **Hamilton**

20+ years in ML & Data platforms



iDIBON

2σ TWO SIGMA

STITCH FIX



HIRI

Honda Research Institute **US**





LLMs



Classic ML

E.g.
LTV prediction
Fraud detection
Spam classification
Personalization



LLMs

E.g.
Text summarization
Q&A Help bot
Generating text



Part I: 

Classic ML Pipelines \approx LLM Pipelines



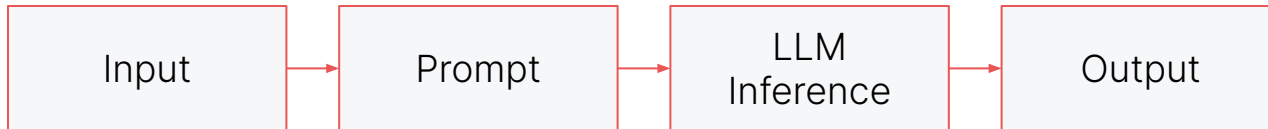
Let's compare some pipelines



Classic ML

E.g. LTV prediction

Let's compare: inference



LLMs

E.g. Text summarization



Classic ML

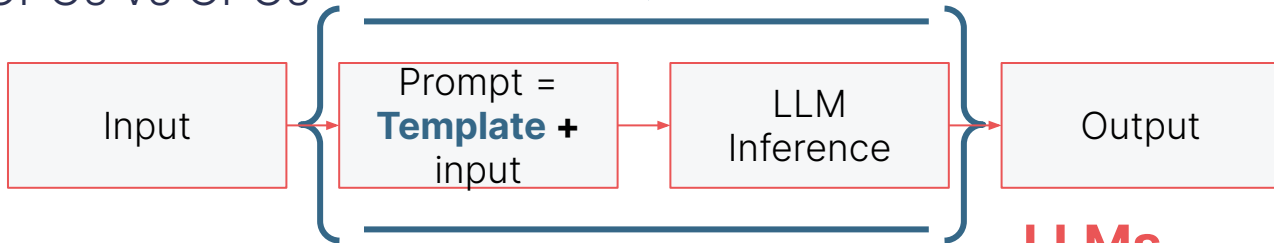
E.g. LTV prediction

Let's compare: inference



- Iteration pace is different!
- Model Versioning
- Observability
- Evaluation
- APIs vs GPUs vs CPUs

“model”



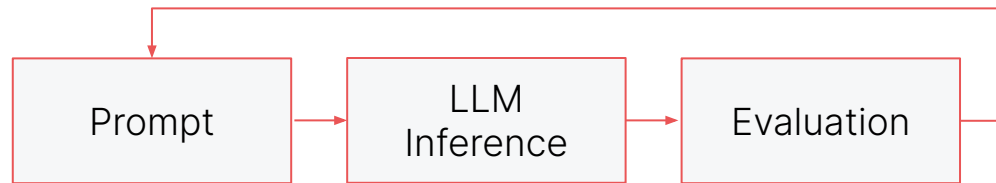
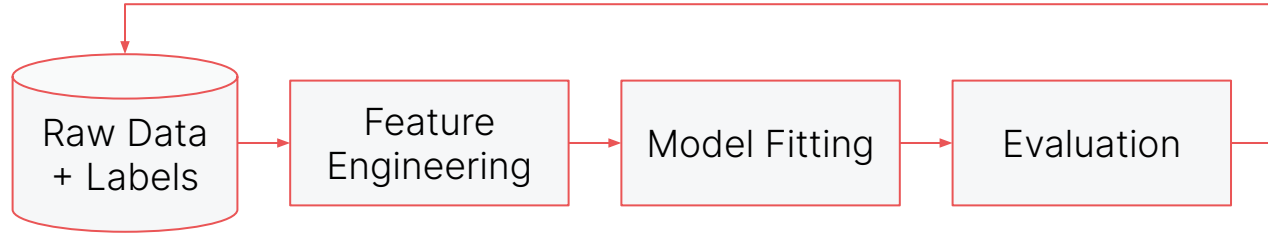
LLMs

E.g. Text summarization



Let's compare: "Training" 1/2

Classic ML

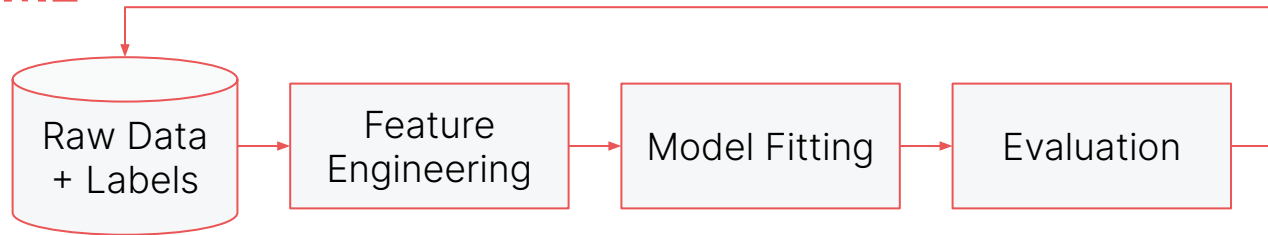


"Hyperparameter tuning"

