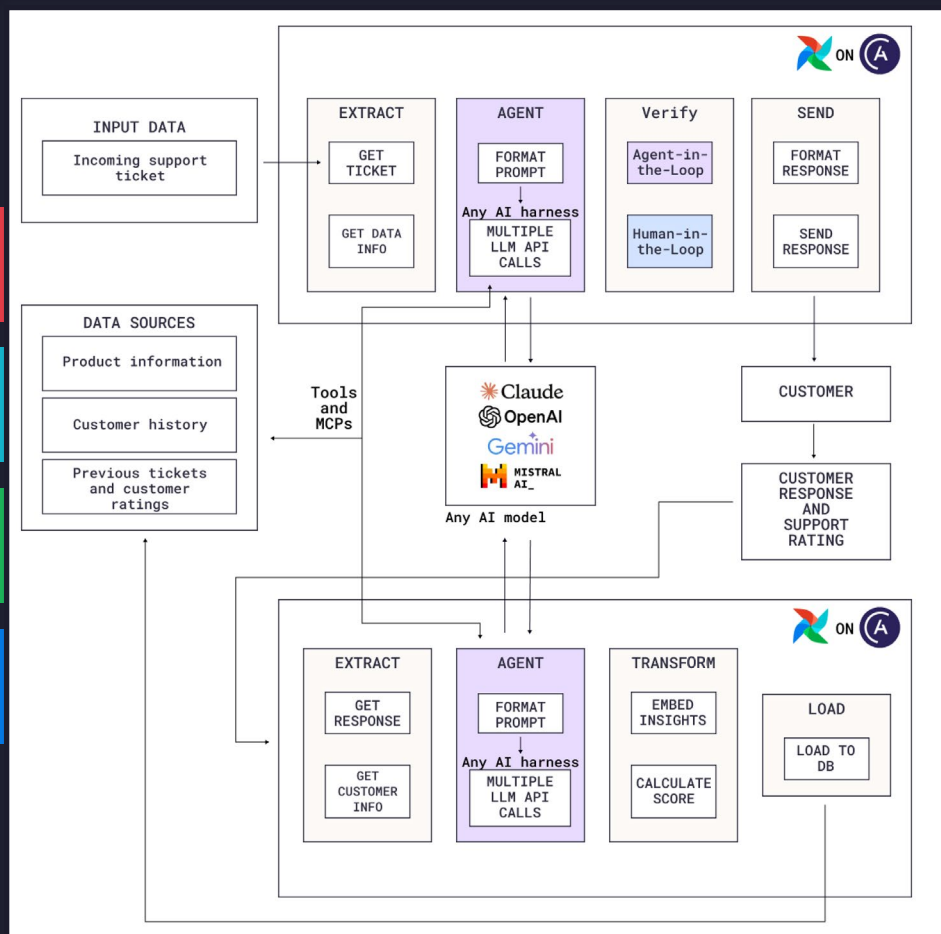


A DATA ENGINEER'S GUIDE TO

AI ORCHESTRATION WITH APACHE AIRFLOW[®]



Macro orchestration for LLMs, Agents, AI evals and
context engineering in one place

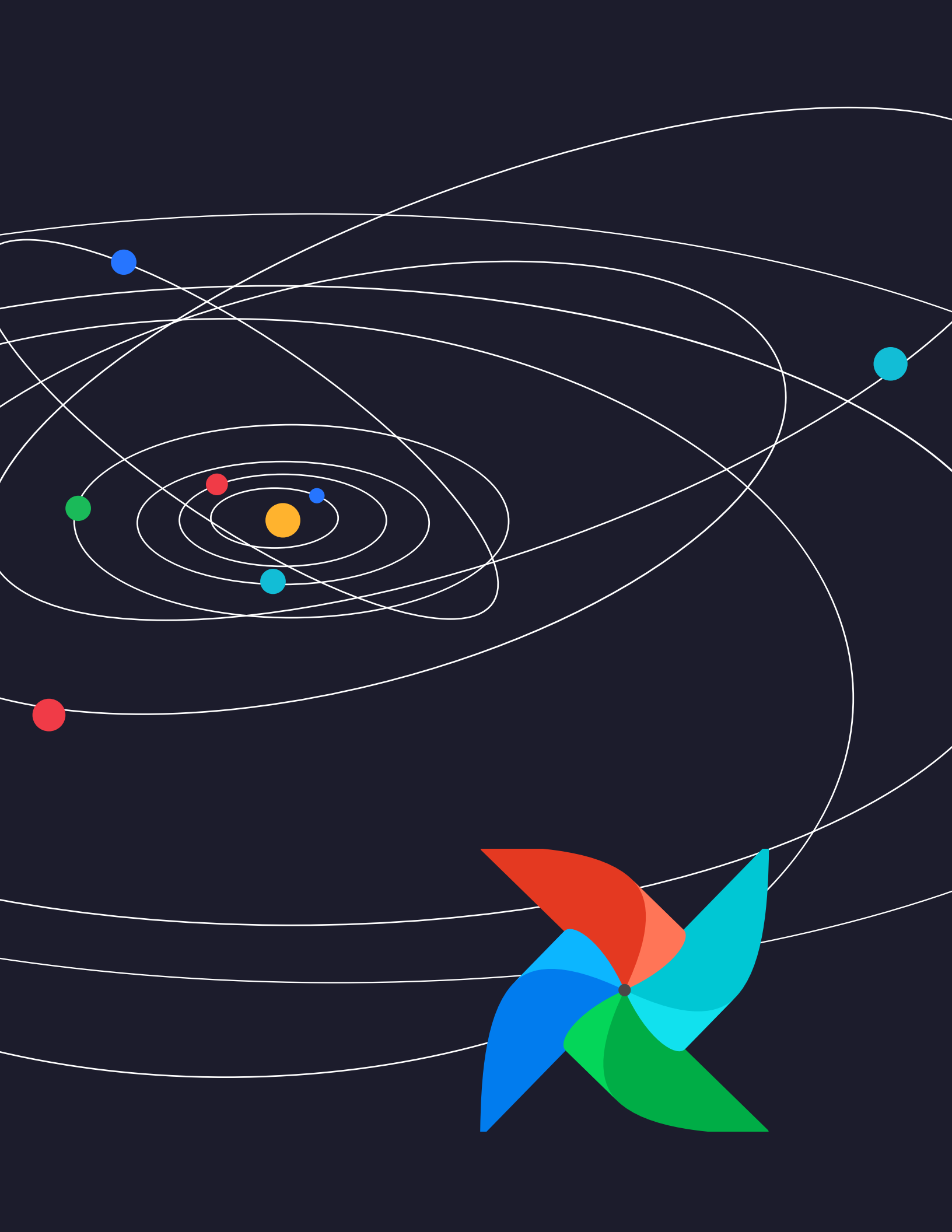
This book was written using Airflow 3.3 and the Common AI provider 0.7.0.

ASTRONOMER

Table of Contents

Introduction	5
Definitions	10
Part 1: AI orchestration patterns	12
LLM Orchestration	15
LLM summarization	19
LLM categorization and routing	22
LLM sentiment analysis	27
LLM transformation	29
LLM extraction of structured information from unstructured data	32
LLM-based task retries	36
LLM asset generation	38
Text-to-SQL	39
Agent Orchestration	43
Toolsets in the Common AI provider	45
Data exploration agent	48
Support ticket agent	50
Agents and humans in the loop	51
Self-improving agents: context graphs and decision traces	54
Dag-as-a-Tool	56
Root-cause analysis and self-healing pipelines with Otto	59
Multi-Agent Orchestration	63
Orchestrator-worker patterns	63
Agentic councils	64
Agentic software development	66
Context Engineering with Airflow	67
RAG pipelines: Document embedding	68

AI evals with Airflow	70
Evaluation signals arrive at different times	71
Gathering the signals	72
Agent harnesses, micro vs macro orchestration	74
Use LangChain/LangGraph with Airflow	74
Use Temporal with Airflow	75
Use CrewAI with Airflow	78
 Part 2: Airflow features for AI	 80
Common AI provider	81
Installation	82
@task.llm	83
@task.agent	86
Toolsets	88
Pre-built toolsets	89
Using Airflow hooks as toolsets	91
Human-in-the-loop review	91
Durable execution	93
Output type	93
Human-In-The-Loop	95
Dynamic Task Mapping	98
Scheduling	100
 Next steps	 102
 Resources	 103



Introduction

Right now, most AI initiatives are stuck as prototypes¹. At Astronomer we talk to some of the largest and most forward-thinking organizations every day, so we set out to answer: **what is missing to put AI agents and LLM inference into production?**

Orchestration. At every level.

The challenge starts with the data foundation: well-governed and documented data forms the basis for any AI initiative. Next, LLMOps pipelines are added. Some orchestrate context engineering, the most well-known example being retrieval-augmented generation (RAG) workflows. Others consistently gather data about your AI output, such as customer feedback over time for evaluation and monitoring purposes. These AI eval pipelines are necessary even before you orchestrate any AI inside Airflow, to evaluate the ROI from AI in chat apps, coding assistants, and microservices.

Once data orchestration has matured to this stage, LLMs and agents are put into the Dags themselves in the AI orchestration patterns this book covers, often starting with single-turn model API calls as an Airflow task (LLM orchestration) and progressing to multi-turn loops with tool calls (Agent orchestration). Finally, use cases are added that coordinate multiple agents in complex workflows.

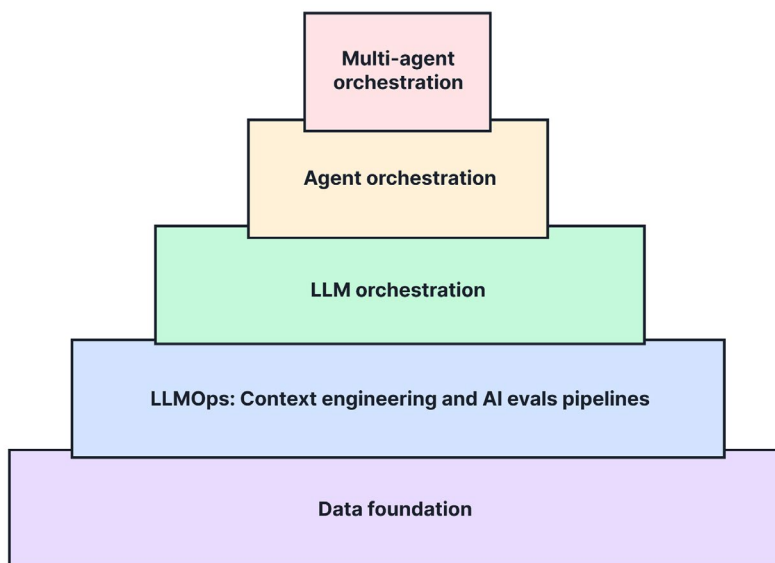


Figure: AI orchestration maturity layers resting on a robust data foundation and LLMOps pipelines.

¹ According to the 2025 MIT NANDA report (https://mlq.ai/media/quarterly_decks/v0.1_State_of_AI_in_Business_2025_Report.pdf)

Apache Airflow® (<https://airflow.apache.org/>) is the industry standard for data orchestration, used by Fortune 500 enterprises and AI startups alike to move and transform data, connecting to any tool in the data ecosystem that has an API, and making sure you and your AI can trust your data.

Moving beyond data orchestration, Airflow is also quickly establishing itself as the orchestrator for data-driven AI, including agents and LLMs. Any time you use AI inside a data pipeline, perform AI orchestration, for any use case, from extracting structured information from unstructured text, to orchestrating complex multi-agent pipelines, you can do so as an Airflow task.

CASE STUDY

Booking.com



Booking.com, one of the world's largest online travel platforms and an Astronomer customer, uses Astro to orchestrate the data and AI foundation behind its public-facing AI travel suite, including the AI Trip Planner, AI Trip Support, AI Voice Support, the AI Rental Helper, and the agentic Smart Messenger and Auto-Reply tools. Its data platform started on-premises on Hadoop and Oozie for batch processing. Early attempts at Airflow, first on-prem and then self-managed in the cloud, didn't deliver the scalability or reliability the team needed, even at a fraction of today's scale. Booking.com standardized on Astro, migrating thousands of Dags in a matter of months as part of one of Europe's largest data platform migrations, and went from days of scheduler downtime per year to near zero.

You can learn more about Booking.com's AI orchestration use case in this case study:

<https://www.astronomer.io/case-studies/booking/>

Orchestrating LLM calls and AI agents as Airflow tasks offers several advantages:

- The Airflow **Common AI provider package** includes a multitude of decorators and operators, such as `@task.llm` and `@task.agent`, built on top of Pydantic AI that make it simple to interact with a wide variety of AI model providers and switch in between them, keeping your AI orchestration pipeline **agnostic** from your model provider.
- Your AI orchestration pipelines can use the **large ecosystem** of Airflow operators (<https://airflow.apache.org/registry/>) with connection information, including credentials and environment variables stored in one place, simplifying governance over access for deterministic and AI tasks alike.
 - Additionally, `@task.agent` tasks can use the `HookToolset` to turn any of hundreds of existing Airflow hooks into a set of tools for the agent to use, using the same Airflow connections used in your operators.
- Airflow has **many options for deciding when a pipeline runs** from regular batch scheduling, to event-driven pipelines, to scheduling AI tasks in a data-driven manner, as soon as pipelines orchestrating necessary context have completed.
- Airflow gives you the **flexibility** to combine tasks running on Airflow worker infrastructure with orchestrating actions in isolated environments, including Kubernetes pods on any cloud or on-prem system. **Remote execution** in Airflow 3 added the ability to run the Airflow workers themselves on any infrastructure, while keeping Airflow core components in a central location.
- You can directly connect the execution of your AI tasks to the upstream context pipelines, as well as use their output in downstream tasks, creating **dependencies** between deterministic and AI-based Airflow tasks.
- Airflow allows you to **automatically retry** tasks in case they encounter transient failures like rate limits. Retries and retry delays can be based on the exception raised, or even decided on by using an LLM with the error and stack trace as context.
- Airflow has a first-class **human-in-the-loop** feature, providing a built-in interface for quality control of AI outputs.

On top of all the advantages of open source Apache Airflow®, Astro adds further capabilities to make your AI orchestration robust. These are just a few of the advantages of running Airflow with Astro:

- **Fully managed deployments:** On Astro, you can create any number of Airflow deployments without needing infrastructure knowledge, using the Astro UI, API, CLI, or the Astro Terraform provider.
- **Worker queues:** The worker queues feature on Astro allows you to define sets of workers of different sizes and assign your tasks to them.
- **Astro environment manager:** On Astro you can store all your connection credentials, Airflow variables, and environment variables in an Astro managed secrets backend and assign them to individual Deployments as needed.
- **Enterprise RBAC + SSO:** Enterprise features that keep your environments secure including role-based access control and audit logging.
- **Otto:** Astronomer's data engineering agent, which can help you answer questions about your Astro environment, write best practice Dags, automatically debug Dags, and migrate Airflow versions. See Otto Overview (<https://www.astronomer.io/docs/astro/otto-overview>) for more information.
- **Astro Observe:** On Astro, you can enable data lineage to trace the inputs to your AI tasks across Dags, define SLAs on complex data products, and configure advanced alerting from a centralized location across Deployments.

You can explore Astro today by starting a free trial: <https://www.astronomer.io/lp/signup/>.

Additionally, Airflow is tool agnostic, which means you can use it with any AI harness, from self-written custom harnesses, to tools like Temporal, LangGraph, CrewAI or Pydantic AI. Airflow even has its own native AI harness, the Common AI provider, built on top of Pydantic AI, which provides `@task.llm` and `@task.agent`, decorators that have a pythonic interface to create AI tasks in your Dags.

You can choose any model, and any harness, the micro orchestration layer for AI agents, and Airflow will be able to handle the big picture: the macro orchestration for your LLMs and AI agents as a larger part of your data ecosystem. Choosing Airflow as the agnostic orchestrator of your AI pipelines means

you are not locked to a specific model provider and can use the most efficient model for your use case, and easily switch between them.

Let's get your AI prototype to production!

AI orchestration is a rapidly evolving field. For this book we made our best effort to use established terms and technologies where they existed. What is more important than individual tools and phrases is the patterns underneath them, which stay relevant across the hype cycles.

AI for Airflow. This book covers how to use Airflow for AI orchestration, but of course you can also use AI to aid you in writing Airflow pipelines. Astronomer's AI agent plugins (Claude Code, Cursor), Astronomer agent skills, the Astro CLI, and Astronomer's own data engineering agent, Otto, all help you write, test, and deploy Airflow pipelines.

For more information on how to use AI to write Airflow Dags see:

- Data engineering with AI Learn guides (<https://www.astronomer.io/docs/learn/local-data-engineering-with-ai-overview>)
- Local data engineering in the agentic era webinar (<https://www.astronomer.io/events/webinars/local-data-engineering-in-the-agentic-era-video>)

This book covers features directly relevant to AI orchestration like `@task.llm`, `@task.agent`, and the `HITLOperator` in detail in later chapters. Depending on your orchestration needs you might also want to download some of our other books:

- AI Context Engineering with Apache Airflow® (<https://www.astronomer.io/ebooks/ai-context-engineering-with-apache-airflow/>): An eBook on preparing data for AI.
- Airflow 3 Dags: The Definitive Guide (<https://www.astronomer.io/ebooks/airflow-3-dags-definitive-guide>): An eBook with broad coverage of Airflow 3 features.
- Apache Airflow 3 Best Practices for ETL and ELT Pipelines (<https://www.astronomer.io/ebooks/apache-airflow-3-best-practices-etl-elt-pipelines>): An eBook similar to this book, which focuses on patterns and best practices for traditional ETL and ELT pipelines.

AI orchestration code snippets. This book combines AI orchestration patterns with select code snippets of relevant features. If you prefer hands-on learning from full Dag examples, check out the AI orchestration examples GitHub repository: (<https://github.com/astronomer/ai-orchestration-examples>).

