Gen AI using Airflow 3

Introduction

Ash Berlin-Taylor
Airflow Committer & PMC Member
Engineering Leader @ Astronomer

Kaxil Naik
Airflow Committer & PMC Member
Engineering Leader @ Astronomer

The Changing AI Landscape

Why New Solutions Are Needed!

Evolving AI Landscape

Explosion of AI Models

Cost Optimization

Increased Focus on
Data Privacy & Control

Increasing Need for
Experimentation

GPUs
are easily accessible

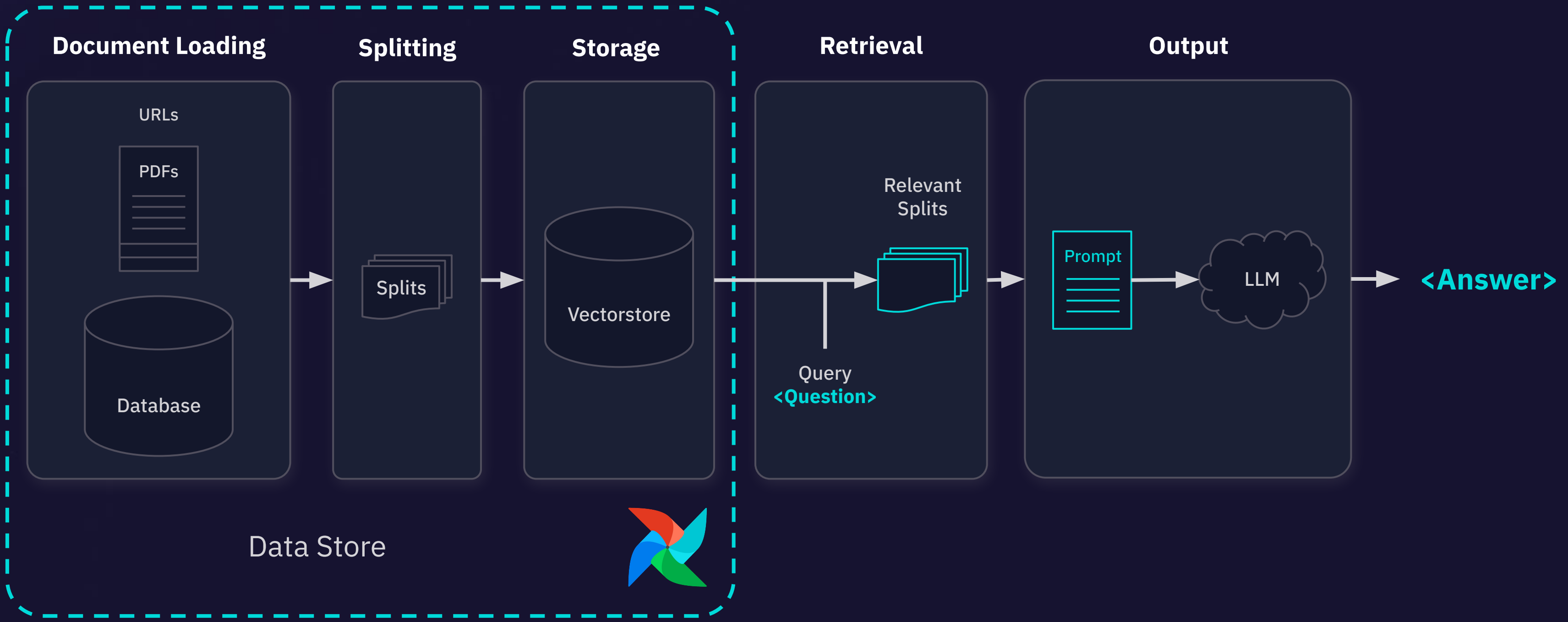
Growing Complexity of
AI Workflows

RAG

Retrieval-Augmented Generation

What is RAG?