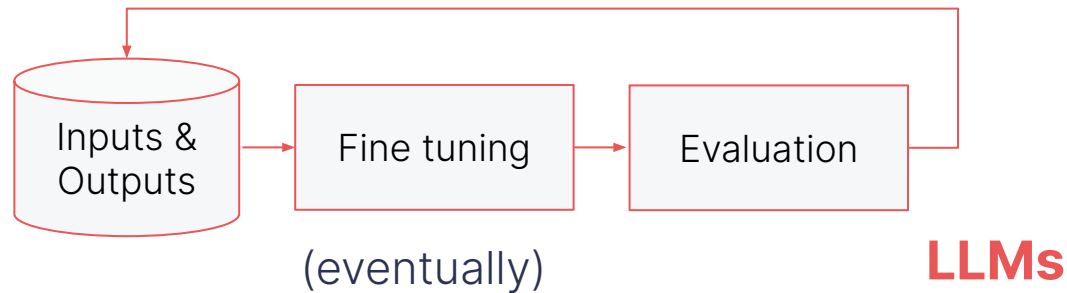
LLMs

Let's compare: "Training" 2/2

Classic ML



- Evaluation fuzzier with LLMs
- CPUs vs GPUs

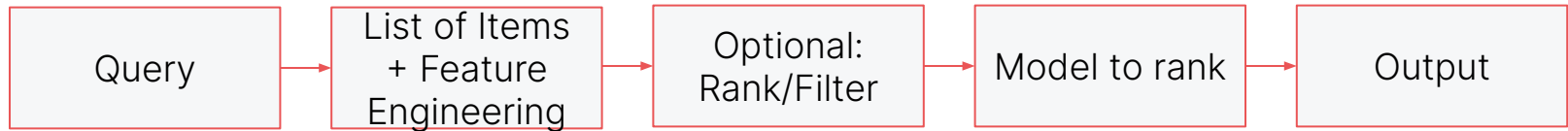




Classic ML

E.g. Recommending clothes

Let's compare: Rec. Sys. vs RAG



- Query → candidate set → model
- Same debugging, observability, evaluation needs
- DBs: Feature Stores vs Vector DBs vs DBs
- ETL: processes need to be built



LLMs

E.g. Knowledge Q&A

Classic ML requires a lot of software engineering

Classic ML image:

Pipeline inheritance 😲

Why?

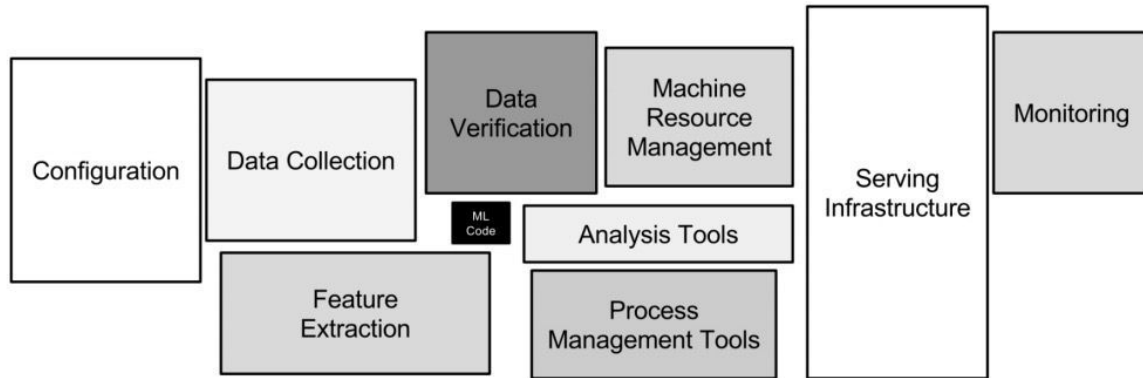
- Overengineered abstractions

General challenges:

- Testing
- Change/Modularity
- Reuse

Hidden Technical Debt in Machine Learning Systems

D. Sculley, Gary Holt, Daniel Golovin, Eugene Davydov, Todd Phillips
{dsculley, gholt, dgg, edavydov, toddphillips}@google.com
Google, Inc.