Definitions

A handful of terms this book uses throughout:

Term	Definition
AI orchestration	AI orchestration is an umbrella term including LLM orchestration, agent orchestration, and multi-agent orchestration.
LLM orchestration	A single inference request to a large language model (LLM) in one Airflow task. The information passed to the LLM includes a system and a user prompt, which can include information that changes at runtime, for example a question submitted by an end user of the pipeline. The response returned by the LLM can be used in downstream tasks.
Agent orchestration	An LLM used in a loop within an Airflow task. The agent can call tools, look at what came back, and decide what to do next, for as many model calls and tool calls as the request takes, with the option to define a limit. Like with LLM orchestration, you can pass information in at runtime as part of the prompt and use the agent's response in downstream tasks.
Multi-agent orchestration	Multiple agents orchestrated with inter-dependency within a pipeline, for example a decider agent routing to specialized downstream agents, or agents cross-checking each other's output.
Harness	Whatever code handles LLM execution and the loop in agent orchestration. Commonly used harnesses include the Common AI provider's decorators (<code>@task.llm</code>, <code>@task.agent</code>), built on Pydantic AI, or a framework like LangChain, LangGraph, Temporal, and CrewAI. You can even write your own harness wrapping the API of a model provider directly. Most pre-built harnesses allow you to work with a wide range of compatible models.
Micro orchestration	The orchestration actions performed by the harness itself, for example tracking token input and output, or preserving tool outputs for durable execution. Micro orchestration is handled by the harness running inside an Airflow task, and you can choose any harness you like.

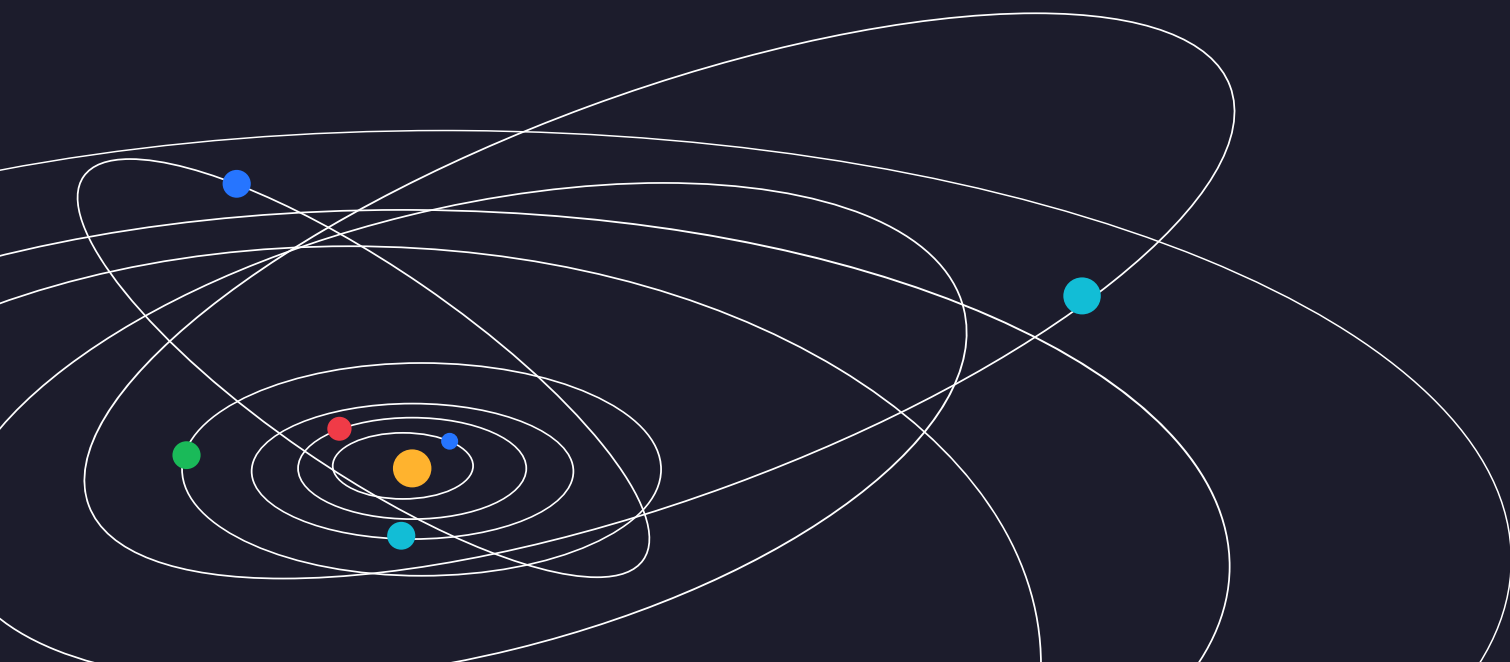
Term	Definition
Macro orchestration	Orchestrating AI (LLMs and agents) as part of your data pipelines. Airflow handles macro orchestration actions like passing in changing prompts at runtime, dynamically creating many AI tasks in parallel, automatically retrying AI tasks in case of transient failures, and processing the AI's output. Macro orchestration also includes governance of the tools and connections given to the AI agent to perform actions in external systems.
Context window	The context window of an LLM is the maximum number of tokens a model can process in a single API request, including the input tokens (system prompt, user prompt, tool results, tool descriptions, etc) and the output tokens (the model response).
Context	The data in the context window for one model API call.
Context engineering	The “delicate art and science of filling the context window [of an AI agent or LLM] with just the right information for the next step” (Karpathy responding to Lütke, June 2025). For more information on context engineering check out our AI Context Engineering with Apache Airflow® eBook. (https://www.astronomer.io/ebooks/ai-context-engineering-with-apache-airflow)
Data-driven LLM/agent	An agent or LLM that is orchestrated as part of a data pipeline and runs on a schedule, in a batch, in a micro-batch, or on demand. Orchestrating these agents includes macro orchestration by Airflow, as well as micro orchestration using a harness.
App-driven LLM/agent	An agent or LLM that runs inside a microservice and answers a single request in real time, for example as part of a chat service. These LLMs and agents typically only use micro orchestration, but the data that they access via tools is prepared by context engineering, and their outputs are evaluated by AI eval pipelines, both orchestrated by Airflow.
AI-based data product (internal)	Any asset that is (partially) generated using an LLM or AI agent which will be utilized by stakeholders internal to your organization. See the <i>Data exploration Agent</i> example in the Agent Orchestration chapter.
AI-based data product (external)	Any asset that is (partially) generated using an LLM or AI agent which will be utilized by stakeholders external to your organization, for example customers or prospects. See the <i>Support ticket agent</i> example in the Agent Orchestration chapter.
Human-in-the-loop	A step in a pipeline that pauses until a person provides input, for example approving, rejecting, or editing an AI-generated output. There are different ways to implement HITL patterns in Airflow. There are human-in-the-loop operators in the Airflow standard provider that create a task waiting for human feedback, and several of the Common AI provider operators and decorators have parameters that add human in the loop functionality as well. For example code see Part 2 of this book.
AI-as-a-judge	A step in a pipeline where an additional LLM or AI agent checks the work of a previous one, for example scoring a drafted reply for accuracy and tone. Sometimes referred to as agent-in-the-loop or AI-in-the-loop. For example code see Part 2 of this book.

PART 001

AI ORCHESTRATION PATTERNS

Over the past decade, data engineers have perfected the Airflow pipeline patterns that build a robust data foundation. As AI emerged as an additional consumer of this data, new patterns, built on top of the familiar ETL and ELT structures, surfaced: LLMOps including context engineering and AI eval pipelines as a subdiscipline of data engineering was created.

With AI orchestration you put LLM and agent calls directly into the Airflow pipelines that already orchestrate the inputs and outputs of AI.



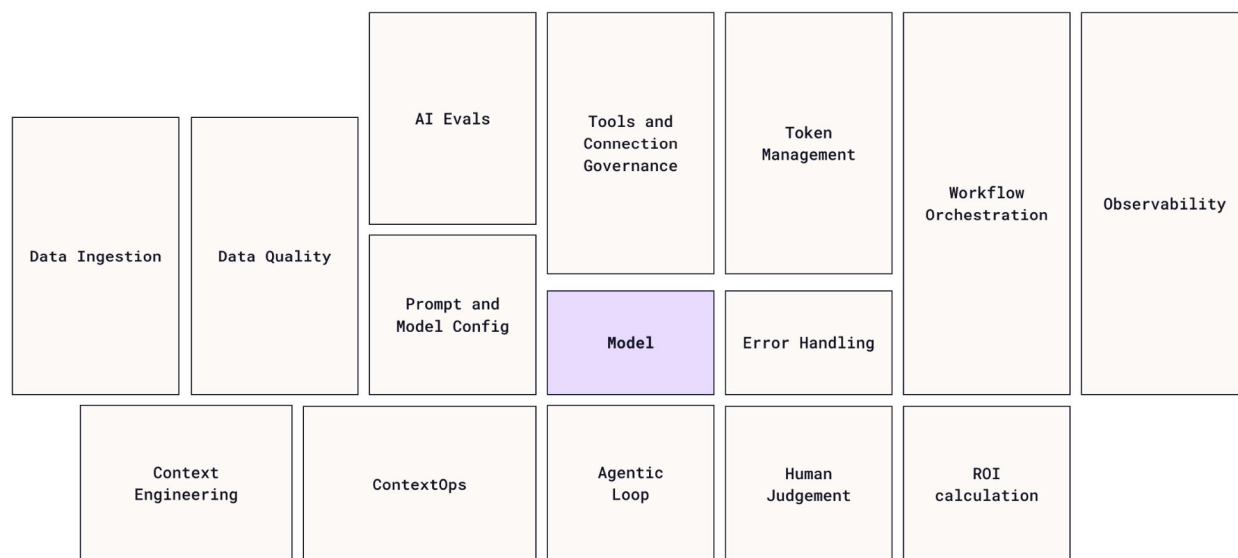


Figure: Visualization of the elements needed in production AI use cases surrounding the AI model. (Adapted and extended from: Hidden Technical Debt in Machine Learning Systems (<https://proceedings.neurips.cc/paper/2015/file/86df7dcfd896fcdf2674f757a2463eba-Paper.pdf> NeurIPS 2015), D. Sculley et al. (Google))

Part 1 of this book covers several LLM and agent orchestration patterns we've seen:

- **LLM orchestration:** When you use Airflow to orchestrate AI inference, many use cases don't need potentially expensive AI agents and can be handled by more efficient LLM tasks. For example, sentiment analysis, extracting structured data from unstructured text, and text-to-SQL applications often only require one LLM inference step if orchestrated in an Airflow Dag.
- **Agent orchestration:** As use cases get more sophisticated and need to interact with other data systems independently via tools, you'll likely move from plain LLMs to putting AI agents into your pipelines, for example to explore data to generate insights or answer customer support tickets.
- **Multi-agent orchestration:** Complex tasks often need structures that involve several specialized agents. Orchestrated with Airflow, these agents can use each other's output and act similar to a team of humans.

Additionally, Part 1 includes LLMOps patterns, the pipelines surrounding the Dags that include AI agents:

- **Context engineering with Airflow:** LLMs and AI agents are only as good as the context (data) they have access to. Airflow already orchestrates your data, additional Dags can extend your ETL/ELT pipelines, preparing data for AI use cases.
- **AI evals with Airflow:** How do you know if your AI orchestration is actually useful? By using Airflow to gather feedback signals, metrics, and data about your AI outcomes, and feeding them back into your data ecosystem.

Lastly, the chapter on **Agent harnesses, micro vs. macro orchestration** shows how Airflow allows you to orchestrate the big picture, functioning as the macro orchestrator, no matter which micro orchestration tool you are using.

These patterns span many types of AI use cases:

- AI creates customer-facing data products, like a report or a recommendation an external user sees.
- AI creates internal-facing data products, the same idea for your own team.
- AI performs business operations directly, acting as assistants and in some cases even as semi-autonomous, tightly monitored, junior employees.
- AI aids other pipelines, an AI step embedded inside an otherwise deterministic pipeline.

Keep in mind that LLMs and AI agents are not deterministic and therefore tasks using these models are not fully idempotent, meaning that the exact output of the AI task will likely differ in a rerun or backfill of the Dag even if the inputs were the exact same.

What you can achieve when orchestrating AI with Airflow is pipelines where each task produces the same end state, even if the exact content differs. For example an AI summarizing an email might return a slightly different summary on a rerun, but it will still return one summary for one email.

These pipelines where each AI task performs as small of an action as possible, is as atomic as possible, are the most robust way to use AI.

Keep all non-AI-based tasks idempotent and, if rerunning or backfilling of AI-based tasks is needed, be aware that outputs can differ.

Almost all AI orchestration pipelines require data to be prepared in patterns that resemble ETL/ELT with some special considerations, see AI Context Engineering with Apache Airflow® (<https://www.astronomer.io/ebooks/ai-context-engineering-with-apache-airflow/>) for a comprehensive introduction to context engineering patterns.

LLM Orchestration

The following sections cover common patterns and use cases for LLM orchestration:

- **LLM summarization:** Condensing long, unstructured text into a shorter form for downstream storage or human review.
- **LLM categorization and routing:** LLMs allow you to categorize unstructured input based on natural language criteria and use the category to decide on the downstream action in the Dag. For example you can categorize issues on a code repository based on likely areas that could be involved in the fix, and use those categories to route the ticket related to the issue to the correct developer team.
 - **LLM sentiment analysis:** A special case of LLM categorization where LLMs score the tone of a piece of text, commonly used to triage reviews, support tickets, or social media mentions before a human sees them.
- **LLM transformation:** LLMs can be used to transform assets, be that in style or tone, format, or even rewriting code in a different programming language. Translating or adapting content into another human language, typically run once per target language, is an example of an LLM use case where the inputs and outputs match in length and general format.
 - **RAG pipelines/Document embedding:** RAG (retrieval augmented generation) pipelines are a special case of transformation pipeline where instead of an LLM an embedding model is used to turn chunks of text into embedding vectors which can be retrieved based on vector similarity search. This pattern is widely used to find matches that regular keyword search would miss, for example to suggest products similar to a purchase based on a product description. See the Context Engineering chapter for an example for a RAG pipeline.

- **LLM extraction of structured information from unstructured data:** Pulling specific fields out of free-form text, like a contract or a scanned document, to load into a database or use as features for traditional ML.
- **LLM-based task retries:** With Airflow you can use an LLM to decide whether or not to retry a task based on the task logs containing the error message. This pattern is distinct as the call to the LLM is not made within an Airflow task, but as part of the retry mechanism, after the execution of any task has failed.
- **LLM asset generation:** Using an LLM to create new content, like a draft report or an email reply.
 - **Text-to-SQL:** A special case of asset generation is to create text that is used inside the same pipeline, for example code that is executed. This pattern poses unique risks and requires additional consideration for guardrails. A common example is to convert a natural language question into a SQL query and execute it within the same Dag.

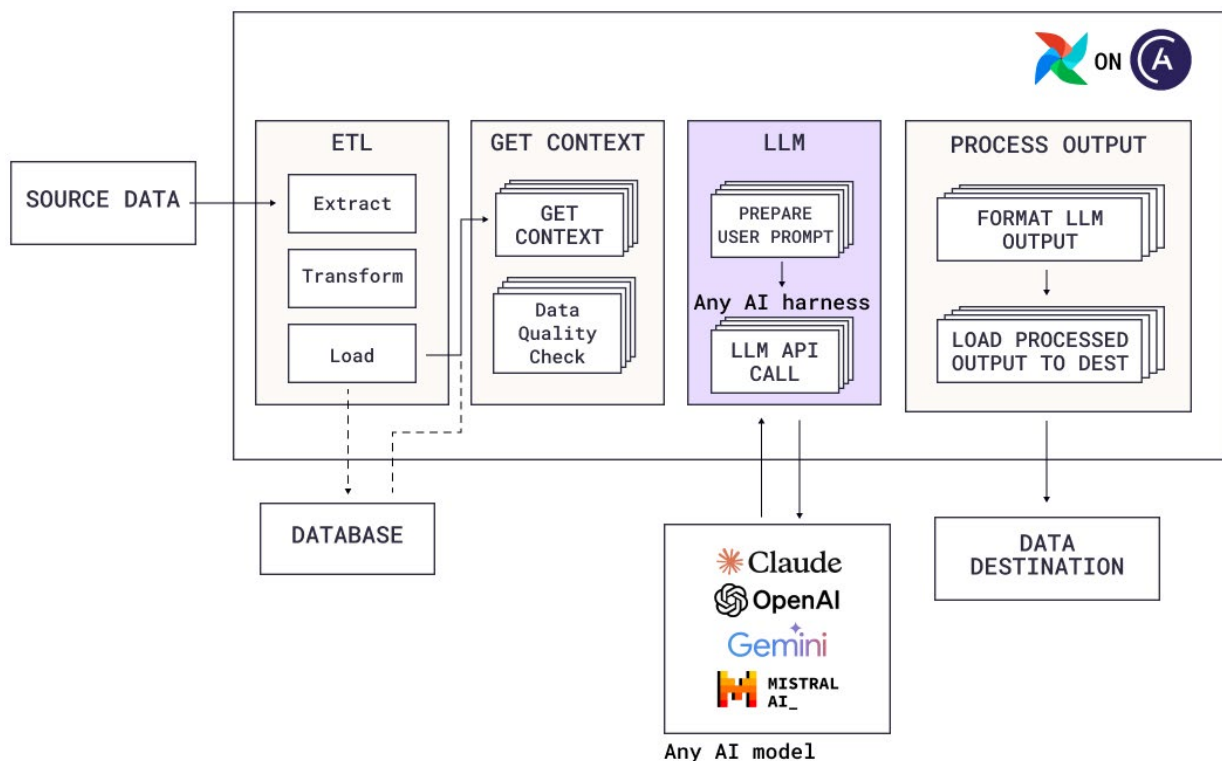


Figure: Architecture diagram of the general structure of an LLM orchestration pipeline.

Orchestrating LLMs as part of data pipelines generally follows this basic structure:

0. Source data is extracted, transformed, and loaded into a destination for use by the LLM orchestration pipeline. Sometimes this data is stored (semi-)permanently in a database, sometimes passed directly to downstream tasks.
1. Data is extracted from a database or upstream pipeline, and formatted into context for the LLM. Additional deterministic steps, for example data quality checks, are performed.
2. The prompt is formatted and passed to the LLM using any type of AI harness, for example `@task.llm` of the Airflow Common AI provider package. The harness completes one API call to any compatible model, for example Claude Sonnet 5, and returns the output of the model. There are two types of prompts that are combined in the model API call request:
 - a. The system prompt, which contains general instructions about the task and the desired outcome, and in the case of AI agents lists available tools. The system prompt only changes when the code is updated and is commonly versioned through a version tag in the system prompt string.
 - b. The user prompt, which contains any data that changes for each asset, for example the customer email that the LLM should summarize. Additional context retrieved by upstream tasks, for example data about that customer's purchase history is passed with the user prompt.
3. The model output is available to downstream tasks in the pipeline. What happens at this step varies by use case, but usually involves processing of the response as well as loading it to a data destination.