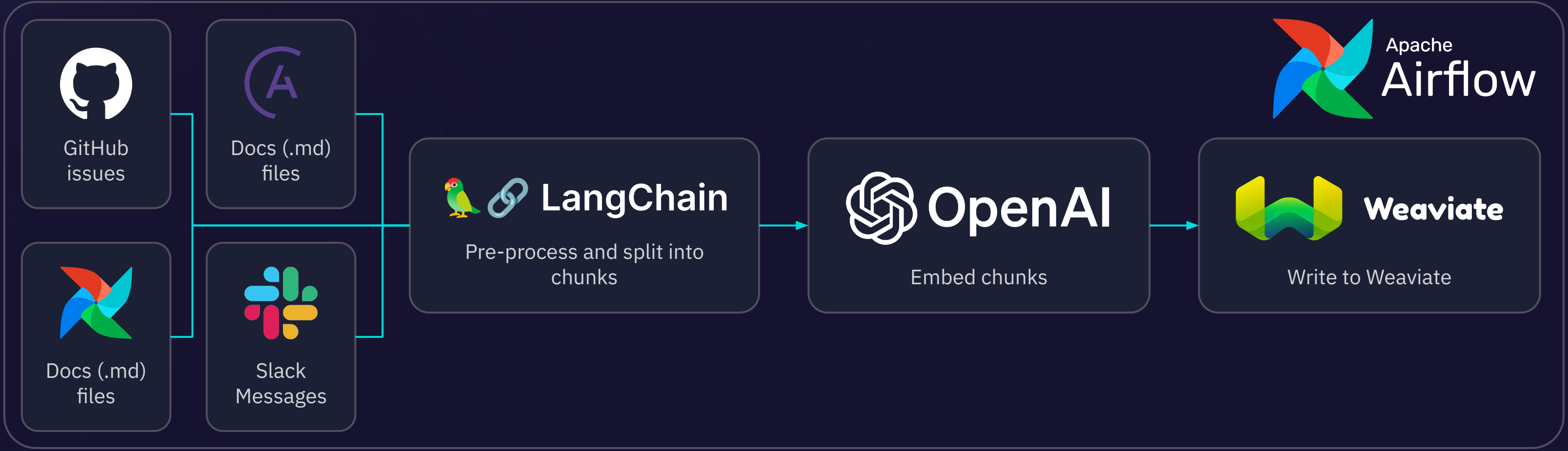
Typical Architecture for Q&A use-case using LLM



RAG (Ingestion) as an Airflow DAG

Large data sets	Dynamic Mapping for large number of incoming datasets (website content, directories of files, .)
Unstructured Data	Reading, chunking, and Transformation Python libraries and frameworks for above Eg: Unstructured, LangChain, etc.
Generate and Store Embeddings	Using AI providers: Open AI, Cohere, etc. Store into Weviate, PgVector, ...

Ask Astro: Data Ingestion, Processing, and Embedding



- Airflow gives a **framework to load data from APIs** & other sources into LangChain
- LangChain helps pre-process and **split documents into smaller chunks** depending on content type

- After content is split into chunks, each chunk is **embedded into vectors** (semantic representations)
- Those vectors are **written to Weaviate** for later retrieval

RAG (Ingestion) as an Airflow DAG

```python
from airflow.decorators import dag, task
from airflow.providers.weaviate.operators.weaviate import
WeaviateDocumentIngestOperator
airflow_docs_base_url = "https://airflow.apache.org/docs/"


@dag(schedule="0 5 * * 2", ... ,)
def ask_astro_load_airflow_docs():
    from include.tasks import chunking_utils
    from include.tasks.extract import airflow_docs

    extracted_airflow_docs = task(chunking_utils.split_html).expand(
        dfs=[airflow_docs.extract_airflow_docs(docs_base_url=airflow_docs_base_url)]
    )

    _import_data = WeaviateDocumentIngestOperator.partial(
        class_name=WEAVIATE_CLASS,
        existing="replace",
        document_column="docLink",
        batch_config_params={"batch_size": 7, "dynamic": False},
        verbose=True,
        conn_id=_WEAVIATE_CONN_ID,
        task_id="WeaviateDocumentIngestOperator",
    ).expand(input_data=[extracted_airflow_docs])
ask_astro_load_airflow_docs()
```

ASTRONOMER

Challenges

Python Dependencies	Supporting varied Python configurations and dependencies between tasks
Selective GPU Execution	Keeping main execution on CPUs, only selectively call out to GPUs on remote clusters
Dynamic model choice	Change LLM model in response to cost/performance/new features