All of this requires a lot of software engineering

LLM world is no different:

~~Classic ML image:~~

Pipeline inheritance 🤪

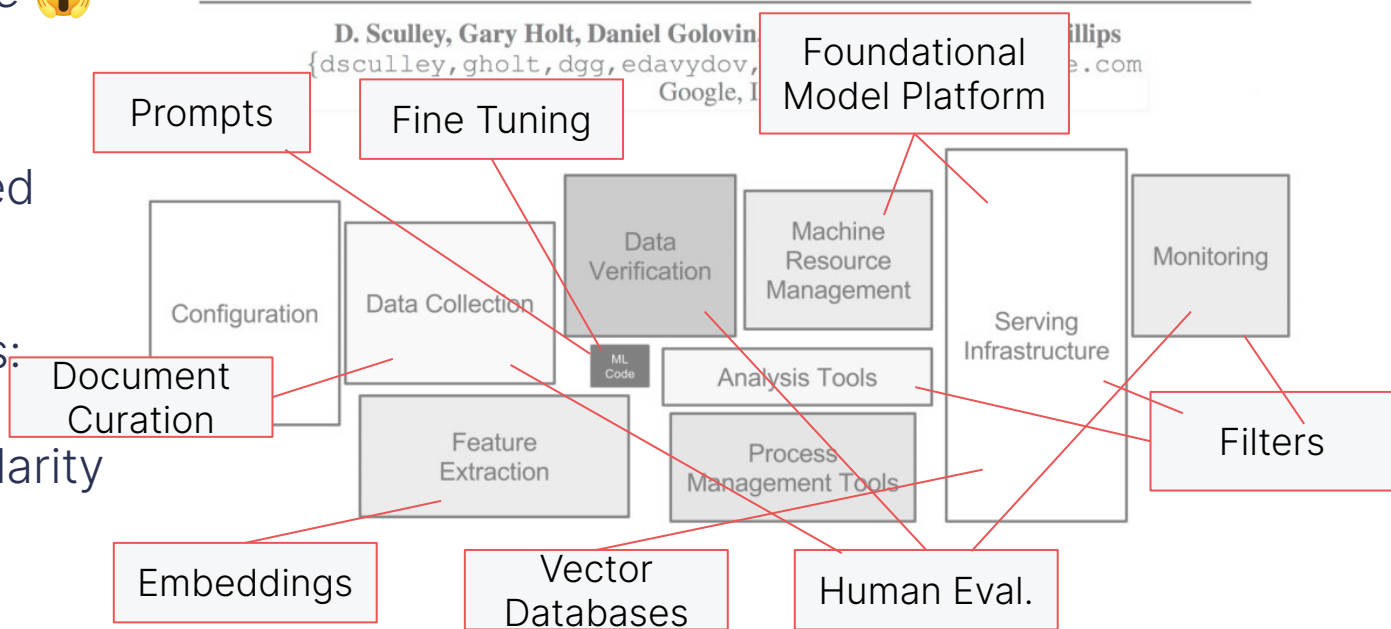
Why?

- Overengineered abstractions

General challenges:

- Testing
- Change/Modularity

LLM Systems Hidden Technical Debt in Machine Learning Systems





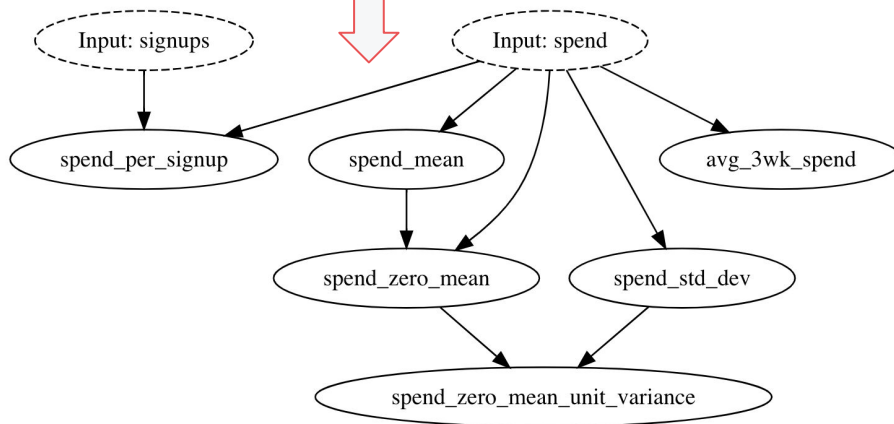
A simple unifying abstraction: Directed Acyclic Graphs (DAGs)





In general, Code, e.g. Pandas: it's a (micro) DAG

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





LLMs: Using Langchain

```
from langchain.llms import OpenAI
llm = OpenAI(temperature=0.9)

text = "Explain the concept of machine learning in"
      "one paragraph"
print(llm(text))
```