The easiest way to get started with LLM orchestration is to use the Common AI provider's `@task.llm` decorator in your Airflow Dag. The code snippet below shows the general structure of this decorator, including how LLM inputs and outputs are handled. See the Common AI provider chapter in Part 2 of this book for more information.

```

from pendulum import duration
from airflow.sdk import task
from pydantic_ai.usage import UsageLimits
from pydantic import BaseModel
from typing import Literal

class MyOutput(BaseModel):
    information_density: Literal["low", "medium", "high"]
    summary: str

@task.llm(
    llm_conn_id="pydanticai_default", # required: your Airflow connection ID
    # Optional parameters
    model_id="<your_provider>:<your_model>", # Override the model from the conn
    system_prompt="You are a helpful AI.", # General instructions
    output_type=MyOutput, # structure the output with a pydantic model
    usage_limits=UsageLimits(total_tokens_limit=4000), # cost control by default
    request_limit is set to 50 by pydantic-ai
    agent_params={}, # passed to the Pydantic AI Agent constructor
    require_approval=False, # if set to True the task waits for human approve/
    reject of the output
    approval_timeout=duration(minutes=10), # default: None; max wait for approval
    allow_modifications=False, # default: False; if set to True the reviewer may
    edit output before approve
)
def my_llm_task(user_context: str) -> str:
    user_prompt = f"Summarize this document: {user_context}" # any transformation
    return user_prompt

my_llm_task(user_context=my_document) # User context, often prepared by upstream
tasks is passed to the LLM task as arguments, the LLM response is pushed to XCom in
the format defined by output_type

```

LLM summarization

A common first use case for LLM orchestration is to summarize long texts to surface their most important facts in an executive report or aid in deciding which documents warrant a more detailed human read.

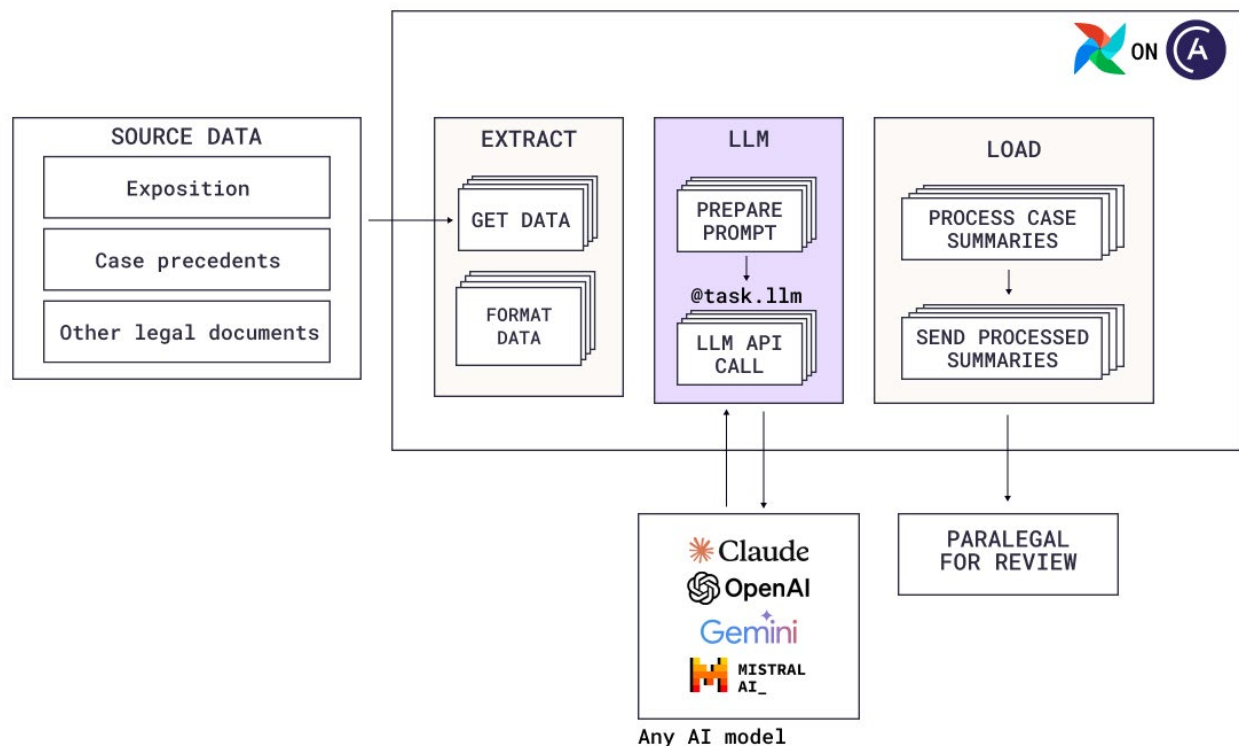


Figure: A simple architecture of LLM summarization.

The architecture diagram shows a possible pipeline to summarize extensive legal documents to assist paralegals in the discovery phase. This is a type of pipeline that is run ad hoc as soon as a new case needs to be processed and all documents are available.

1. The pipeline ingests several long-form dense documents and formats the data to be passed to the AI as part of the user prompt.
2. The LLM summarizes the text and surfaces important information using a structured output class like the one shown in the code snippet below.
3. The summaries are processed and sent to the paralegal for review, aiding in the decision of which documents to read manually and include in the case.

Because many documents are processed for a case, all tasks are dynamically mapped (see the Dynamic Task Mapping chapter in Part 2 of this book) over the list of input documents, automatically creating one summary output per input document.

The structured output defines the schema of the LLM's generated text. When using the `@task.llm` decorator of the Common AI provider you can provide an output defined as a Pydantic BaseModel to the `output_type` parameter.

```
class LegalLLMSummary(BaseModel):
    document_type: Literal[
        "contract", "correspondence", "court_filing",
        "deposition_transcript", "discovery_document", "other",
    ]
    summary: str
    parties: list[str] = Field(
        description="Named individuals or entities involved in the document"
    )
    key_dates: list[str] = Field(
        description="Deadlines, effective dates, or hearing dates, e.g. 'Response due: 2026-08-01'"
    )
    flagged_clauses: list[str] = Field(
        default_factory=list,
        description="Provisions that may need attorney review, e.g. indemnification, non-compete, arbitration",
    )
    source_references: list[str] = Field(
        description="Page or section citations backing the summary above, e.g. 'p.4 §2.3'"
    )
    requires_attorney_review: bool = Field(
        description="True if the document contains ambiguous language, unusual terms, or high monetary exposure"
    )
    potentially_privileged: bool = Field(
        description="True if the document may be covered by attorney-client privilege or work product"
    )
```

This pattern uses a single LLM call with the `@task.llm` decorator as a harness, the code handling the interaction with the API of your LLM model provider. The `@task.llm` decorator prepares the input to the API call based on your `system_prompt` and the value returned from the decorated function, which turns into your user prompt, alongside any other information such as `usage_limits` or `agent_params` you provided.

Using an LLM over an agent is a reasonable economic choice for this use case because summarization doesn't need tools or multi-step agentic reasoning: one document goes in, one structured summary comes out. That also makes it cheaper and faster to run at the volume a discovery phase requires. If the source text's length exceeds the LLM's context window, you'll likely want to add a chunking step as part of pre-LLM context engineering.

In this example, the human review happens outside the pipeline, because all subsequent steps are done by humans. In other types of summarization pipelines, human review is often part of the pipeline itself using a human-in-the-loop operator, and downstream actions happen automatically based on the human review response.

LLM categorization and routing

How many unread emails are currently in your inbox? LLM categorization and routing might be able to help with that!

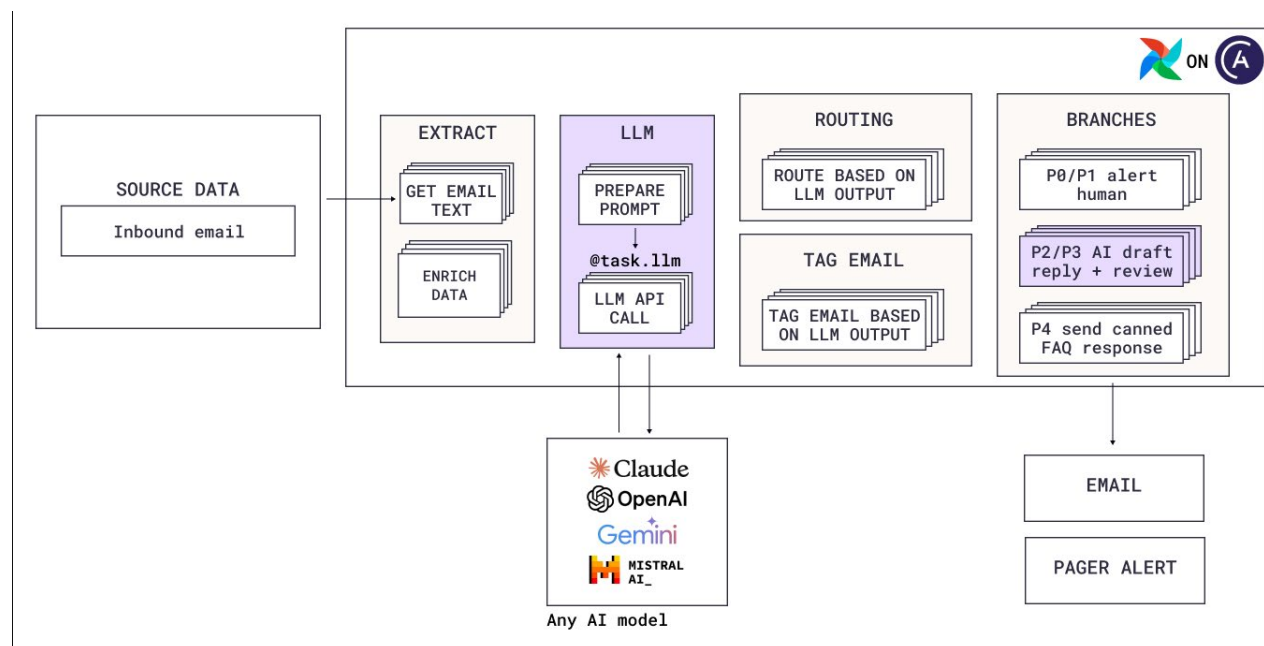


Figure: Example architecture using an LLM to categorize emails by topic and priority, using the priority category for downstream routing.

This pattern, which can run on a schedule, for example batch processing all new emails once an hour, pairs a single LLM call asking for email categorization with two independent downstream actions built on two different classifications. The topic tag is used to categorize the email to aid in later searches. The priority tag is used to route downstream actions. P0 and P1 emails alert a human directly, P2/P3 emails get an AI-drafted reply that still needs review, and routine P4 questions get a canned FAQ response with no human involved at all.

1. The first task extracts the email text and enriches it with any additional context the classifier needs, for example previous conversations or account information.
2. The LLM classifies the email by priority and topic tags in a single call, using a structured output class the same way LLM summarization does.
3. Two tasks use different fields in the classification independently: one routes to exactly one response path based on email priority, and the other applies the topic tags.

Because inbound email arrives in batches, all of this, classifying, routing, and tagging, is dynamically mapped over the list of new messages, handling each one independently. The code snippet below shows the dynamic mapping of a task group over the list of emails (`process_email.expand(email_text=_fetch_email)`), and how `@task.branch` is used to pick a downstream branch in the Airflow Dag.

```
from __future__ import annotations
from datetime import datetime
from typing import Literal
from pydantic import BaseModel, Field
from airflow.sdk import chain, dag, task, task_group
from include.email_server import (
    EMAIL_CLASSIFICATION_SYSTEM_PROMPT,
    fetch_emails_from_server,
)

class EmailClassification(BaseModel):
    priority: Literal["P0", "P1", "P2", "P3", "P4"] = Field(
        description="Incident-style severity, P0 most urgent, P4 least"
    )

@dag
def email_routing_dag():
    @task
    def fetch_email() -> list[str]:
        return fetch_emails_from_server()

    @task_group
    def process_email(email_text: str):
        @task.llm(
            llm_conn_id="pydanticai_default",
            system_prompt=EMAIL_CLASSIFICATION_SYSTEM_PROMPT,
            output_type=EmailClassification,
        )
        def classify_email(email_text: str) -> str:
            return f"Classify this email:\n\n{email_text}"
```

```

@task.branch
def route_by_priority(classification: EmailClassification) -> str:
    if classification.priority in ("P0", "P1"):
        return "process_email.alert_human"
    if classification.priority in ("P2", "P3"):
        return "process_email.draft_and_review"
    return "process_email.send_faq_response"

@task
def alert_human(email_text: str) -> None: ...

@task
def draft_and_review(email_text: str) -> None: ...

@task
def send_faq_response(email_text: str) -> None: ...
    _classification = classify_email(email_text)
    _route = route_by_priority(_classification)

chain(
    _route,
    [
        alert_human(email_text),
        draft_and_review(email_text),
        send_faq_response(email_text),
    ],
)
_fetch_email = fetch_email()
process_email.expand(email_text=_fetch_email)

email_routing_dag()

```

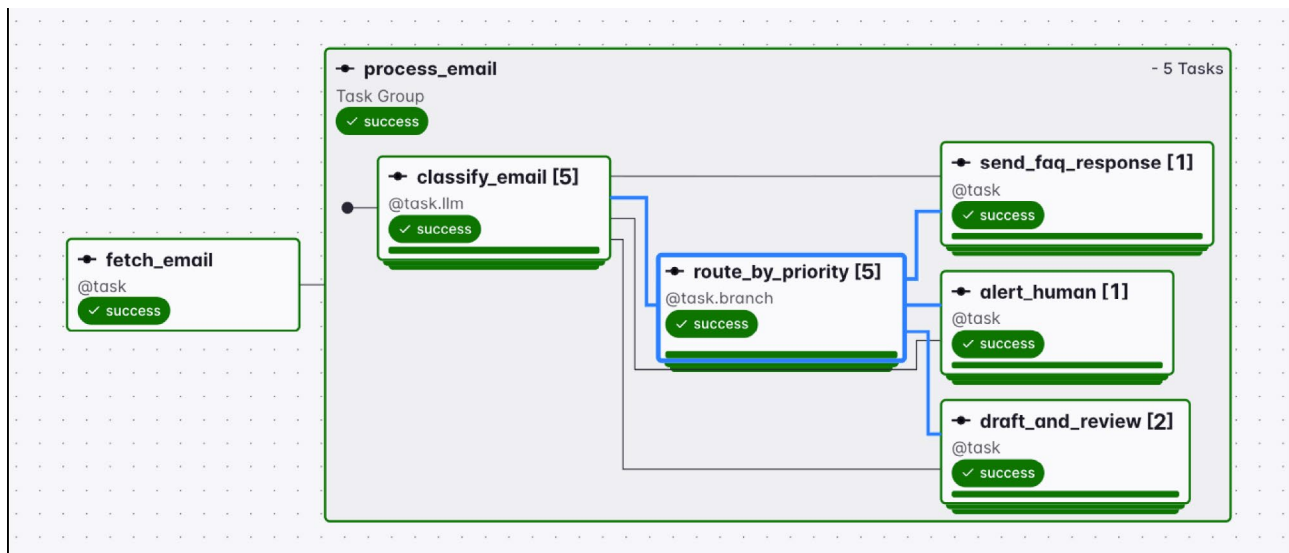


Figure: Screenshot of the Dag dynamically mapped over the list of emails classifying the emails by priority and routing them accordingly.

You also have the option to let the LLM directly pick a branch by using the `@task.llm_branch` decorator, as shown in the example code below. This is a common pattern when using a classification only for routing, as opposed to our more complex example where the LLM classified the email on two axes: priority and tags, and both outputs were used downstream in different tasks.

```

from airflow.sdk import dag, task, chain
from include.fixtures import EMAIL_CRITERIA

@dag
def example_llm_branch():
    @task
    def fetch_email():
        from include.func import get_email
        email = get_email()
        return email

    @task.llm_branch(
        llm_conn_id="pydanticai_default",
        system_prompt=f"Pick a downstream branch depending on email priority according to {EMAIL_CRITERIA}"
    )
    def route_email(email: str) -> str:
        return f"Email: {email}"
  
```

```

@task
def handle_p0_p1(): ...

@task
def handle_p2_p3(): ...

@task
def handle_p4(): ...

chain(
    route_email(fetch_email()),
    [
        handle_p0_p1(),
        handle_p2_p3(),
        handle_p4(),
    ]
)

example_llm_branch()

```

The case of drafting an AI reply for P2/P3 emails is a second, separate LLM call, an example of LLM asset generation and how LLM orchestration patterns can be combined and nested.

This is the first pattern where an external-facing AI-generated asset is created. Always be mindful when sending out AI-generated text, as it can contain errors, be written in an inappropriate tone, or disclose any sensitive information that was loaded into the context, whether on purpose or by accident.

LLM sentiment analysis

Using LLMs for sentiment analysis follows the same pattern as any other LLM-based task: extract unstructured text, format it, call the model, get a score back. But a sentiment score is rarely useful on its own. It starts to be actionable once it's combined with other metrics about a customer, for example product usage data, ticket volume, and account history.