How Airflow 3 Helps

Solution part1: Task Execution Interface

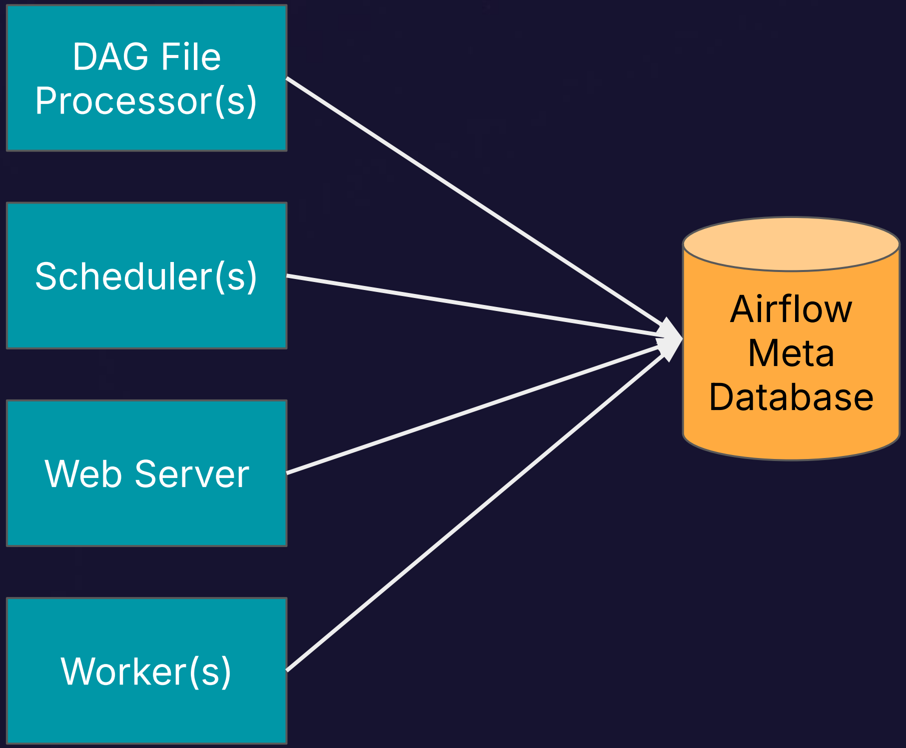
Python dependencies:

- Different python dependencies for different tasks

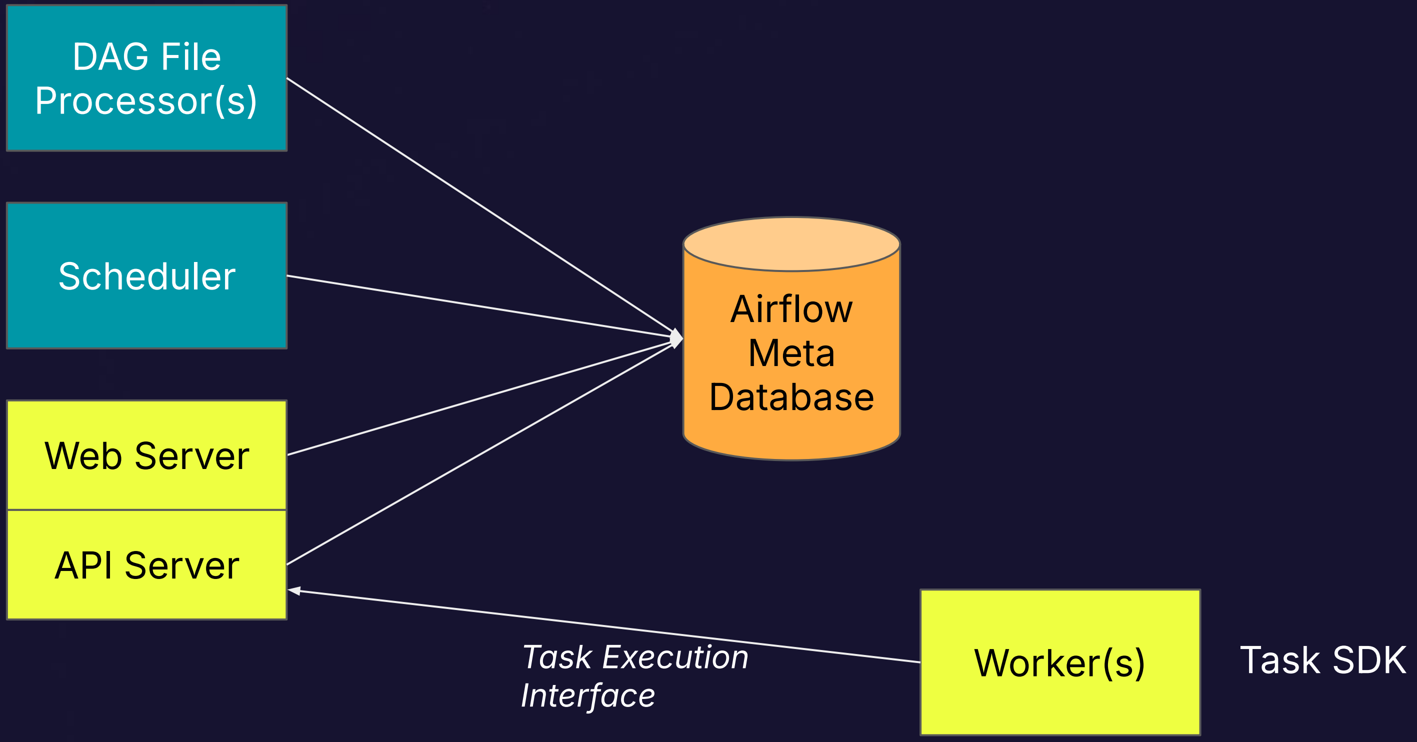
Cost-optimal Task Execution:

- Data cleaning, Data transformation with CPUs
- Model training w/ GPU as needed - less than 10% of tasks in a DAG

Current Airflow architecture



Architectural decoupling:
Task Execution Interface



Solution part2: `common.llm`

Selective model choice:
- Different model performance & accuracy
- Complexity vs. Cost & response time tradeoff
- Dynamic selection based on task requirements and constraints

AI provider selection:
- Based on execution environment (e.g., GPUs, CPUs)
- Data security constraints for external vs local models

Solution part2: `common.llm`

```
LLMOperator(
    task_id="openai_task",
    embedding="OpenAI",
    source_dataset=Dataset("/usr/local/airflow/dags/data/github.pdf"),
    target_dataset=Index(uri="pgvector://postgres", name="airflow_summit_test"),
    embedding_params={
        "embedder_model": "text-embedding-ada-002",
        "encoding_name": "cl100k_base",
    },
)
```

Solution part2: `common.llm`

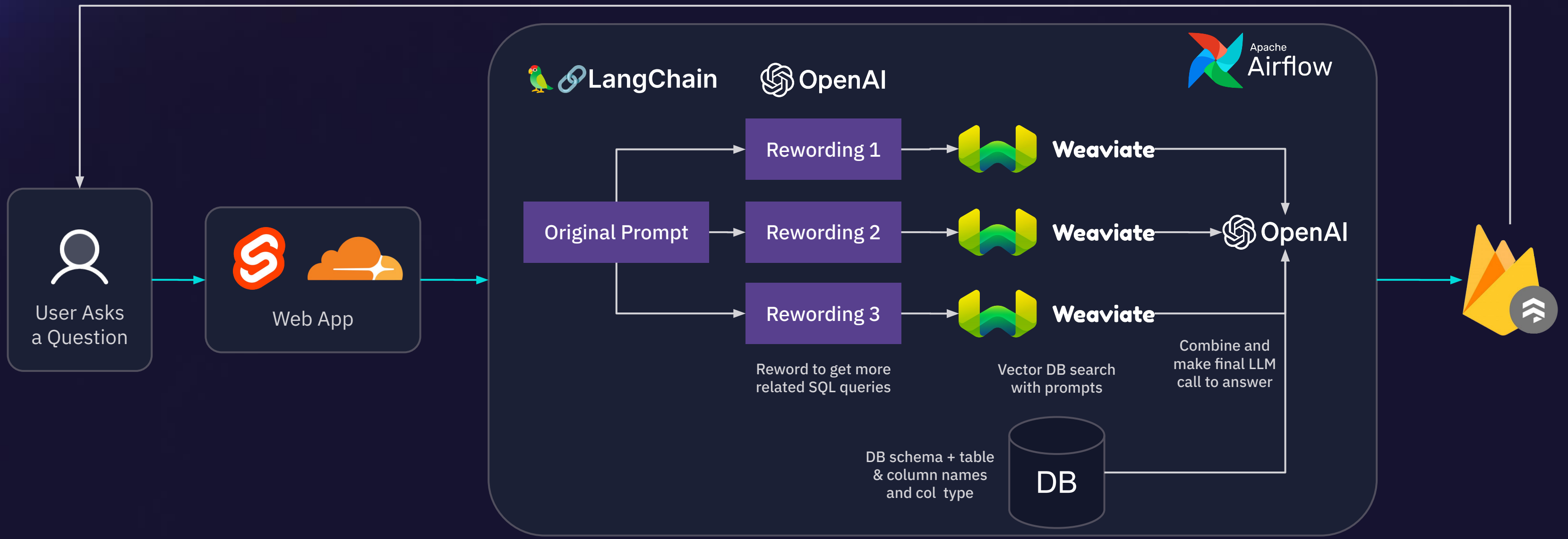
```python
LLMIngestOperator(
    task_id="dynamic_llm_task",
    embedding="auto",
    source_dataset=Dataset("/usr/local/airflow/dags/data/github.pdf"),
    target_dataset=Index(uri="pgvector://postgres", name="airflow_summit_test"),
    existing="replace",

    # Parameters for dynamic decision-making
    embedder_options=[
          {"provider": "OpenAI", "model": "text-embedding-ada-002", "use_gpu": False, "cost": "medium"},
          {"provider": "Local", "model": "local-embedder", "use_gpu": True, "cost": "low"},
          {"provider": "HuggingFace", "model": "bert-large-uncased", "use_gpu": True, "cost": "high"}
    ],
    selection_criteria={
        "cost_threshold": "medium",  # Dynamically choose based on cost constraints
        "use_gpu_if_necessary": True,  # Use GPU if the task requires higher performance
        "privacy_sensitive": True  # Use local models if the data is sensitive
    }
)
```