LLMs: Using Langchain, it's a (micro) DAG

```
from langchain.llms import OpenAI
llm = OpenAI(temperature=0.9)

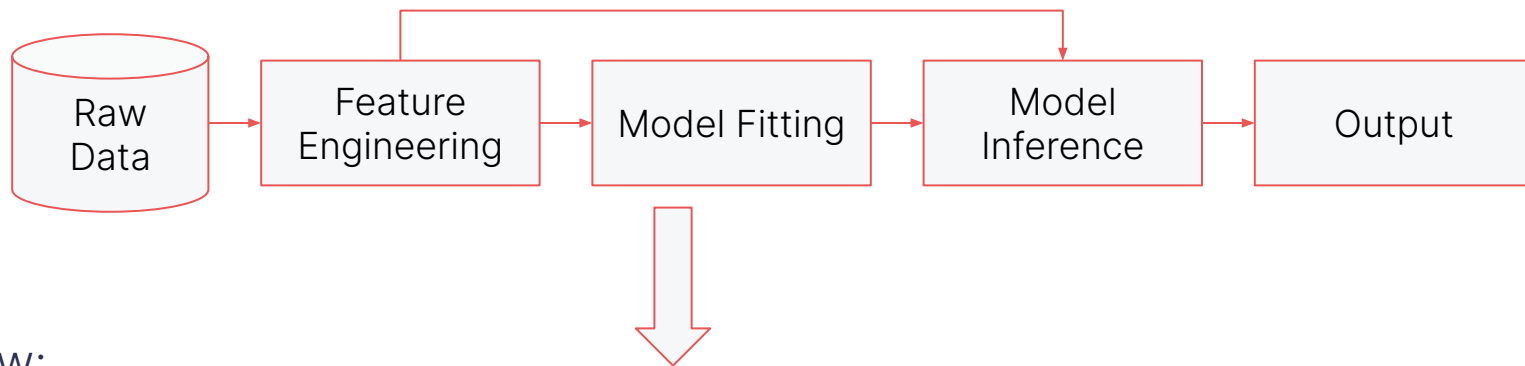
text = "Explain the concept of machine learning in"
one paragraph"
print(llm(text))
```



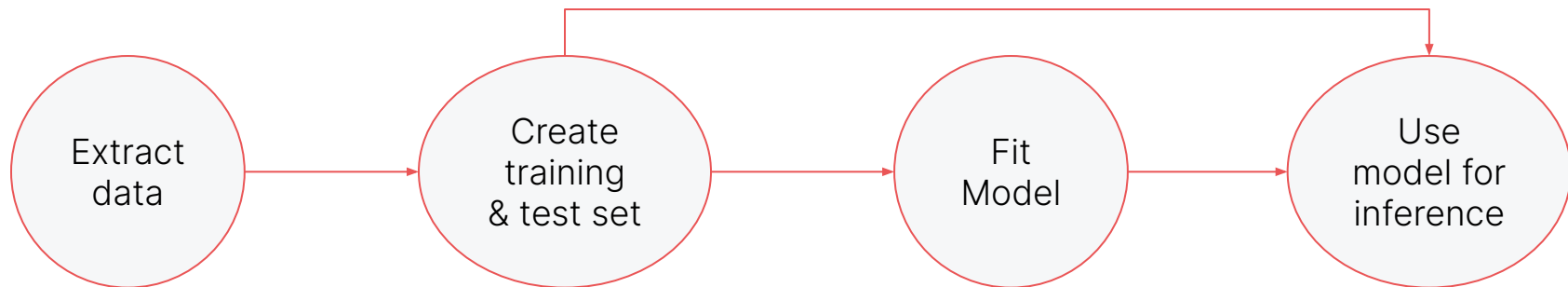
See https://www.reddit.com/r/LangChain/comments/13fcw36/langchain_is_pointless/



ML Pipelines are a (macro) DAG

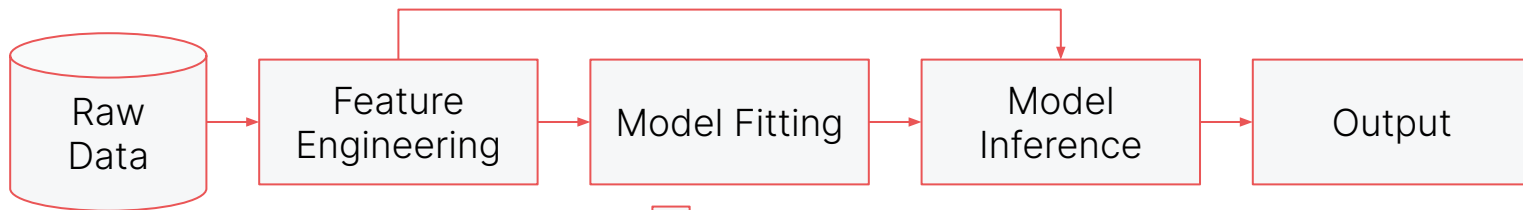


In Airflow:

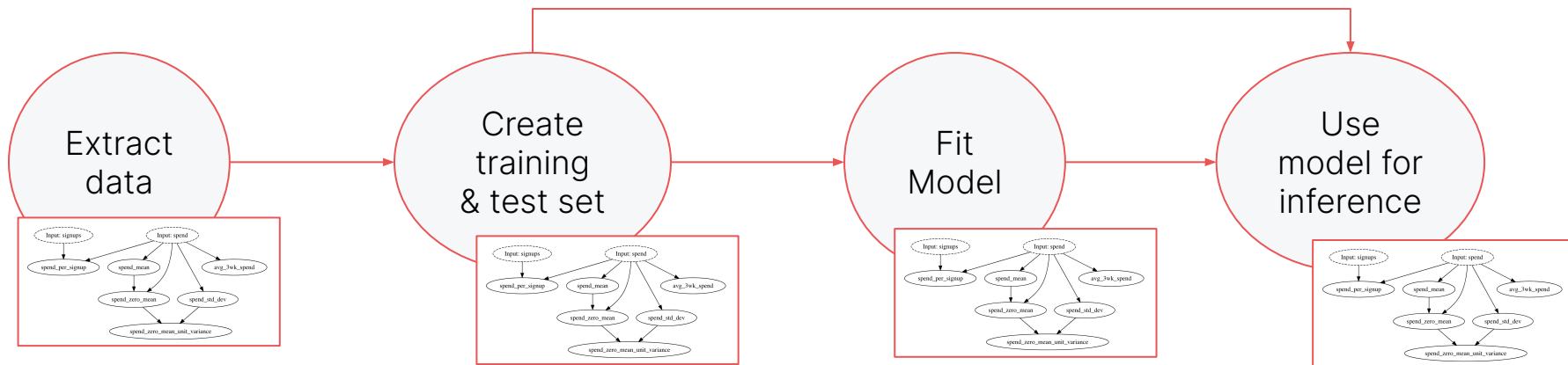




ML Pipelines are a DAG of DAGs



In Airflow:

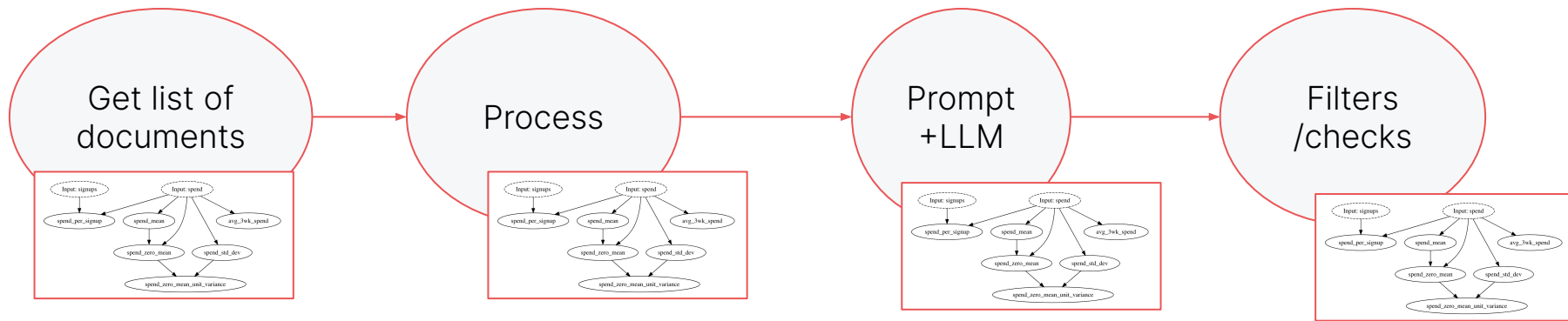
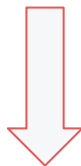




LLM Pipelines are also a DAG of DAGs



In Langchain/Custom code:



Part I Summary: ML Pipelines \approx LLMs Pipelines

1. Pipelines: LLM world is broadly equivalent to Classic ML
 - a. Shared software engineering challenges
 - b. But, with LLMs:
 - i. Applications can be developed much faster.
 - ii. So you need to design for change (modularity)
2. At their core, they can (and should!) be both modeled by DAGs.

Hot take:

“A DAG is all you need – the rest is an overengineered abstraction”



Part II: Hamilton



What is Hamilton?



What is Hamilton?

Micro-orchestration framework for defining DAGs using declarative functions

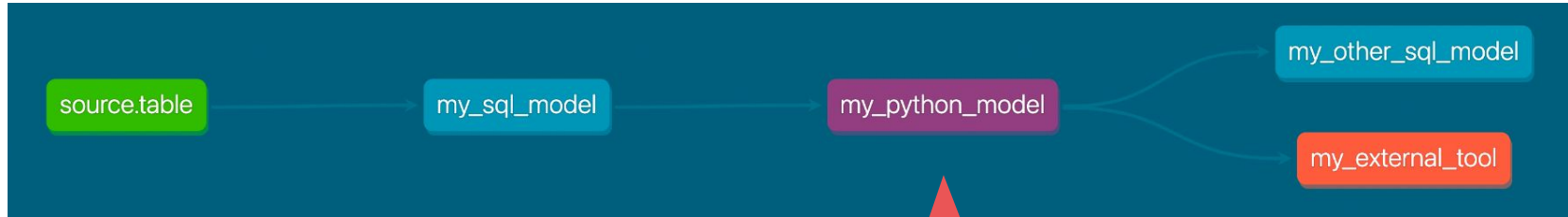
SWE best practices: testing documentation modularity/reuse

```
pip install sf-hamilton [came from Stitch Fix]
```

www.tryhamilton.dev ← uses pyodide!