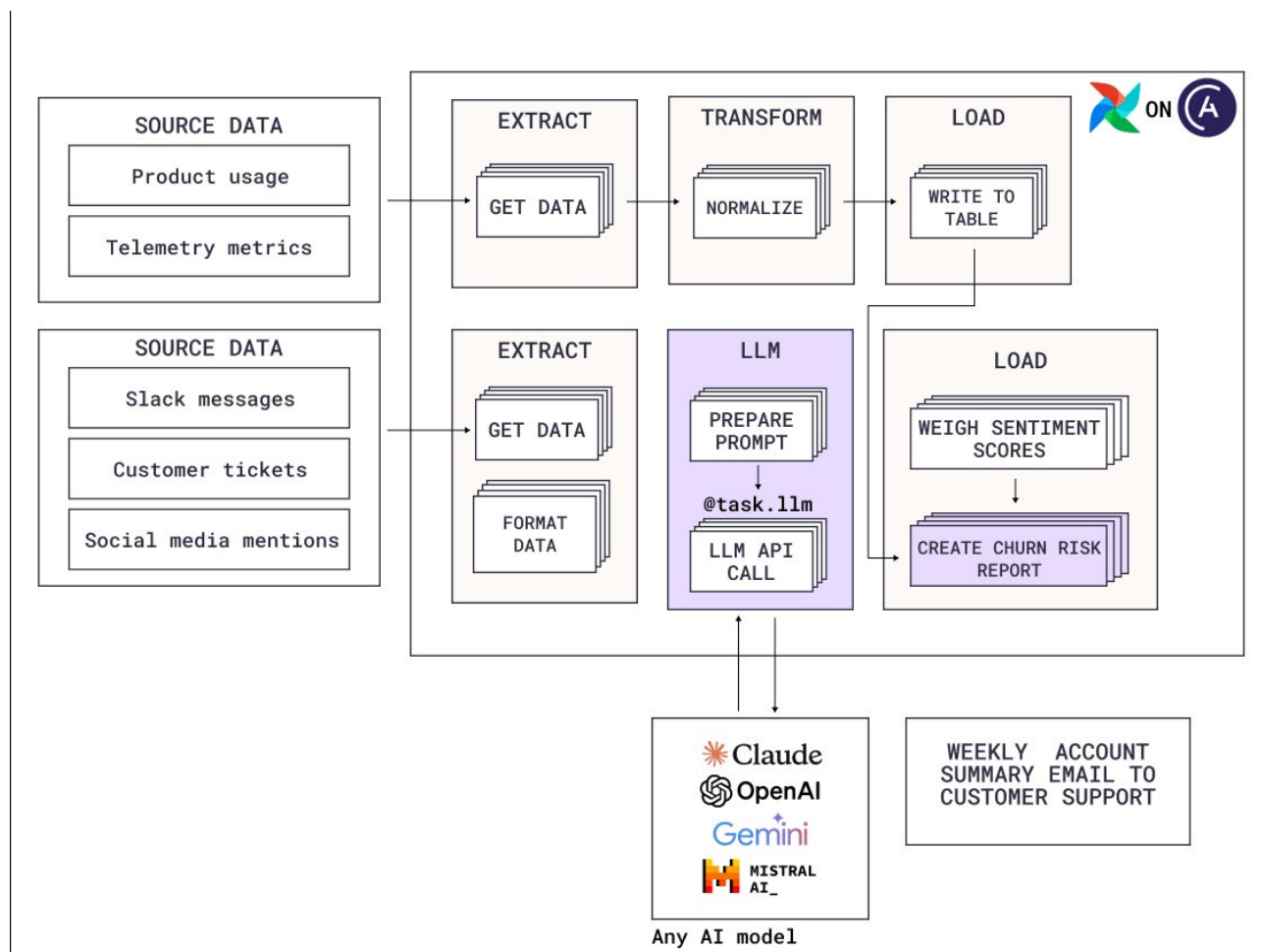


Figure: Architecture diagram for a sentiment analysis pipeline that combines a deterministic ETL Dag with AI orchestration to create a churn risk report.

The diagram shows two independent pipelines feeding into one downstream report. A deterministic ETL pipeline extracts product usage and telemetry metrics, normalizes them, and writes them to a table. A separate pipeline extracts unstructured customer signals (Slack messages, support tickets, social media mentions), scores each one's sentiment with an LLM, then deterministically weights and

aggregates those individual scores into a single number per customer. Once both pipelines are done, a second LLM call combines that weighted sentiment score with the deterministic metrics into a churn risk report, which goes out to customer support in a weekly summary email.

To schedule the report generation once both upstream pipelines have finished, Airflow offers conditional asset scheduling. The deterministic Dag creates an asset event once it writes its results to the target table. The AI Dag creates an asset event once its sentiment scores are ready.

```
from airflow.sdk import Asset, task
# in the deterministic Dag
@task(outlets=[Asset("customer_metrics")])
def load_customer_metrics():
    pass # your logic here
load_customer_metrics()

# in the AI Orchestration Dag
@task(outlets=[Asset("sentiment_scores")])
def weight_sentiment_scores():
    pass # your logic here
weight_sentiment_scores()
```

The report Dag is scheduled to run only once both assets have been updated. This is an example of a very simple conditional asset schedule. You can freely combine & (AND) and | (OR) conditions when scheduling a Dag on tens or even hundreds of asset events. You can also partition your asset events for more fine-grained control.

```
from airflow.sdk import Asset, dag
@dag(
    schedule=(
        Asset("customer_metrics") & Asset("sentiment_scores")
    ),
)
def create_sentiment_report():
    pass # your tasks here
create_sentiment_report()
```

You can learn more about assets in Airflow in the following guides:

- Basic asset-based scheduling in Apache Airflow® (<https://www.astronomer.io/docs/learn/airflow-datasets>)
- Advanced asset-based scheduling in Apache Airflow® (<https://www.astronomer.io/docs/learn/airflow-advanced-asset-scheduling>)
- Partitioned Dag runs and asset events in Apache Airflow® (<https://www.astronomer.io/docs/learn/airflow-partitioned-runs>)

LLM transformation

LLMs can be used to transform unstructured data according to rules. Some of the most common transformations include:

- **Converting between formats:** For example .txt to Markdown.
- **Translating between programming languages:** For example rewriting legacy code from COBOL to Java, or legacy workflows from Control-M to Airflow Dags.
- **Translating between human languages:** This is one of the oldest use cases for LLMs. Translation and localization can give you an advantage when selling in global markets, and can even be legally required, for example for product safety information.

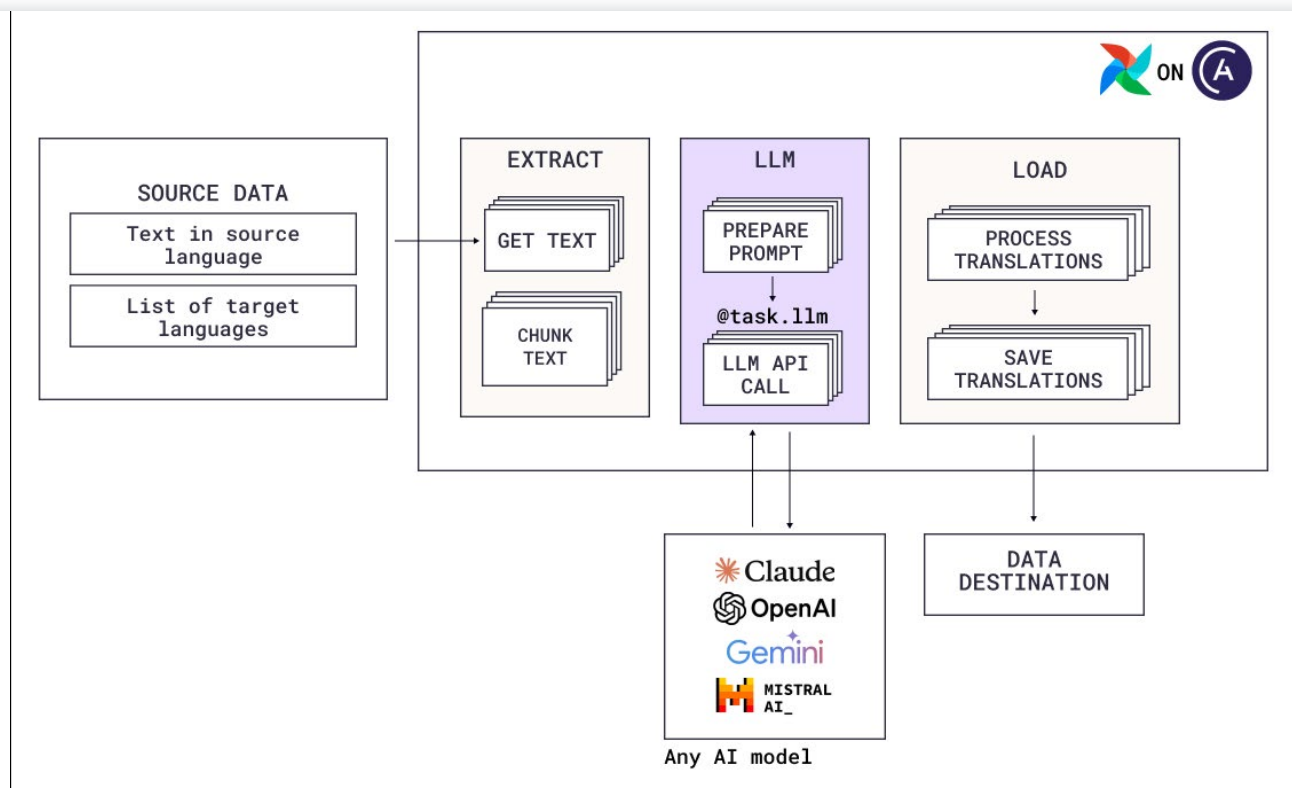


Figure. Example architecture for LLM translation.

The diagram shows a pipeline that fetches a piece of source content, for example a help center article, and translates it into several languages. Input here comes in two lists: the text itself, and the list of target languages it needs to exist in.

1. The pipeline gets the source text and chunks it into pieces small enough to fit comfortably inside one model call. The list of target languages is a separate, simpler input, used later for mapping.
2. The LLM translates one chunk into one target language per call, using whatever harness you've set up.
3. The translated chunks are reassembled in order and saved to their destination, for example back into your documentation system.

This pattern is dynamically mapped across two lists at once instead of one: every chunk gets translated into every target language, so a ten-chunk article going into eight languages creates eighty parallel LLM calls.

```
from airflow.sdk import task

@task.llm(
    llm_conn_id="pydanticai_default",
    system_prompt="""Translate the given text into the target language. Preserve
formatting, tone, and any technical terminology exactly. Return only the translated
text."""
)
def translate_chunk(text_chunk: str, target_language: str) -> str:
    return f"Translate the following text into {target_language}:\n\n{text_chunk}"

translate_chunk.expand(text_chunk=list_of_chunks, target_language=list_of_
languages) # creating a cross product with one task instance per chunk+language
combination
```

The main risk with chunking translation is consistency: a term translated one way in chunk three and a different way in chunk seven could lead to confusion. This is why for localization efforts, you usually create a separate context engineering pipeline that extracts common terms from documents and creates one canonical translation per term and language. These terms, assembled by language, are given to the translating LLMs as additional reference context.

If you'd alternatively like to have a separate Dag run per target language, you can schedule the Dag on asset partitions using a segment partition key. For more information see the Partitioned Dag runs and asset events in Apache Airflow® guide (<https://www.astronomer.io/docs/learn/airflow-partitioned-runs>).

LLM extraction of structured information from unstructured data

Extracting structured fields from free-form text is a similar pattern to the sentiment analysis and summarization use cases, in particular relying on a well-defined output class (`output_type`). The difference lies in what happens to the extracted data fields after the LLM call. For example, they get loaded into a database, or turned into features for a traditional ML model. The AI step exists in service of a more traditional ETL or MLOps pipeline.

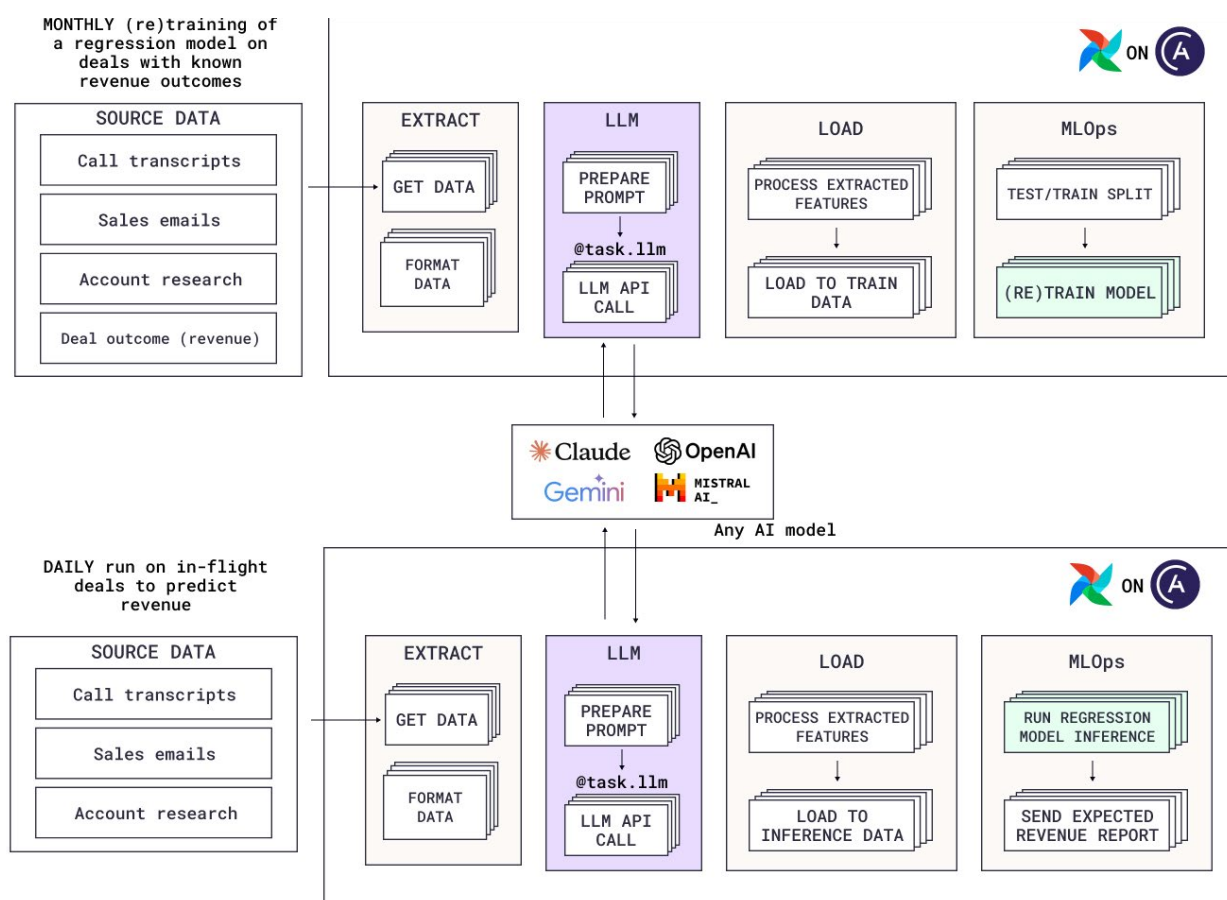


Figure: Architecture diagram showing a training and batch inference pipeline for a traditional regression model that uses AI features as input.

The diagram shows an MLOps pipeline for a sales team to predict expected revenue on open deals using a traditional regression model. A monthly Dag extracts structured signals (objections raised,

competitors mentioned, stated timeline) from call transcripts, sales emails, and account research on deals with a known outcome, and uses them to (re)train and test the model. A daily Dag runs the exact same extraction on in-flight deals, whose outcome isn't known yet, and feeds the result into that model to predict expected revenue.

- Both Dags extract the same unstructured sources and format them the same way, reusing modularized code.
- Both Dags run the same LLM call with the same structured output class to pull out deal signals like objections, competitors mentioned, and stated timeline.
- From there the Dags diverge: the monthly Dag adds the known deal outcome and retrains the model, the inference Dag creates a prediction and sends out a projected revenue report to sales leaders.

Following the best practice of modularization and having one source of truth, any code that is shared between the training and inference Dags gets stored in a module and from there imported into both Dags.

```
# include/extraction.py is shared across both Dags
from airflow.sdk import task
from pydantic import BaseModel, Field

class DealFeatures(BaseModel):
    objections_raised: list[str] = Field(
        description="Objections the prospect raised, e.g. ['price', 'timeline']"
    )
    competitors_mentioned: list[str] = Field(
        description="Competitor names mentioned in calls or emails"
    )
    stated_timeline: str = Field(
        description="The prospect's stated decision timeline, in their own words"
    )
    budget_confirmed: bool = Field(
        description="True if the prospect explicitly confirmed budget"
    )
```

```

@task.llm(
    llm_conn_id="pydanticai_default",
    system_prompt="Extract structured deal signals from sales call transcripts,
emails, and account research.",
    output_type=DealFeatures,
)
def extract_deal_features(deal_context: str) -> str:
    return f"Extract deal signals from the following:\n\n{deal_context}"

# dags/train_revenue_model.py
from include.extraction import extract_deal_features
features = extract_deal_features.override(task_id="extract_train_features")(deal_
context=deal_context) # you can override task parameters using the .override
method

# dags/predict_revenue.py
from include.extraction import extract_deal_features
features = extract_deal_features.override(task_id="extract_inference_features")
(deal_context=deal_context)

```

Both Dags import `extract_deal_features` and call it. The only difference is what happens to the result afterward.

This pattern of course can be combined with others, for example including categorization or sentiment analysis as ML features. It is also common practice to combine deterministic features with AI-generated features.

Remember that any features generated by an LLM aren't deterministic. If you ever backfill this pipeline, extracting features from the same call transcript twice isn't guaranteed to produce identical feature values, which changes your training data. Mark AI-derived features as such, for example with a naming convention `ai_` prefix.

CASE STUDY

popety.io



Popety.io, a Swiss PropTech platform and Astronomer customer, uses a similar pattern to extract complex information necessary to determine building rules for development plots (for example: zoning rules, building heights, floor limits) from municipal and cantonal legal documents across 26 Swiss cantons and more than 2,000 municipalities. Reading these regulations page by page used to be manual work, and covering the full corpus by hand would have run to more than 40,000 hours. Popety has since automated this process by running LLM extraction inside Airflow Dags on the same Astro platform that already orchestrates their data.

Because many of the source documents are scanned PDFs, their pipeline adds an OCR step with a vision-language model before extraction, then searches the resulting text for the relevant passages, combining keyword and semantic search. An LLM, called through LangChain as the harness, extracts the structured zoning fields from those passages. The pipelines run over 7,000 LLM extraction tasks a month, with every extracted value traceable back to its source document.

You can learn more about Popety.io's LLM orchestration use case in this case study: <https://www.astronomer.io/case-studies/popety/>

LLM-based task retries

At large scales, it is likely that some task instances will run into transient failures like rate limits or model API outages. Airflow has had the capability to retry task instances with a defined `retry_delay` and optional `retry_exponential_backoff` for many years, but not all failures need the same response. An authentication issue due to expired credentials won't be fixed by waiting the way a rate limit error will be. This is why Airflow 3.3 adds pluggable retry policies where you can define what happens for a list of errors.