Example Inference as an Airflow DAG

Rephrase the question	Use both original and re-phrased versions
Submit and get results	Query all versions of the question De-duplicate the results
Return results	Optionally verify and rank the results Return results with sources

AI SQL Assistant: Inference



Users enter a question in Natural language in the AI Assist Editor on the UI

- Original prompt gets reworded 3x using gpt-3.5-turbo
- DB Schema incl. table & column names & type is retrieved
- Answer is generated by combining answers from each prompt and making a gpt-4 call

Challenges and upcoming enhancements

Batch-triggered Dag Runs & Experimentation	Eliminate the execution date constraint Concurrent runs of the same DAG i.e. non-data-interval DAGs.
Dynamic model choice	`commom.llm` to dynamically change AI provider and model
Synchronous DAG run	Inference DAGs return results upon completion Trigger API to support synchronous execution

Solution part3: Ad-hoc Dag Runs

Batch-triggered Dag Runs

- Non-data-interval based: No reliance on execution dates or schedules.
- Ad-hoc invocation via API calls for inference allowing multiple instances to be triggered by API calls at the same time.

Enables Experimentation

- Run the same DAG with different parameters simultaneously, independent of the execution date.
- Ideal for AI/ML workflows like:
  - Experiment with multiple models for embedding
  - Retraining models
  - Experimenting a new data source for RAG
  - Hyperparameter tuning

Solution part4: Experimentation Tracking

Data Assets

- **Dataset** renamed to **Data Asset** to include Models, Reports, Embedding etc
- **Versioned** Assets: Improved experiment tracking & Iterative changes
- Enhanced UI support that allow visualization of “Data Asset **Metadata**”.
    - Example: RMSE value changes due to different parameters
- Audit: Every version of data assets can be audited and compared across different experimental runs.

Solution part5: Synchronous DAG run

Consumer of Inference DAG runs need results:

- Current model: Final Task in DAG to store results in Blob storage
- Ideal to add API support for it
- Will support long-running DAGs, since timing is unpredictable

Example:

- Laurel: Automated timekeeping
- Does not require “real-time chatbot style responses”

Other examples:

- Evaluation of mortgage applications

Solution part5: "Synchronous" DAG run

```python
@dag
def workflow():
    @task
    def prepare_data(): ...

    llm_op = LLMOperator(task_id="openai_task")

    prepare_date() >> llm_op

    # By returning the task, this marks it as the "return" value for the API
    return llm_op

workflow()
```

How Airflow 3 helps?

common.llm

Explosion of AI Models

Cost Optimization

common.llm

Task Execution Interface

Increased Focus on Data Privacy & Control

Increasing Need for Experimentation

Ad-hoc Dag Runs

Data Assets

Task Execution Interface

GPUs are easily accessible

Growing Complexity of AI Workflows

Sync. DAG run

In Summary

Many organizations already using Airflow for Gen AI applications

We need your feedback as we add these capabilities into Airflow 3
Recruiting beta users:

- Building Gen AI platforms and use cases

Come speak at the next Airflow Summit about your use case on Airflow 3!