Micro-orchestration vs Macro-orchestration

Macro-orchestration handles this whole thing:

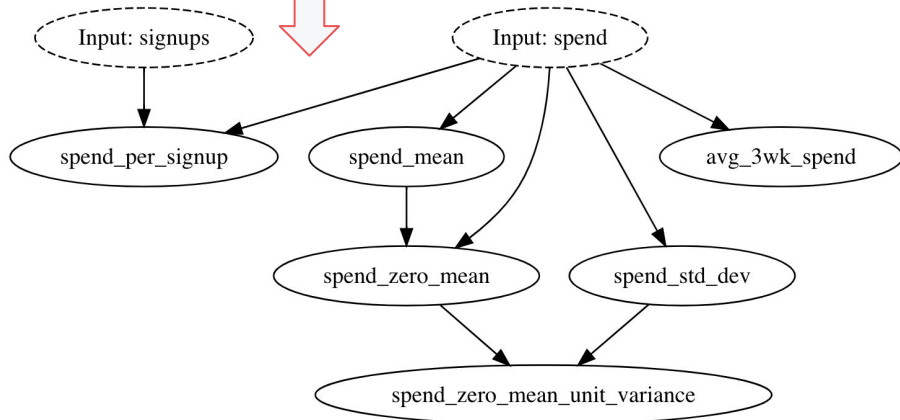


Micro-orchestration handles what happens within this step

What type of DAGs?

DAGs that represent your procedural code (i.e. the micro):

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





Declarative functions?

Functions *declare*:

- What they create in the DAG.
- 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?

And... 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
```

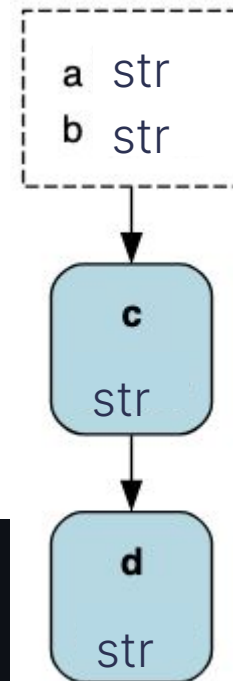
Full Hello World

(Note: works for any python object type)

Functions

```
# llm_chain.py
def c(a: str, b: int) -> str:
    """Creates prompt"""
    return f"Some prompt using {a} & {b}"

def d(c: str) -> str:
    """Transform/send to LLM ..."""
    llm_response = _llm_api_call(c)
    return llm_response
```



Driver says what/when to execute

```
# run.py
from hamilton import driver
import llm_chain
dr = driver.Driver({'a': ..., 'b': ...}, llm_chain)
df_result = dr.execute(['c', 'd'])
print(df_result)
```



Things to mention, but 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 Examples



Hamilton for ML (feature engineering)

TL;DR

1. Define one function per feature
2. Join together as part of the Hamilton Driver
3. Utilize `@config.when` to swap between online/offline implementations
4. Expand to more components of the ML lifecycle



Hamilton for Classic ML (feature engineering)

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

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

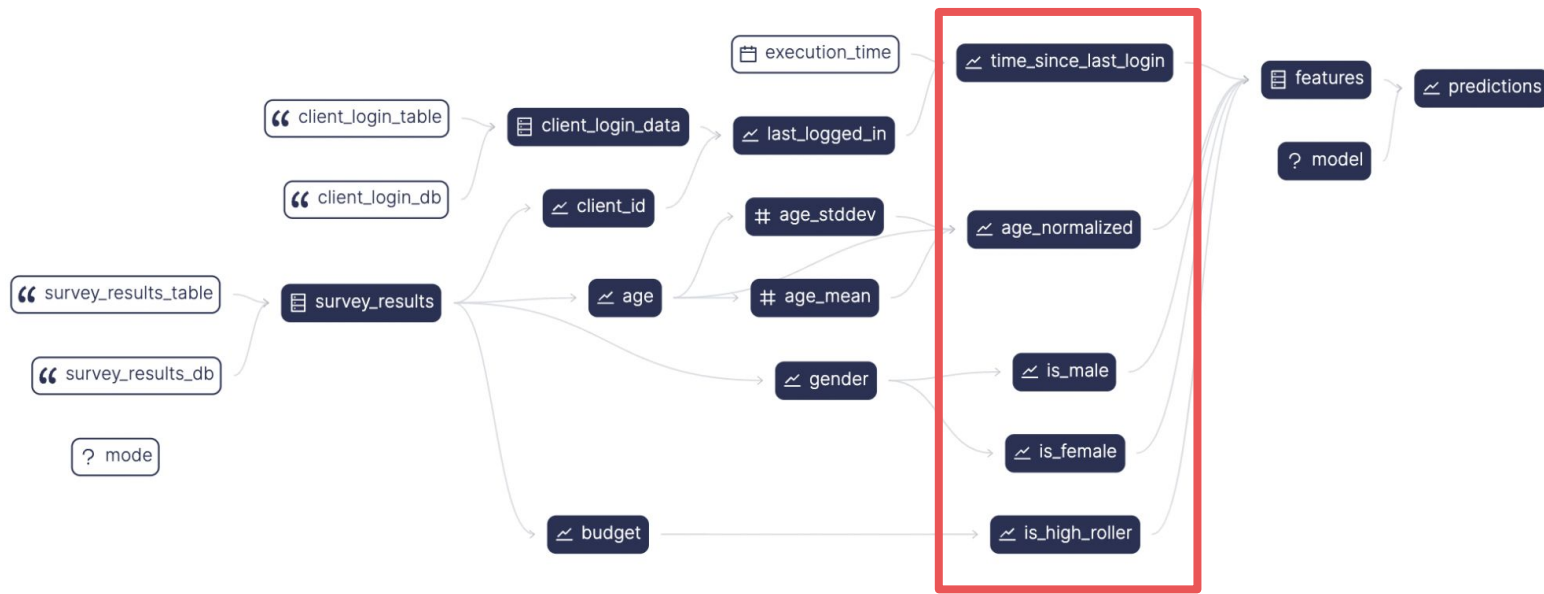
```
def is_high_roller(budget: pd.Series) -> pd.Series:  
    return budget > 100
```

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

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



Hamilton for Classic ML (feature engineering)





Hamilton for Classic ML (feature engineering)

Swap between batch/online with @config