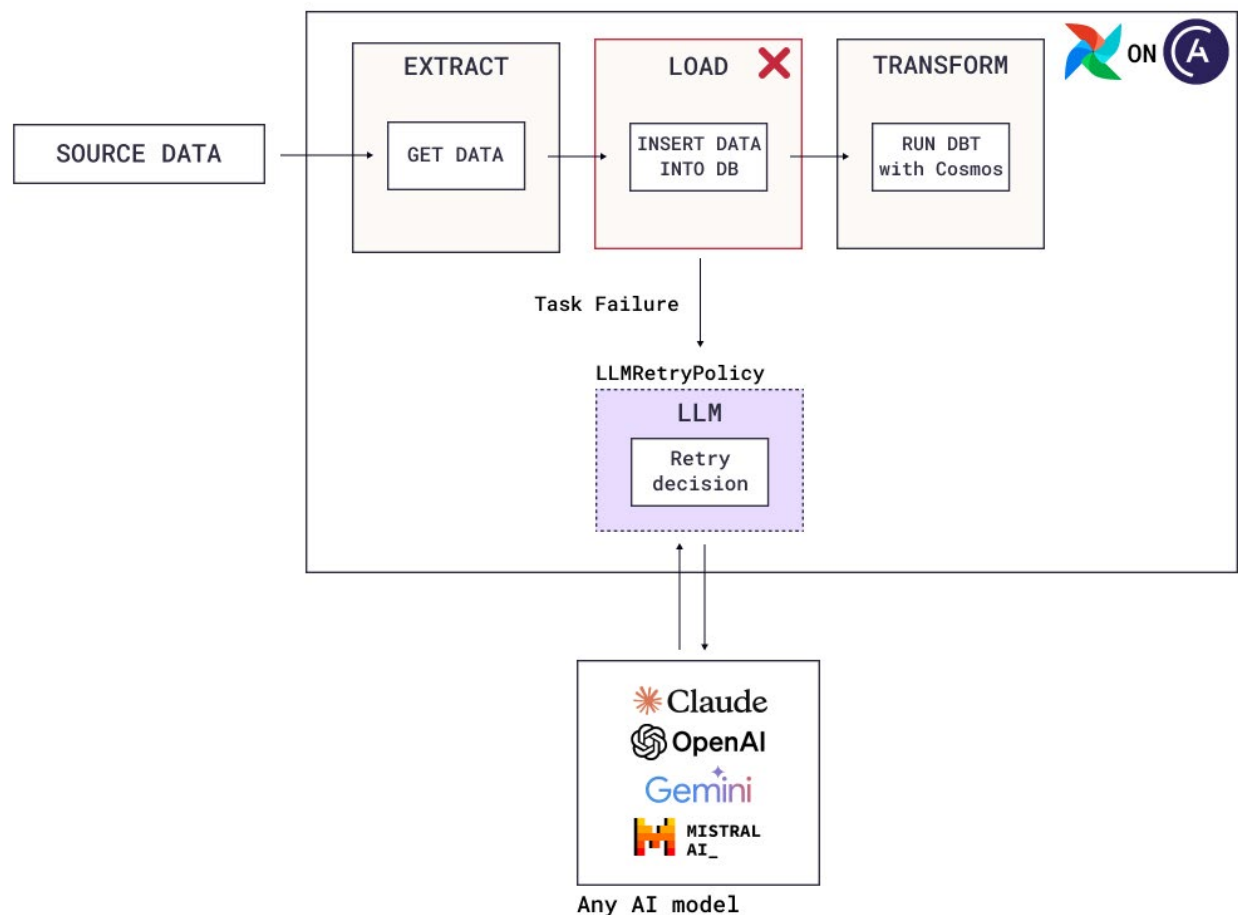


Figure: Architecture diagram showing an `LLMRetryPolicy` deciding whether or not to retry a task that has failed its previous attempt.

You can even prepare for errors that you have not anticipated with the Common AI provider's `LLMRetryPolicy`: when a task fails, an LLM reads the exception and decides whether to retry, how long to wait, and records its reasoning.

```

from airflow.sdk import dag, task
from pendulum import duration
from airflow.providers.common.ai.policies.retry import LLMRetryPolicy
from airflow.sdk.definitions.retry_policy import RetryAction, RetryRule
llm_policy = LLMRetryPolicy(
    llm_conn_id="pydanticai_default",
    timeout=30.0, # max seconds to wait for LLM response
    fallback_rules=[ # used when LLM call fails
        RetryRule(exception=ConnectionError, action=RetryAction.RETRY, retry_
delay=duration(seconds=10)),
        RetryRule(exception=PermissionError, action=RetryAction.FAIL),
    ],
)
# the AI chooses the delay for example hitting a rate limit:
# LLM error classification: category=rate_limit, should_retry=True, retry_delay=60,
reasoning=The error message says 'Try again in 48s', implying that the API has
throttled usage due to exceeding allowed requests and that it will be available
again in under a minute.
@dag
def pluggable_retries_llm():
    # pluggable retries can be added to regular tasks
    @task(
        retries=5,
        retry_delay=duration(seconds=5),
        retry_policy=llm_policy
    )
    def my_task(**context):
        pass # your logic

    my_task()
    # pluggable retries can be added to @task.agent and @task.llm tasks
    @task.llm(
        llm_conn_id="pydanticai_default",
        system_prompt="My system prompt",
        retries=5,
        retry_delay=duration(seconds=5),
        retry_policy=llm_policy
    )
    def my_llm_task(**context):
        return "My user prompt"

    my_llm_task()
pluggable_retries_llm()

```

In the example code above, if either of the tasks fails, the exception is sent to an LLM which decides whether to retry the task, and if so, how long the wait time before the next retry should be. If the LLM call itself fails, the `fallback_rules` are used: a `ConnectionError` retries after 10 seconds and a `PermissionError` fails the task immediately. If neither rule fits, the task falls back to its regular retry behavior defined by `retries` and `retry_delay` on the task. If the task has used up all its retries set on the `retries` parameter, it fails without making another call to the LLM. You can pass additional information to the LLM making the retry decision with the `instructions` parameter of the `LLMRetryPolicy` class.

LLM asset generation

You can ask LLMs to generate any kind of text, from prose to yaml to code, and more generally LMMs (large multimodal models) to create any type of asset, from internal-facing AI data products like executive summaries, to external-facing data products like personalized product recommendations based on a customer's purchase history.

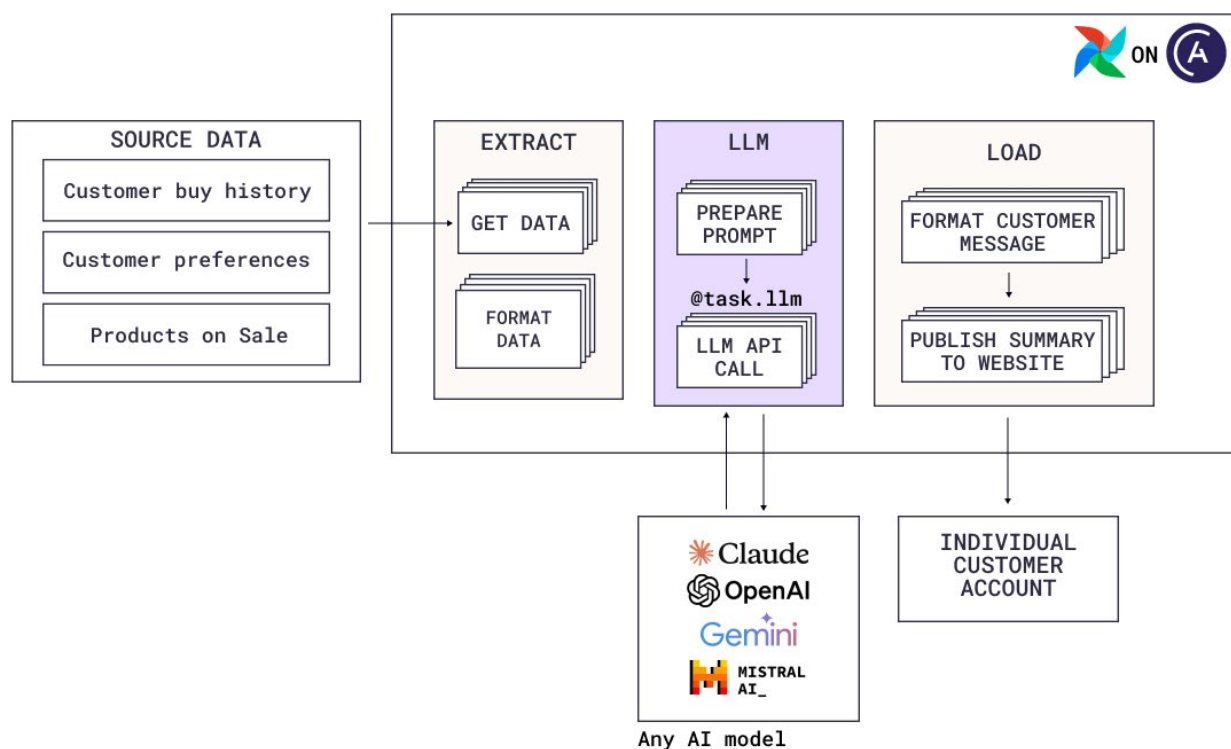


Figure: Example architecture to create assets as a customer-facing AI data product.

The diagram shows a daily pipeline that runs once per customer: pull their purchase history, stated preferences, and the current sale catalog, generate a short recommendation message, format it, and publish it to their account dashboard.

This pipeline is again dynamically mapped over your entire customer base, one task instance per customer, once a day. That's also where this pattern runs into a potential limit as your organization scales: dynamic task mapping caps out at 1024 mapped task instances per task by default, controlled by the `max_map_length` config (<https://airflow.apache.org/docs/apache-airflow/stable/configurations-ref.html#max-map-length>).

Raising `max_map_length` is an option for a couple thousand parallel tasks, but beyond that we recommend partitioning your Dag runs by customer segment, especially if you already have distinct customer clusters, for example high-value accounts, new signups, or customers at risk of churn, where you might want to adjust the format and prompt for each cluster.

Text-to-SQL

Text-to-SQL is a specific case of asset generation where the LLM turns a natural language question into a runnable SQL query, so a stakeholder who can't write SQL can still ask a direct question of the data.

Executing AI-generated SQL needs proper guardrails. There is always the possibility of the AI making a mistake or the risk of (accidental) prompt injection, for example the stakeholder copying a data request that includes an unrelated statement about deleting something.

The only safe guardrail for AI lies at the level of permissions. The connection used to execute the SQL query needs to be scoped to the permissions of the stakeholder, typically read-only permissions on selected tables and records.

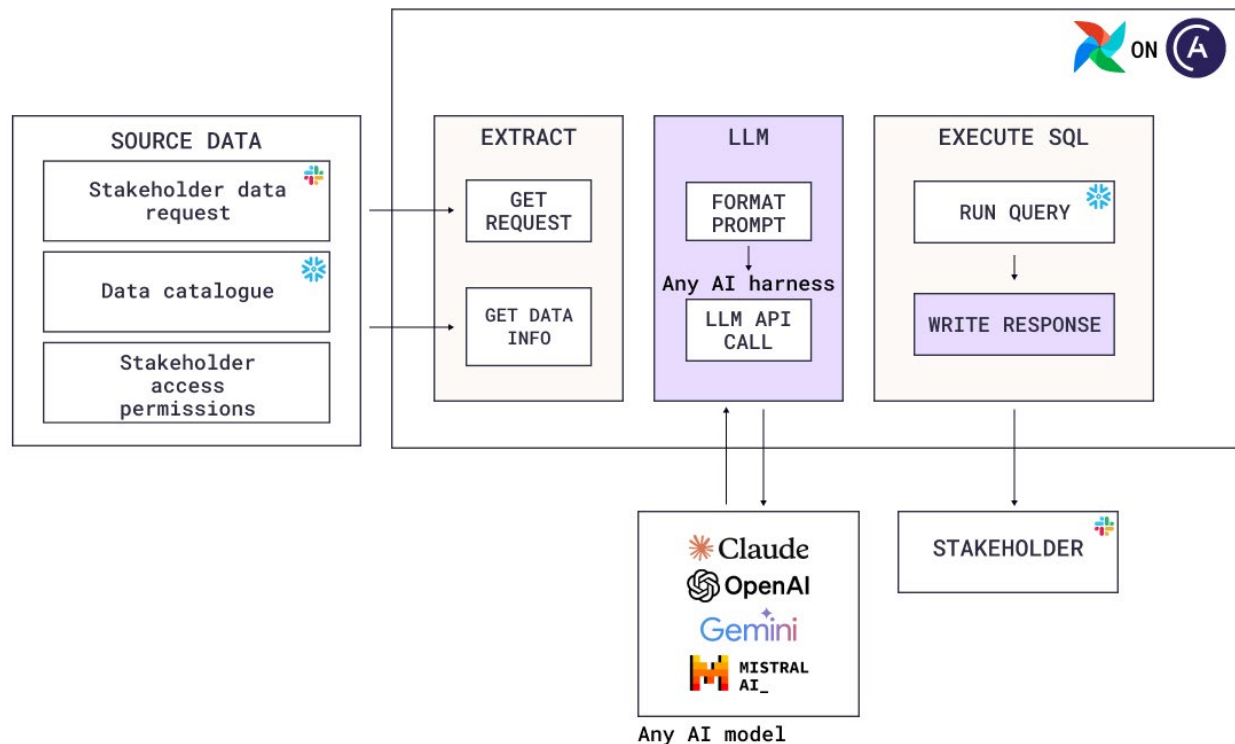


Figure: Example of a simple text-to-SQL pipeline using an LLM task to create the query, as well as write up the response with the query results.

The diagram shows a stakeholder asking a data question directly in Slack. The pipeline pulls in the request, has an LLM write the SQL, then executes the SQL using a traditional operator. The query runs, and the result gets turned back into a plain-language answer sent back to the stakeholder in Slack.

- Unlike every other pattern so far, this one isn't dynamically mapped. It runs ad hoc, with one Dag run per stakeholder request.
- The LLM gets the question alongside a read-only connection scoped to the stakeholders permissions. Note that this pattern uses the `@task.llm_sql` decorator, a specialized decorator that generates SQL based on parameter and prompt inputs that is validated using sqlglot, with optional human review. See the Common AI guide for more information (https://www.astronomer.io/docs/learn/airflow-common-ai-provider#%40task-llm_sql).
- A `SQLExecuteQueryOperator` instance runs the query using a `conn_id` scoped to what that stakeholder is allowed to see.
- The result gets written into a plain-language response by an `@task.llm` task. The response is sent back to the stakeholder.

The most important aspect of this use case is to scope the `conn_id` correctly. Only access-permission guardrails are real guardrails.

```
from airflow.providers.common.sql.operators.sql import SQLExecuteQueryOperator
from airflow.sdk import chain, task

# upstream tasks fetch the question and stakeholder id

@task
def get_scoped_read_connection(stakeholder: str) -> str:
    # deterministic lookup to map the stakeholder to the
    # Airflow connection scoped to what they're allowed to see
    return READ_CONNECTION_BY_ROLE[get_role(stakeholder)]

_read_conn = get_scoped_read_connection(stakeholder=_stakeholder)

@task.llm_sql(
    llm_conn_id="pydanticai_default",
    db_conn_id="db_schema_conn", # Connection ID for database schema introspection
    table_names=["bookings", "payments", "cancellations"], # can also be templated
    at runtime if necessary
    validate_sql=True,
)
def generate_sql(question: str) -> str:
    return question

_generate_sql = generate_sql(question=_question)

_run_sql = SQLExecuteQueryOperator(
    task_id="run_sql",
    conn_id=_read_conn,
    sql=_generate_sql,
)

@task.llm(
```

```

    llm_conn_id="pydanticai_default",
    system_prompt="Answer the user's question based on the query result.",
)
def generate_natural_language_answer(question:str, result:str) -> str:
    return f"Question: {question}, Query result: {result}"

generate_natural_language_answer(question=_question, result=_run_sql.output)

chain([_generate_sql, _read_conn], _run_sql)

# downstream tasks send the answer back to the stakeholder

```

This pattern shows the boundaries of what you can do with a single LLM call. There is no interactivity or ability to look up additional information during the LLM call. Data exploration is a use case that is often better served with an AI agent.

Agent Orchestration

As AI workflows scale and become more complex, organizations often want their AI tasks to have the ability to interact with external systems and perform multi-step processes. This is the moment where you want to replace the LLM in your Airflow Dag with an agent. AI agents interact with the same underlying base models as LLMs and have the option to use **tools** to give them access to any system that has an API, as well as to execute LLM calls in a loop to iterate and reason over more complex requests.

As soon as tools are added to an AI agent, the agent can call those tools and perform any action that is possible through them. If you are adding tools, especially custom tooling with access to production systems, make sure to scope the AI agents' access to prevent destructive actions like dropping a table or deleting a file. Never rely on instructions in a prompt to restrict your agent from performing actions, always enforce access control programmatically.

The following sections cover common use cases for agent orchestration:

- **Data exploration agent:** Exploring an organization's data through natural language, with an agent deciding what to query next based on what it finds.
- **Support ticket agent:** Processing support tickets end to end, looking up account details, checking relevant systems, and drafting replies or taking the next action.
- **Self-improving agents: context graphs and decision traces:** Recording an agent's own past decisions, human inputs and overrides, and their outcomes, then using that history as context on future runs so the agent gets better at its task over time.
- **Dag-as-a-Tool:** Turning the Dag itself into a tool an agent can call, useful with any type of agent and for increasing token efficiency on parameterized multi-step workflows.
- **Root-cause analysis and self-healing pipelines with Otto:** Investigating pipeline failures automatically. Otto, Astronomer's data engineering agent, looks at logs, past runs, deployment configurations, and related tasks based on lineage to find the root cause and propose a fix. You can also create pipelines that automatically act on the type of failure, including opening a PR when the problem lies in the pipeline code itself.