```
#data_loaders.py
@config.when(mode="batch")
@extract_columns("budget", "age", "gender", "client_id")
def survey_results(survey_results_table: str, survey_results_db: str) -> pd.DataFrame:
    """Map operation to explode survey results to all fields
    Data comes in JSON, we've grouped it into a series.
    """
    return utils.query_table(table=survey_results_table, db=survey_results_db)

@config.when(mode="batch")
def client_login_data_batch(client_login_db: str, client_login_table: str) -> pd.DataFrame:
    return utils.query_table(table=client_login_table, db=client_login_db)

# join.py
@config.when(mode="batch")
def last_logged_in_batch(client_id: pd.Series, client_login_data: pd.DataFrame) -> pd.Series:
    return pd.merge(client_id, client_login_data, left_on="client_id", right_index=True)[
        "last_logged_in"
    ]
```



```
@config.when(mode="online")
@extract_columns(
    "budget",
    "age",
    "gender",
)
def survey_results__online(client_id: int) -> pd.DataFrame:
    """Map operation to explode survey results to all fields
    Data comes in JSON, we've grouped it into a series.
    """
    return utils.query_survey_results(client_id=client_id)

@config.when(mode="online")
def last_logged_in_online(client_id: int) -> pd.Series:
    return utils.query_login_data(client_id=client_id)["last_logged_in"]
```



Hamilton for Classic ML (feature engineering)

Add in ML inference/training as needed...

```
f components.model.features

def features(
    time_since_last_login: pd.Series,
    is_male: pd.Series,
    is_female: pd.Series,
    is_high_roller: pd.Series,
    age_normalized: pd.Series,
) -> pd.DataFrame:
    """Aggregate all features into a single dataframe.
    :param time_since_last_login: Feature the model cares about
    :param is_male: Feature the model cares about
    :param is_female: Feature the model cares about
    :param is_high_roller: Feature the model cares about
    :param age_normalized: Feature the model cares about
    :return: All features concatenated into a single dataframe
    """
    return pd.DataFrame(locals())
```

```
f components.model.predictions

def predictions(features: pd.DataFrame, model: Model) -> pd.Series:
    """Simple call to your model over your features."""
    return model.predict(features)
```

```
f components.model.model

def model() -> Model:
    return Model()
```



Hamilton for LLM pipelines

Define capabilities (chains) as DAGs. Nodes (fns) are:

- Prompts
- API calls
- External data queries
- Data transformation

[optional] Control flow (agent) to tie them together:

- Take result, feed to next execution
- Use LLM to decide next DAG

Swap between foundational models, vector stores, etc... with **@config.when**



Hamilton for LLMs

Caption an Image

```
f caption_images.core_prompt
```

```
def core_prompt() -> str:  
    return "Please provide a caption for this image."
```

```
f caption_images.prompt
```

```
def prompt(  
    core_prompt: str,  
    additional_prompt: Optional[str] = None,  
    descriptiveness: Optional[str] = None,  
) -> str:  
    """Returns the prompt used to describe an image"""  
    out = core_prompt  
    if descriptiveness is not None:  
        out += f" The caption should be {descriptiveness} descriptive."  
    if additional_prompt is not None:  
        out += f" {additional_prompt}"  
    return out
```

```
f caption_images.processed_image_url
```

```
def processed_image_url(image_url: str) -> str:  
    """Returns a processed image URL -- base-64 encoded if it is local,  
    otherwise remote if it is a URL"""  
    is_local = urllib.parse.urlparse(image_url).scheme == ""  
    if is_local:  
        # In this case we load up/encode  
        encoded_image = _encode_image(image_url)  
        extension = image_url.split(".")[-1]  
        return f"data:image/{extension};base64,{encoded_image}"  
    # In this case we just return the URL  
    return image_url
```

```
f caption_images.generated_caption
```

```
def generated_caption(  
    processed_image_url: str,  
    prompt: str,  
    model: str = "gpt-4-vision-preview",  
    max_tokens: int = 300) -> str:  
    """Returns the response to a given chat"""  
    messages = [{  
        "role": "user",  
        "content": [  
            {  
                "type": "text",  
                "text": prompt,  
            },  
            {  
                "type": "image_url",  
                "image_url": {  
                    "url": f"{processed_image_url}"  
                }  
            }  
        ]  
    }]  
    client = openai.OpenAI()  
    response = client.chat.completions.create(  
        model=model,  
        messages=messages,  
        max_tokens=max_tokens,  
    )  
    return response.choices[0].message.content
```

Hamilton for LLMs

Generate an Image:

```
f generate_images.prompt
```

```
def prompt(
    base_prompt: str,
    style: str = None,
    additional_prompt: str = None) -> str:
    """Returns the prompt used to generate an image"""
    prompt_out = base_prompt
    if style is not None:
        prompt_out += f" The image should be in the {style} style."
    if additional_prompt is not None:
        prompt_out += f" {additional_prompt}"
    return prompt_out
```

```
f generate_images.generated_image
```

```
def generated_image(prompt: str, size: str = "1024x1024", hd: bool = False) -> str:
    """Returns the generated image"""
    client = openai.OpenAI()

    response = client.images.generate(
        model="dall-e-3",
        prompt=prompt,
        size=size,
        quality="standard" if not hd else "hd",
        n=1,
    )
    image_url = response.data[0].url
    return image_url
```



save_image



Hamilton for LLMs

Build out a toolbox

- Select which DAG to execute based on context
- Run in process
- Do something with the outputs
 - Display to user
 - Store as state/context, feed back to next execution
 - Write back to storage for fine-tuning

Embrace chaos – feed a caption from chatGPT to an image in Dalle with a while loop + two lines of Hamilton code...*

*Blog post coming soon!



Hamilton for LLMs

In a mesmerizing collision of cosmic and terrestrial vistas, we gaze upon an otherworldly landscape where the impossible becomes possible. The scene is divided into two striking halves by a brilliant, vertical beam of light, signifying a rift between two dimensions or realities. On the left, we are treated to the familiar serenity and natural beauty of an earthly beach under a night sky. Foam-laced waves gently kiss the sandy shore, swirling around pockets of calm water that reflect the celestial wonders above. A crescent moon, accompanied by a scatter of twinkling stars and a nebulous galactic canvas, hangs suspended in the quiet darkness. Rich hues of blue and browns palette the tranquil nocturnal beach.

Transitioning through the radiant divide, the right paints a stark contrast, an epic swirl of galactic clouds in an astonishing dance of light and color. We are catapulted into the vastness of space, witnessing the ethereal beauty of a swirling blue galaxy adorned with starbursts and clouds of interstellar dust, which emit a myriad of colors: from warm golds to cool purples, embodying the chaotic and ever-evolving nature of the universe. This otherworldly skyscape also bathes a duplicate beach in its surreal light, but here, the sea churns with greater ferocity, infused with the tumultuous energy of the cosmos.

This scene, likely the work of digital artistry, offers a vision bordering on science fiction, encouraging the viewer to contemplate the beauty and mystique of both our own planet and the infinite universe that embraces it. The composition speaks of contrasts: tranquil versus tumultuous, warm versus cool colors, the familiar versus the unknown, and the possible versus the fantastical. The contrast is not just visual; it's thematic, exploring the intersection of our reality with the boundless realms of the imagination





Hamilton for both

Many problems you hit will require both...

E.G: customize embeddings:

- Pass in pairs of similar/dissimilar text
- [LLM] Query foundational model for embeddings
- [ML] Train a model on your pairs
- [ML] Project onto that space



Hamilton for both

Hamilton for LLM + ML workflows enables you to:

- Allows you to BYO tooling (LLMs, ML, your DB, ...)
- Swap out components, implementations, etc
- Visualize, test, reuse components
- Do ^^ with one tool



Part II: Hamilton Summary




Modeling your DAGs as self-descriptive functions allows you to:

- Use the same tooling for LLM + ML + data processing
- Rerun in multiple contexts to save code, headaches
- Reuse DAGs to build really cool things!
- Build all of ^^ as modular, testable, self-documenting software
 - Never fear inheriting someone else code; 😱 → 😎



Overall Summary

Talk Summary

-  Pipelines: LLMs are broadly equivalent to classic ML
 - But LLM field is moving/changing quickly;
 - Traditional ML isn't going away.
-  You can model ML pipelines as DAGs as well as LLM pipelines
-  Hamilton is your unifying toolset
 - SW best practices for free!
 - Can do both LLMs + ML and is made to handle change

Next steps:

1. Download Hamilton `pip install sf-hamilton` <10 mins to get started.
2. <https://www.tryhamilton.dev/> to learn the basics.
3. hub.dagworks.io to get started with something quickly
4. Star the repository - <https://github.com/DAGWorks-Inc/hamilton>
5. Join [slack](#) if you have questions!

Hamilton Dataflow Hub

Your place to find Hamilton dataflows

Integrate with your code in 5 minutes





Fin. Questions?