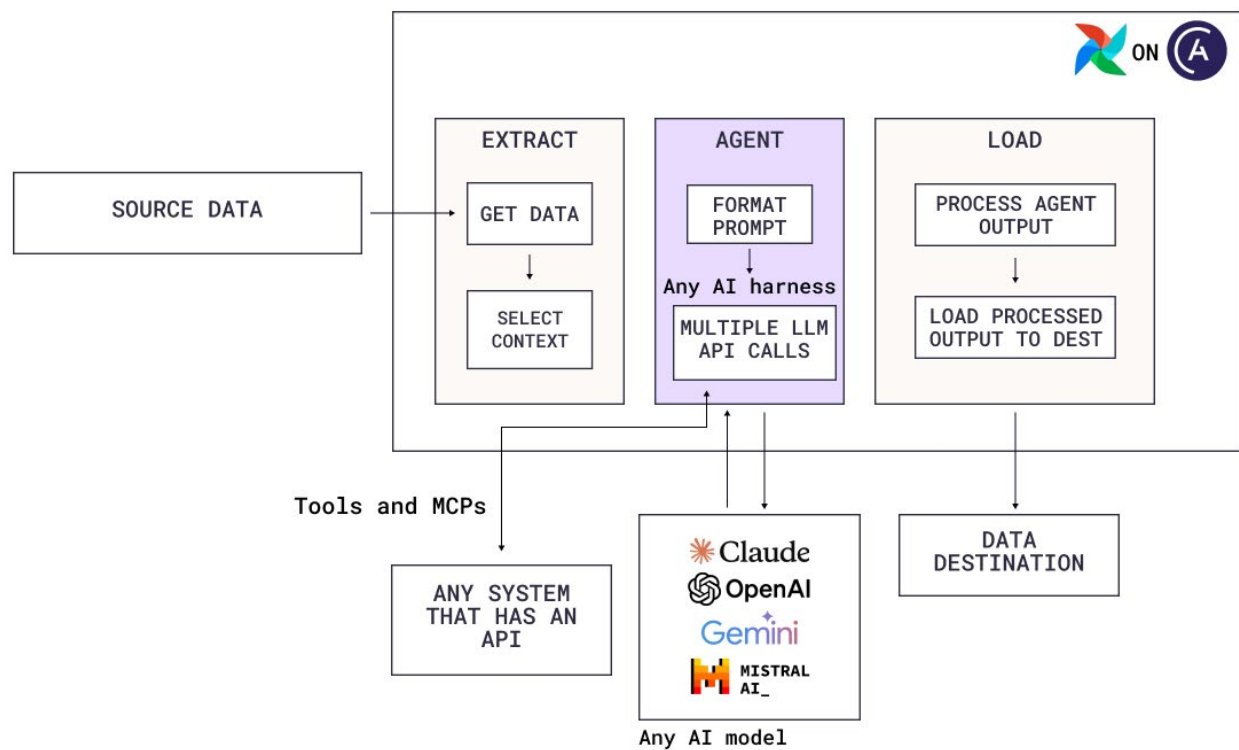


Figure: General diagram of a pipeline orchestrating an AI agent as an Airflow task.

Orchestrating agents as part of data pipelines follows nearly the same structure as a single LLM call: extract data, select what becomes the initial context, and load the result. There is one difference: instead of one API call, the agent makes as many as it needs, and along the way it can call tools or MCPs in addition to the model API itself.

The easiest way to get started with agent orchestration in an Airflow Dag is to use the Common AI provider's `@task.agent` decorator, which offers a similar interface to `@task.llm` but with additional agent-specific functionality. The code snippet below shows the general syntax. See the Common AI provider section in Part 2 for more information.

```

from pendulum import duration
from airflow.sdk import task
from pydantic import BaseModel
from pydantic_ai import FunctionToolset

class MyToolset(FunctionToolset): ... # your custom toolset

class MyOutputClass(BaseModel): ... # your custom output class

```

```

@task.agent(
    llm_conn_id="pydanticai_default", # required: your Airflow connection ID
    # Optional parameters:
    model_id="<your_provider>:<your_model>", # override the model from the conn
    system_prompt="<your_system_prompt>", # default: ""
    output_type=MyOutputClass, # subclass of pydantic.BaseModel or primitive
    toolsets=[MyToolset()], # can be set to a list of pydantic_ai.toolset
instances
    agent_params={"tool_timeout": 60.0}, # default: None
    durable=False, # cache tool outputs
    enable_hitl_review=True, # default: False
    max_hitl_iterations=5, # default: 5
    hitl_timeout=duration(minutes=5), # default: None
    hitl_poll_interval=10.0, # default: 10.0
)
def my_agentic_task(my_input: str) -> str:
    return f"This is the user prompt! {my_input}"

my_agentic_task(my_input="Say hello!")

```

TOOLSETS IN THE COMMON AI PROVIDER

The most salient difference between LLM and agent calls is that you can provide tools to AI agents, enabling them to perform actions. The Common AI provider comes with a set of preexisting toolsets for different use cases.

The `SQLToolset` gives an agent access to a SQL database, restricted to `list_tables`, `get_schema`, `query` (SELECT only by default), and `check_query`.

```

from airflow.providers.common.ai.toolsets.sql import SQLToolset

toolsets=[
    SQLToolset(
        db_conn_id="my_sql_conn",
        allowed_tables=["my_table"], # restrict which tables the agent can access
        # through list_tables and get_schema
        schema="my_schema", # Database schema/namespace for table listing and
        # introspection.
        allow_writes=False, # If True, modify operations are allowed (insert,
        # update, delete). Default: False.
        max_rows=1000 # Maximum number of rows to return from a query. Default:
50.
    )
],

```

The `allowed_tables` parameter is enforced in Common AI 0.5.0+ using sqlglot. This still has some limitations in Common AI 0.6.0: sqlglot cannot evaluate data retrieved through functions or COPY statements. This means that in some cases an LLM can still query tables outside this list if the name is retrieved through these methods. Common AI 0.7.0 closed these gaps and added a `allowed_functions` parameter.

To be fully safe: for query-level restrictions, use database-level permissions (for example a read-only role with grants limited to specific tables).

Setting `allow_writes=True` allows the agent to perform modify operations (insert, update, delete) on the database. Use at your own risk.

The `HookToolset` allows you to use any Airflow hook as a toolset with the methods on the hook being the available agent tools.

```
from airflow.providers.common.ai.toolsets.hook import HookToolset

toolsets=[
    HookToolset(
        hook=MyHook(conn_id="my_conn"),
        allowed_methods=["method1", "method2"],
        tool_name_prefix="my_prefix_", # optional, default: ""
    )
],
```

The `MCPToolset` connects an agent to an MCP server through an Airflow connection.

```
from airflow.providers.common.ai.toolsets.mcp import MCPToolset

toolsets=[MCPToolset(mcp_conn_id="mcp_default")],
```

These are just three of the pre-built toolsets available in the Airflow Common AI provider, see the provider documentation for an up-to-date list at <https://airflow.apache.org/docs/apache-airflow-providers-common-ai/stable/toolsets.html>

Data exploration agent

As you've seen in the previous chapter, data exploration is a use case that can benefit from more than a single LLM call in a pipeline. An agent does not just create one query, but can sequentially create and execute multiple queries, and reason over their results. This architecture shows how you can orchestrate a data exploration agent as part of an on-demand Airflow Dag.

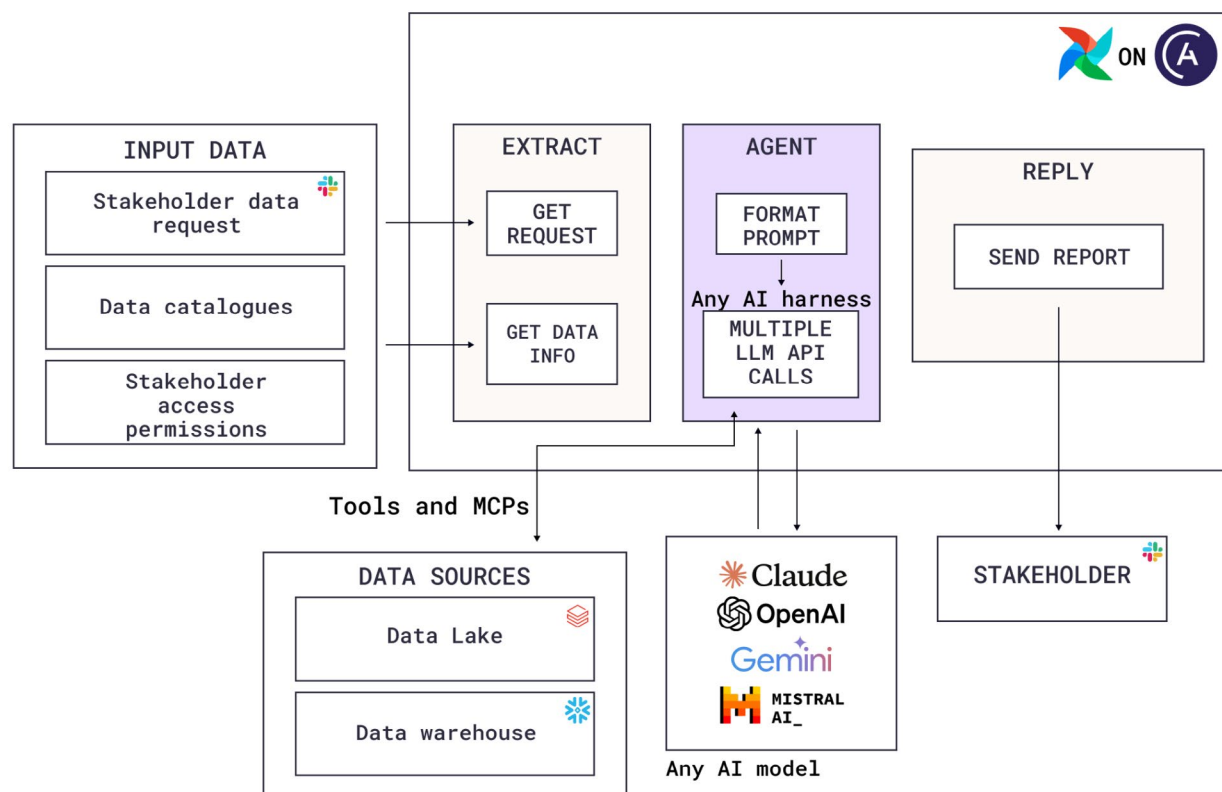


Figure: Diagram showing a possible architecture for a data exploration agent.

The pattern starts the same way Text-to-SQL did: a stakeholder asks a data question in Slack, and the pipeline pulls in the request, the data catalogue, and the stakeholder's access permissions. Next, the AI step uses an agent instead of a single LLM call. The agent runs its own loop using whichever AI harness you prefer, querying your data, reasoning over query results, and deciding what to explore next. Once it has enough information to answer the question, it writes a report and sends it back to the stakeholder in Slack.

At Astronomer we have an internal data assistant called Kepler that uses a similar architecture. (<https://www.astronomer.io/blog/building-kepler-astronomer-internal-data-assistant>)

Like in the Text-to-SQL example, this architecture needs guardrails to avoid executing dangerous queries. There are two main concerns:

1. The agent cannot be allowed to execute queries that modify or delete the data. Accidental **DELETE** queries still make occasional headlines (<https://www.theguardian.com/technology/2026/apr/29/claude-ai-deletes-firm-database>) and can have catastrophic consequences for organizations.
2. The agent should only be able to access data that the stakeholder is allowed to access. A product manager asking about feature adoption should not be served back sensitive PII.

While you can write “never delete any tables” in your prompt, there is no guarantee that the agent will follow the rule. There are many ways, from prompt injection to loss of instructions during context window compaction, that AI agents stop following rules given in their context. For examples, see the ContextOops chapter in the AI context engineering with Apache Airflow® eBook (<https://www.astronomer.io/ebooks/ai-context-engineering-with-apache-airflow/>).

There is only one effective guardrail to protect yourself against unintended queries. Every tool the agent can call needs its own access, scoped at the level of the Airflow connection, to what the stakeholder is allowed to see and do via this specific agent.

Once you’ve created a properly scoped Airflow connection, you can use the pre-built toolsets from the Common AI provider as shown in the section above.

This pattern uses an agent and lets the agent explore, reason, and return its results to the internal stakeholder that sent the original query. In case there is anything wrong or missing, the employee can ask again or escalate the issue to the data team.

As soon as you start to create external-facing agentic data products, you are going to need more quality control *within* your pipeline. You would not want a customer to get a response saying “sorry, I don’t know what to do about your login issues”, if the agent makes a mistake or can’t find an answer. This is why the next pattern, a support ticket agent, needs a human-in-the-loop.

Support ticket agent

AI agents can be used to assist employees in some of their tasks, often gathering relevant information and making a first pass at creating text based on that information. The data exploration agent shown in the previous section is an example of an AI-based data product for *internal* stakeholders. As soon as the AI-generated asset is sent to people outside of your company, you've created an AI-based data product for *external* stakeholders, usually with higher stakes.

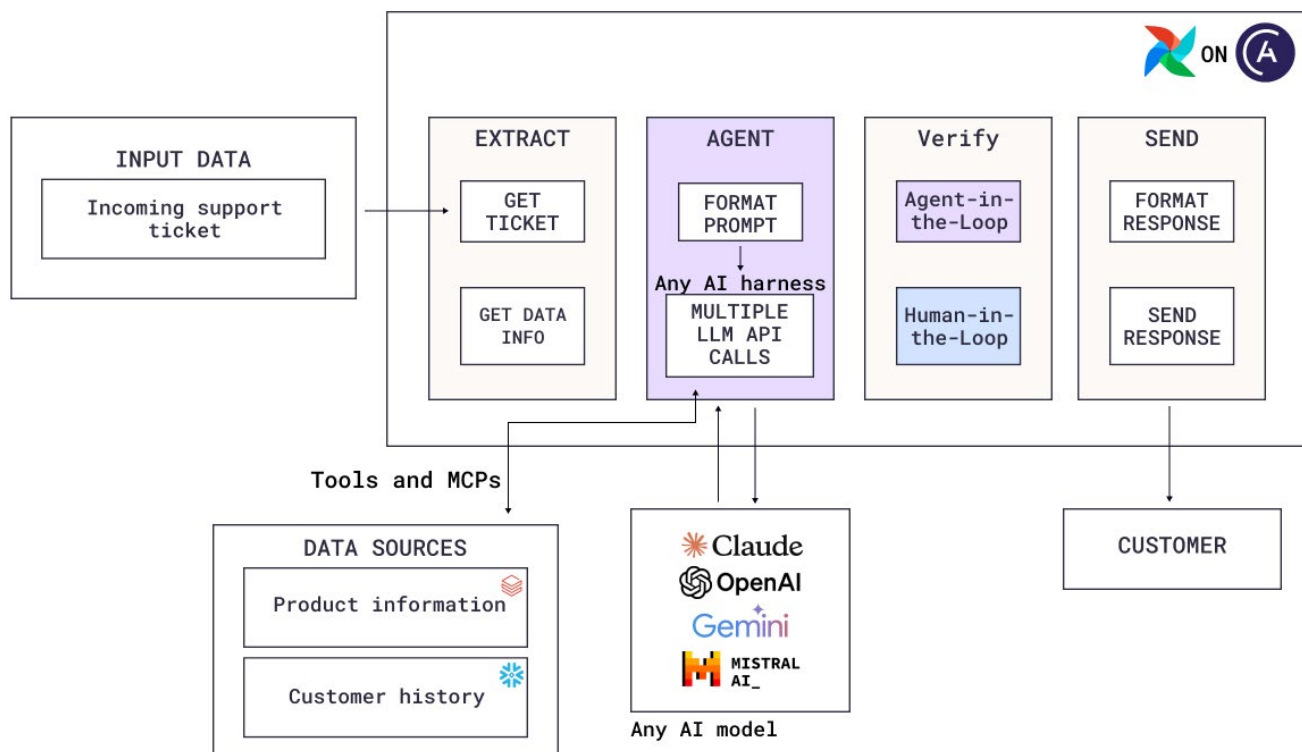


Figure: Diagram showing a possible architecture for a support ticket agent with two quality control steps: *agent-in-the-loop* and *human-in-the-loop*.

A support ticket agent is one example of such an external-facing data product. The incoming support ticket starts the Dag, which fetches the ticket and then uses an AI agent to draft a reply. The AI agent has access to information about your product and this specific customer through tools and MCPs. As described in the previous section it is very important to scope the connection querying customer information correctly, to avoid potentially leaking the purchase history of customer A to customer B.

At Astronomer we use this pattern internally in the pipeline running the Edu Agent, which we use to answer queries from students taking courses in the Astronomer Academy. For details see the AI-powered education operations with Apache Airflow® reference architecture (<https://www.astronomer.io/docs/learn/airflow-3/reference-architecture-ai-education-operations>).

AGENTS AND HUMANS IN THE LOOP

Because this pipeline generates text meant to be sent out to customers, extra quality control steps are added:

- **Agent-in-the-loop**, which is sometimes also referred to as **agent-as-a-judge**, is a pattern where one or more downstream tasks use additional AI agents to cross-check the work of the initial agent(s) in a pipeline.
- **Human-in-the-loop (HITL)**, which describes the act of having a step in your pipeline that requires the input of a human. In the Airflow standard provider (<https://airflow.apache.org/docs/apache-airflow-providers-standard/stable/index.html>) there are several pre-created human-in-the-loop operators for this purpose.

It is common to combine these steps sequentially, and you can even have multiple agents scoring the AI-generated text from different perspectives, such as accuracy and tone.

The human-in-the-loop step comes last, and the human responding to it takes on the responsibility for the information in the response being correct.

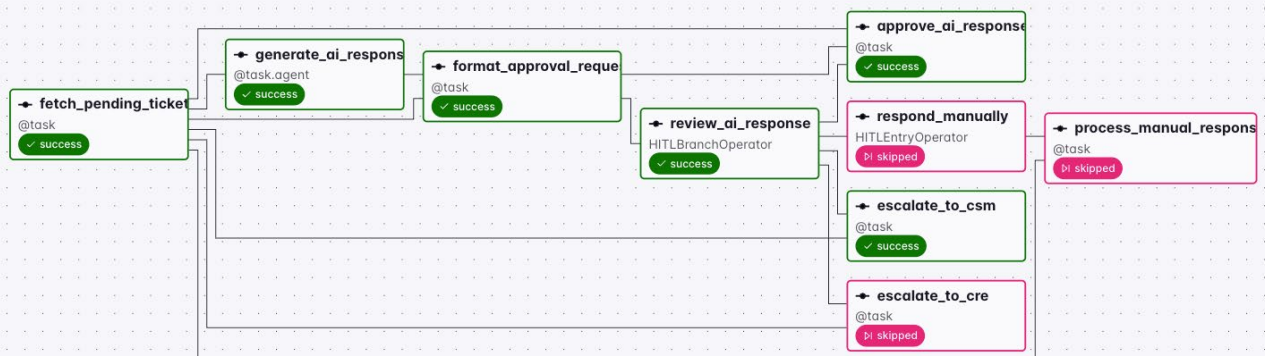


Figure: An Airflow Dag showing a pattern with a HITLBranchOperator, a subclass of the HITLOperator that lets the human pick a downstream branch in a Dag.

This Dag shows a simple implementation of a human-in-the-loop step in a support ticket agent pipeline. The AI drafts the ticket response, and the human chooses among four options. The code snippet below shows the `@task.agent` task, as well as the `HITLBranchOperator`. You can find the full version of this Dag in the book companion repository (<https://github.com/astromer/ai-orchestration-examples>).

```
# ...
@task.agent(
    llm_conn_id="pydanticai_default",
    system_prompt=SYSTEM_PROMPT,
    output_type=TicketResponse,
    toolsets=[shipment_toolset],
)
def generate_ai_response(ticket: dict) -> str:
    return build_user_prompt(ticket)

_generate_ai_response = generate_ai_response(_fetch_pending_ticket)
# ...

_review_ai_response = HITLBranchOperator(
    task_id="review_ai_response",
    subject="AI Support Response Ready for Review",
    body="""**Please review the AI-generated support ticket response below:**

{{ ti.xcom_pull(task_ids='format_approval_request')['ticket_info'] }}

**AI Summary:**
{{ ti.xcom_pull(task_ids='format_approval_request')['summary'] }}

**AI Suggested Priority:** {{ ti.xcom_pull(task_ids='format_approval_request')
['priority'] }}

**AI Confidence:** {{ "%.0f" | format(ti.xcom_pull(task_ids='format_approval_
request')['confidence'] * 100) }}%

**Suggested Tags:** {{ ti.xcom_pull(task_ids='format_approval_request')['suggested_
tags'] | join(', ') }}"

```

```

**AI Response:**
...

{{ ti.xcom_pull(task_ids='format_approval_request')['ai_response'] }}
...

**Instructions:**
- **Approve**: Send this response to the customer
- **Reject**: Route to human agent for manual response

Please review for accuracy, tone, and completeness.""",
    options=[
        "Approve AI Response",
        "Respond Manually",
        "Escalate To CRE",
        "Escalate To CSM",
    ],
    options_mapping={
        "Approve AI Response": "approve_ai_response",
        "Respond Manually": "respond_manually",
        "Escalate To CRE": "escalate_to_cre",
        "Escalate To CSM": "escalate_to_csm",
    },
    defaults=["Escalate To CSM"],
    multiple=False,
    response_timeout=timedelta(hours=4),
)

```

Self-improving agents: context graphs and decision traces

The agent in the previous example can be very useful, but it cannot learn from its own mistakes. When the agent creates a ticket response that is incomplete or wrong and the human makes changes to it, there is no pipeline feeding these human decisions back to the agent. The *decision trace* is not captured and never turned into a *context graph*.

A **decision trace** contains all data relevant to one specific decision instance: the inputs (for example: definitions, available discounts, policies, exceptions), all decisions involved (AI and human), and the outcome.

For a full example of a decision trace, see the Context graphs and decision tracing chapter in the AI Context Engineering with Apache Airflow® eBook (<https://www.astronomer.io/ebooks/ai-context-engineering-with-apache-airflow/>).

A **context graph** is the accumulated structure formed by those traces: a collection of decision traces for the same or highly similar business processes. Foundation Capital (<https://foundationcapital.com/ideas/context-graphs-ais-trillion-dollar-opportunity>) calls it “a living record of decision traces stitched across entities and time so precedent becomes searchable.” An agent handling similar requests can search the graph for precedent and make more informed suggestions, and can eventually handle more requests autonomously.

To trace a decision, you need to understand the full business process. Next, start capturing decisions as part of your pipelines and define how and in what format a decision trace gets recorded.

In this pattern, which creates a self-improving sales assistant agent, several Dags orchestrate different actions.

- **Context gathering Dag:** Retrieves existing context and memory from the context store, including relevant past decision traces. This gives the AI agent access to how similar customers were handled previously, as well as detailed information about the customer and market signals in their business domain.
- **Draft communication Dag:** This is the agent orchestration Dag. In this pattern, it uses the gathered context to generate an AI draft proposal, then evaluates it with an AI-as-a-judge step. A human-in-the-loop step sends a message to the requesting stakeholder to ask for approval, edit suggestions, and the reasoning behind any proposed changes. Depending on the outcome, the

draft is either approved and the next Dag is triggered, or the draft is rejected and a second run of the draft communication Dag is triggered to propose an improved draft.

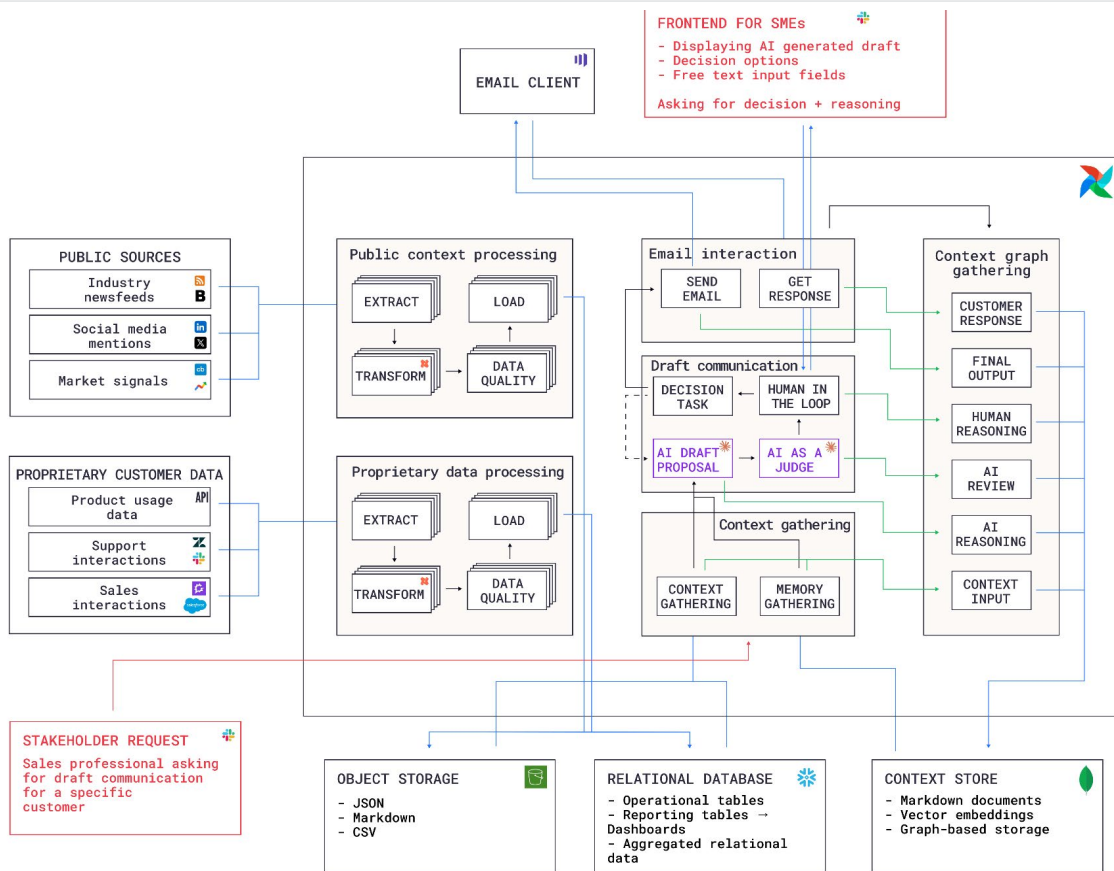


Figure: An architecture diagram of an AI orchestration pipeline assisting sales professionals in drafting outreach emails.

- **Email interaction Dag:** Sends the approved communication through the email client and captures the customer's response.
- **Context graph gathering Dag:** Assembles the complete context graph by collecting the context input, AI draft and reasoning, AI review, human decision and reasoning, final output, and customer response. Stores the assembled context graph in the context store for future retrieval by the AI agent.

The exact composition of these Dags will vary depending on your use case. The important element of self-improving AI patterns is to gather AI evaluation information and feed it back into the context that is available to your agent at inference time. For a proper context graph, you'll want to gather human intervention and decisions for similar cases over time alongside the humans' reasoning for their changes.

Dag-as-a-Tool

AI orchestration with Airflow means running an LLM or AI agent as an Airflow task, and context engineering pipelines defined as Airflow Dags are already an emerging industry standard. But there is a third way to use Airflow with AI agents, turning it upside down and letting your agents run your Dags!

If you look at your existing AI skills, some of them likely describe a multi-step workflow you'd like your AI agent to follow as diligently as possible. Which they do... most of the time. By turning skills into Airflow Dags you ensure that all steps are executed with the right dependencies.

The second advantage of this pattern is token saving. Take for example the following AI agent skill that performs a business operation: onboarding a new employee.

```
---
name: onboard-employee
description: Provisions a new employee across company systems and sends a welcome email
---
1. Create an account for the new employee in the identity provider
2. Once that account is confirmed, create matching accounts in chat, source control, and the payroll system
3. Create additional accounts in systems based on the employee's role
4. Retry any account creation that fails
5. Once all accounts are confirmed, send the employee a personalized welcome email with their login details
```

This skill is very inefficient. The agent re-creates the code to create new accounts every time it uses the skill, as well as a monitor to wait until the identity provider creation has completed successfully, and code to check if any of the actions failed and to retry them.

This can be improved a little bit by creating tools for each individual action, but the most efficient way to perform this process is as an Airflow Dag, where you can have automatic retries, and tasks wait for upstream completions already.

The only step in this skill that actually profits from using AI is the last one, writing a personalized welcome email. All other steps could be performed cheaper and more reliably by code that is written once and wrapped into an Airflow Dag, as shown in the screenshot.

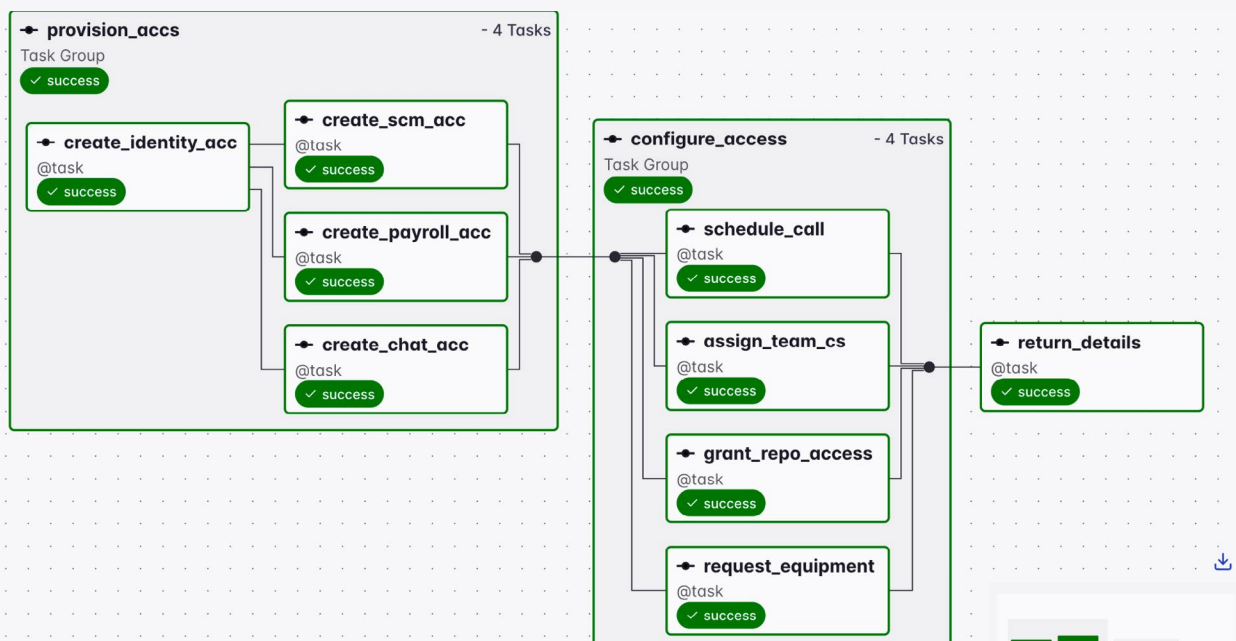


Figure: Screenshot of a Dag running deterministic tasks to onboard a new employee.

Now the skill is much shorter and you only spend AI tokens on the creation of the personalized email.

```

---
name: onboard-employee
description: Provisions a new employee across company systems and sends a welcome email
---

1. Run the `onboard_employee` Airflow Dag using your `run_airflow_dag` tool
2. Use the account details that are returned to send the employee a personalized welcome email with their login details
  
```

The above example was for an AI agent with a local harness like Claude Code, running on the laptop of an HR employee, but the same pattern can be used with agents deployed inside an Airflow task.

For more on converting skills into Airflow Dags, see this blog post: Skills vs. Pipeline: Two Ways to Build the Same AI Workflow by Vikram Koka (<https://medium.com/apache-airflow/skills-vs-pipeline-two-ways-to-build-the-same-ai-workflow-b5dd7088e122>)

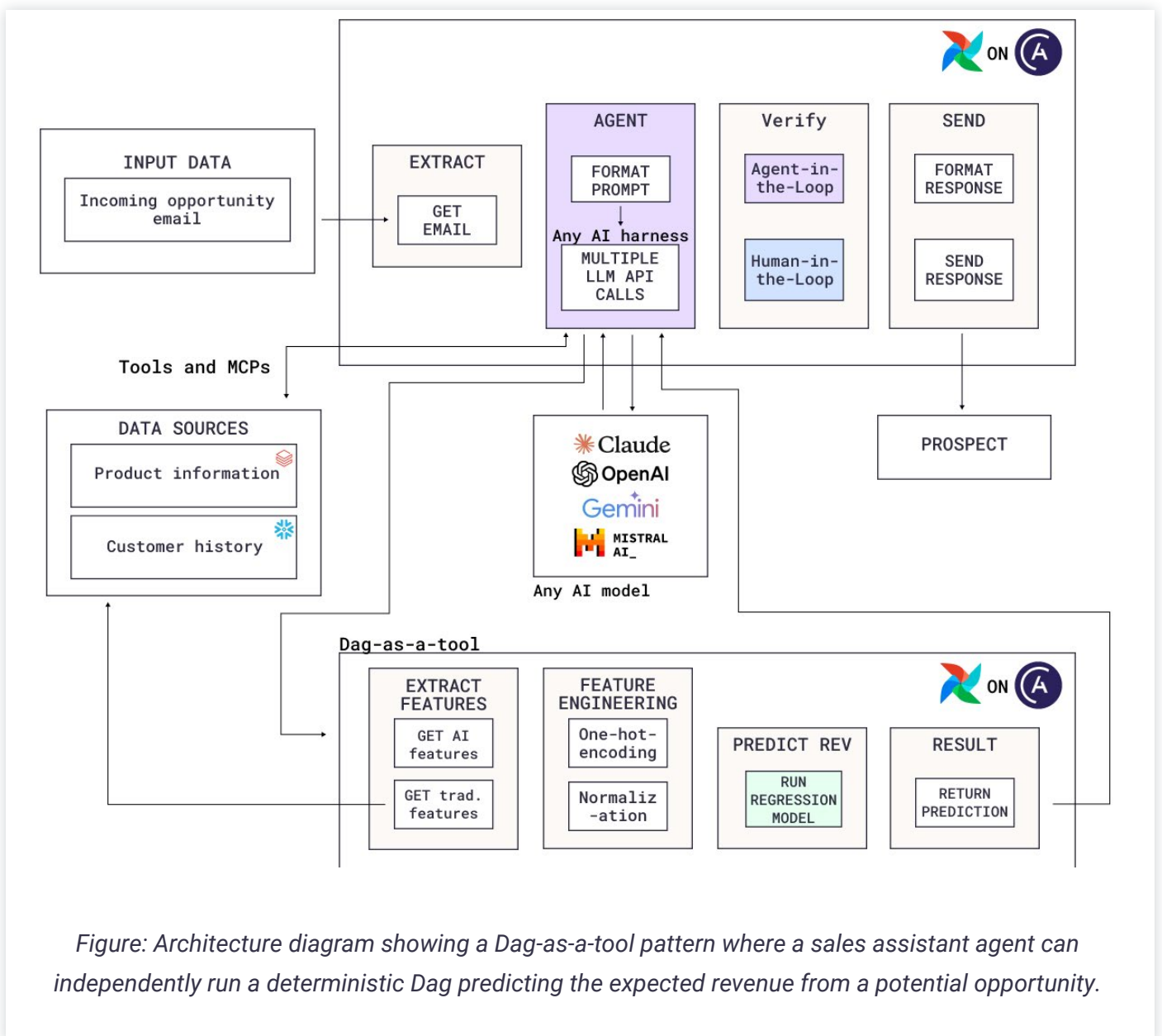


Figure: Architecture diagram showing a Dag-as-a-tool pattern where a sales assistant agent can independently run a deterministic Dag predicting the expected revenue from a potential opportunity.

The sales assistant agent from the previous pattern would benefit from being able to predict the potential revenue for each incoming opportunity to know which product tier and features to put front and center in the outreach email. Predicting a continuous value like revenue is a regression problem, one of the types of machine learning where specialized traditional models are more efficient to use than LLMs.

This is a great use case to add a Dag-as-a-Tool, specifically a Dag that runs ML inference using a trained regression model. If the AI agent determines it could benefit from a revenue estimate it runs this Dag using the Airflow REST API (<https://www.astronomer.io/docs/astro/airflow-api>) for the opportunity being assessed and gets the revenue prediction as a result.

In this example the Dag that is being used as a tool is fully deterministic, but you can use any Dag as a tool, including Dags that use LLMs or agents as individual steps. For example, you could give the sales assistant agent a tool to run the LLM localization and translation Dag to translate responses for inquiries in a foreign language in order to ensure that the outreach email uses the same localized terms as your product documentation.

What you should avoid is an accidental infinite loop. If Dag A contains an agent that can start Dag B, Dag B should *not* contain an agent that can in turn start Dag A.

Root-cause analysis and self-healing pipelines with Otto

The term “self-healing” is overused. Vendors use it for anything from a retry loop to Kubernetes autoscaling. Here we mean something concrete: when a Dag run fails, an AI agent with access to information about your Airflow environment investigates the failure, generates a hypothesis about the cause, and emits a recommendation. Based on the recommendation, you can trigger downstream actions like alerting a stakeholder or opening a PR to fix a bug in the Dag code.

For example, you can implement the following pattern:

1. Any of your Dag runs fail.
2. An Astro Alert configured to fire on Dag run failures sends a notification and triggers the `auto_fix` Dag.
3. The `auto_fix` Dag contains a task `get_diagnosis` that calls Otto, Astronomer’s AI agent, via an API endpoint to investigate the Dag run failure.
4. If the failure is due to a mistake in the Dag code, a downstream task using the `@task.llm` decorator will write a PR based on Otto’s suggestion for a code fix.
5. A final task submits a PR to GitHub with the fix.
6. A human reviews the PR on GitHub and merges it.

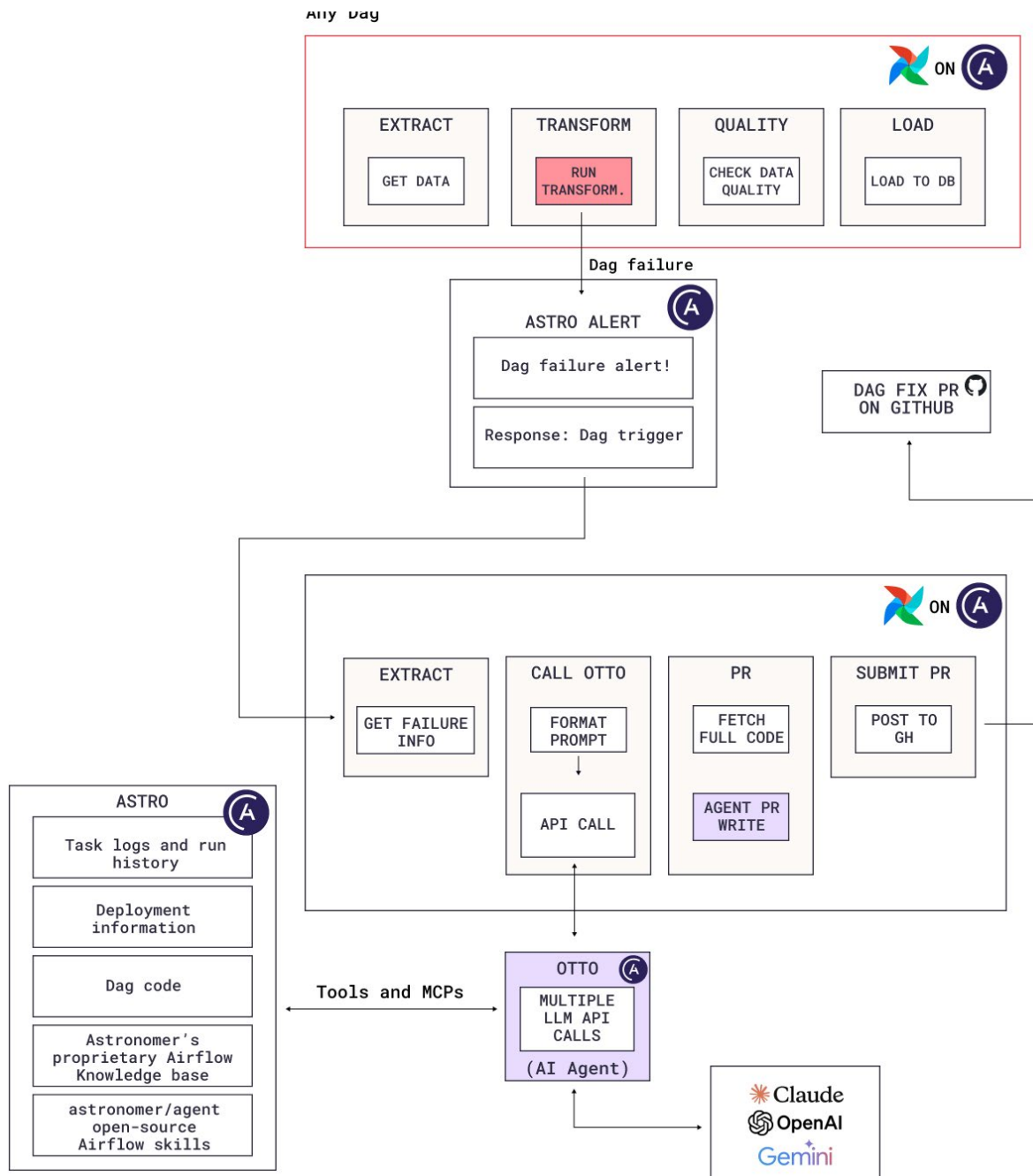


Figure: Architecture diagram for an auto_fix Dag which creates PRs to fix pipeline failures based on Otto's investigations.

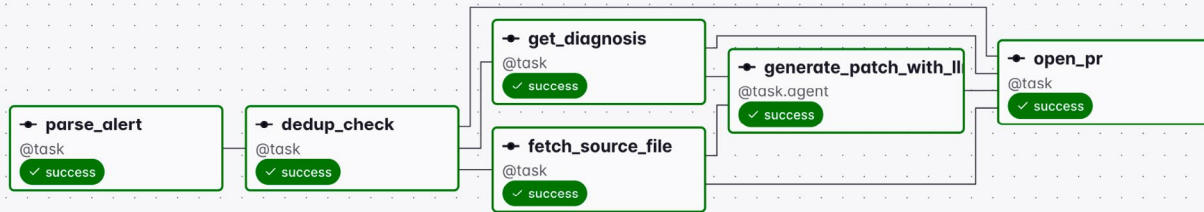


Figure: A possible structure of an `auto_fix` Dag with a task running the investigation agent, one to generate the code suggestion, and one to submit a PR to GitHub

See the How to use Otto to automatically investigate Dag failures and PR a fix tutorial (<https://www.astronomer.io/docs/learn/airflow-3/airflow-otto-rca-auto-fix>) for step-by-step instructions and example code on how to implement this use case on Astro.

[AUTOMATED PR] Snowflake COPY INTO fails casting "62.7%" string to numeric column #5

Ready to merge

Code

Open TJaniF wants to merge 1 commit into main from auto-fix/fetch_cargo_manifests/manual-2026-06-30T23-36-

Conversation 0 Commits 1 Checks 0 Files changed 1

+1 -1



TJaniF commented on Jul 1

Owner

Auto-generated by `auto_fix_pr_handler` based on Astro Investigation Agent diagnosis.

Summary

The `copy_to_staging` task failed with Snowflake `ProgrammingError 100071 (22000): Failed to cast variant value "62.7%" to FIXED`. The DAG generates `cargo_fill_pct` as a formatted percentage string (e.g. "62.7%") and writes it into the staged Parquet file, but the target Snowflake column `RAW_STG_CARGO_MANIFESTS.CARGO_FILL_PCT` is a numeric (FIXED) type, so the COPY's `MATCH_BY_COLUMN_NAME` cast cannot convert the string value. The same broken code was already present in the prior deploy, so the failure is a pre-existing bug rather than a regression introduced by the latest deploy.

Root cause

- Cause:** In `dags/examples/fetch_cargo_manifests.py` (line 104), `generate_cargo_manifests` produces `cargo_fill_pct` as `f"{random.uniform(35.0, 98.0):.1f}%"` — a string with a trailing "%". The Parquet file staged to S3 therefore carries `cargo_fill_pct` as a string column. The downstream `copy_to_staging` task runs `COPY INTO DWH.PROD.RAW_STG_CARGO_MANIFESTS ... FILE_FORMAT = (TYPE = PARQUET) MATCH_BY_COLUMN_NAME = CASE_INSENSITIVE`, and Snowflake raises `ProgrammingError 100071 (22000): Failed to cast variant value "62.7%" to FIXED` because the destination column is numeric. The same broken code was present in the previous deploy, so this is a pre-existing data-shape bug, not a regression from the latest deploy.
- Task:** `fetch_cargo_manifests.copy_to_staging`
- Type:** `USER_CODE_PYTHON_CODE_ERROR (PERMANENT)`

Reviewers

Copilot Lite

Request

Still in progress? Convert to draft

Assignees

No one—assign yourself

Labels

auto-fix:fetch_cargo_manifests

Projects

None yet

Milestone

No milestone

Development

Successfully merging this pull request may close these issues.

None yet

Figure: Example of an automatically generated PR to GitHub based on this auto-fix pattern

CASE STUDY

Janus Henderson

INVESTORS



Janus Henderson Investors, a global asset manager with roughly \$480 billion (as of Q1 2026) under management, uses Otto, Astronomer's data engineering agent, to diagnose failures across their 130+ production Airflow pipelines that produce quantitative trading signals before market open. Instead of an engineer manually digging through logs when a Dag run fails, Otto investigates using proprietary Airflow, Astro, and observability context, and returns a root cause, severity, and suggested fix, work that previously took anywhere from five minutes to an hour per incident.

Janus Henderson built on top of Otto's diagnostics with Lighthouse, an internal remediation framework that routes every production failure to the right handler, opens a pull request when the fix is a code change, and retriggers failed tasks through the Astro API. Across more than 230,000 monthly successful task runs in 27 production Deployments, this cut their mean time to resolution by 95 percent.

You can learn more about Janus Henderson's use of Otto for root-cause analysis and self-healing pipelines in this case study: <https://www.astronomer.io/case-studies/janushenderson/>.

Multi-Agent Orchestration

Like with humans, sometimes a challenge can be solved more efficiently by an entire team, instead of just one (silicon) mind.

This section covers three foundational multi-agent patterns.

- **Orchestrator-worker patterns** with intelligent model routing: This is the most common and efficient pattern when orchestrating multiple agents. The newest generation frontier model (the orchestrator) splits up a complex task into easier subtasks and cheaper models (the workers) perform those tasks.
- **Agentic councils**: This pattern describes using AI agents to evaluate information from different viewpoints and through that form a more robust conclusion.
- **Agentic software development**: We are all already using agents to write code. This architecture describes how you can automate bug fixing and feature creation even for complex code bases by using multiple agents, working together and evaluating each other's work like a team of junior engineers.

Orchestrator-worker patterns

Most complex tasks can be broken down into simpler individual units of work. This principle is behind the orchestrator-worker pattern, which is often implemented as a cost-saving measure. While you can use the latest frontier model to perform the entire task, this is often quite expensive and your tokens are better spent breaking down the task into smaller, easier units of work and then letting smaller models, or even locally hosted models, complete those tasks.

Orchestrator-worker patterns start with one orchestrator task using a sophisticated model. This orchestrator agent performs three actions:

- Breaking down the work to be completed into smaller portions of work and writing instructions for downstream worker agents, which are passed to them as user prompts.
- Deciding the appropriate model and available toolsets for each downstream task, based on the complexity of the assigned portion of work. For some tasks like summarization a simple LLM

might be sufficient, while other tasks might necessitate an agent with database access. This work is often referred to as *intelligent model routing*.

- Defining success criteria for each individual portion of work as well as for checking the quality of the completed work, which is usually done by a downstream agent-in-the-loop step.

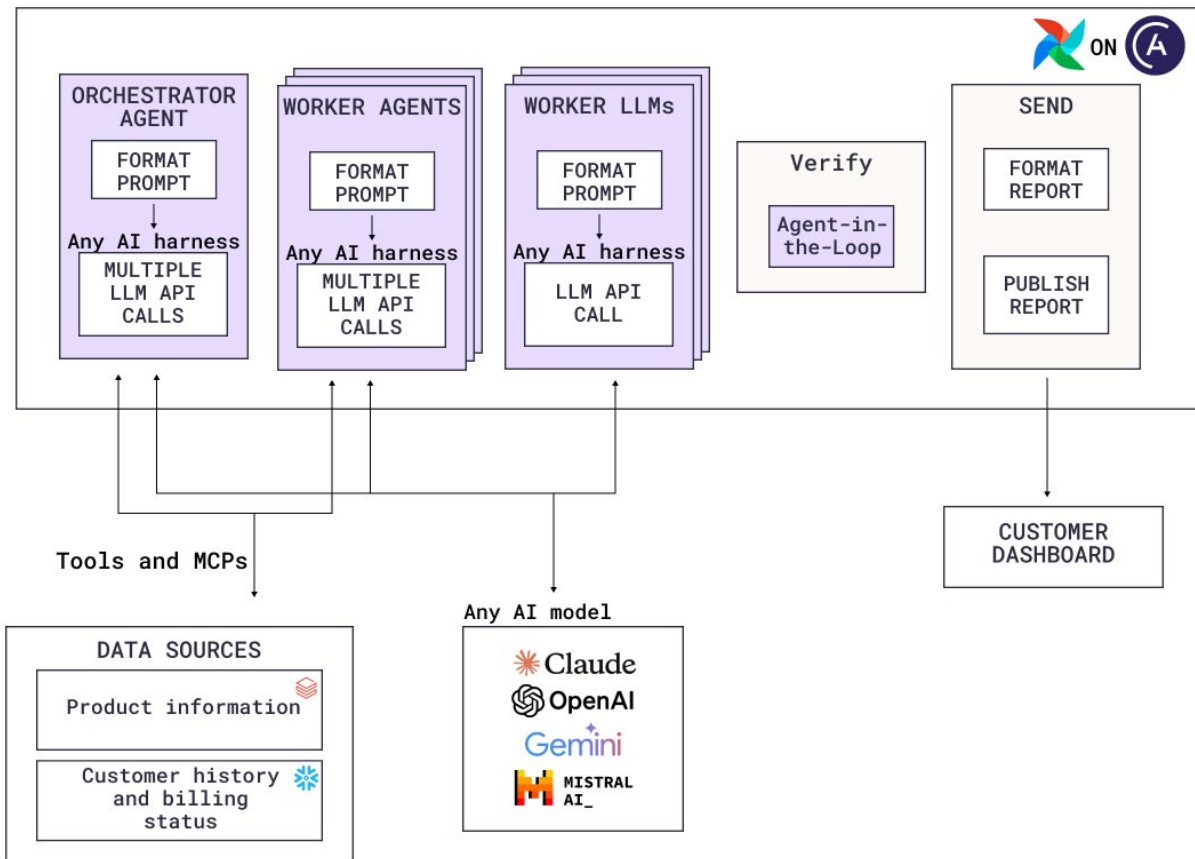


Figure: Architecture diagram of an orchestrator-worker pattern used to write customized summaries of account activity on a customer-facing dashboard.

Agentic councils

When evaluating complex data, it can be helpful to assess it from different perspectives. This principle holds true for AI agents as well. If these agents start exchanging information between themselves, it turns into an agentic council pattern.

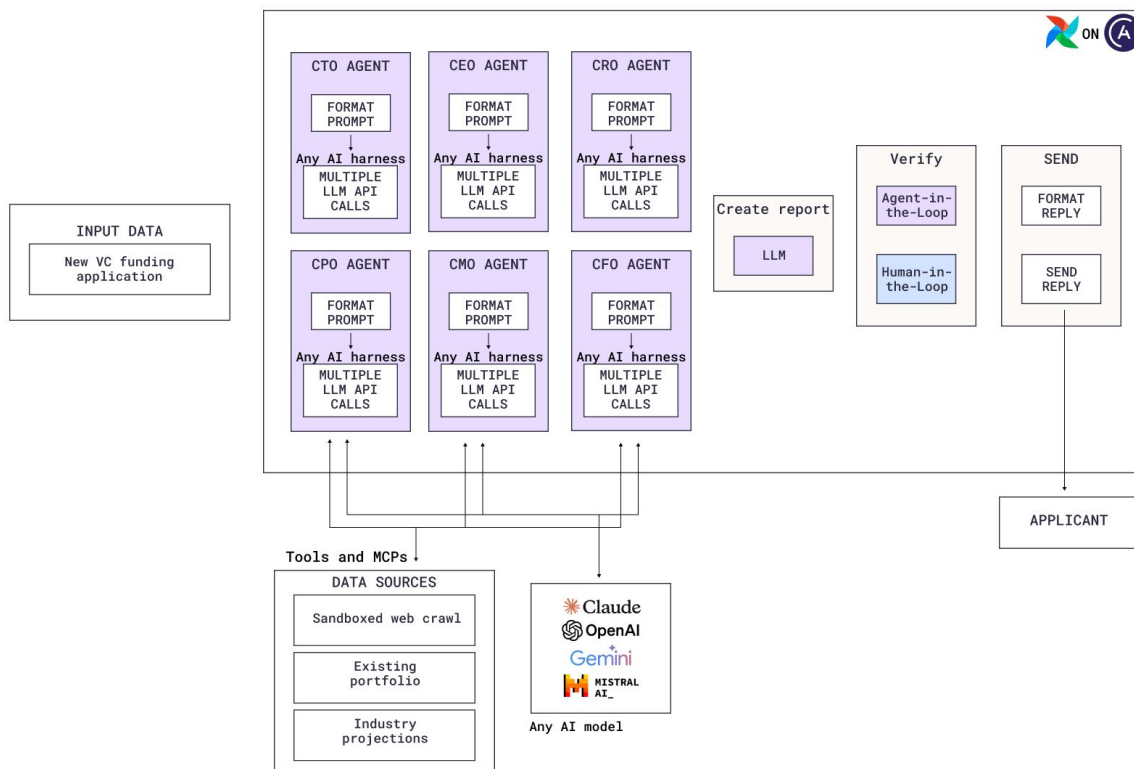


Figure: Diagram of an agentic council evaluating an application for VC funding from the perspective of different roles in a startup.

Typically all agents in a council are given access to a system that allows them to share information. This can be as simple as using an S3 bucket as an “inbox” and letting each agent create new files and read all existing files. You can even have private inbox locations if you want your agents to be able to exchange direct messages with each other.

In the example above a group of six agents evaluate the application of a startup for more funding from different executive perspectives, from the CTO evaluating technical feasibility to the CFO plotting a potential path to profitability. Once all agents have come to their conclusions, an LLM step combines the assessments into a report, which goes through two verification steps (one agentic, one human) before the reply is sent to the applying startup.

Agentic software development

At many organizations, a majority of code is now written by AI and a lot of software developers are moving from working with local harnesses to developing deployed agentic systems that automatically write code at larger scales.

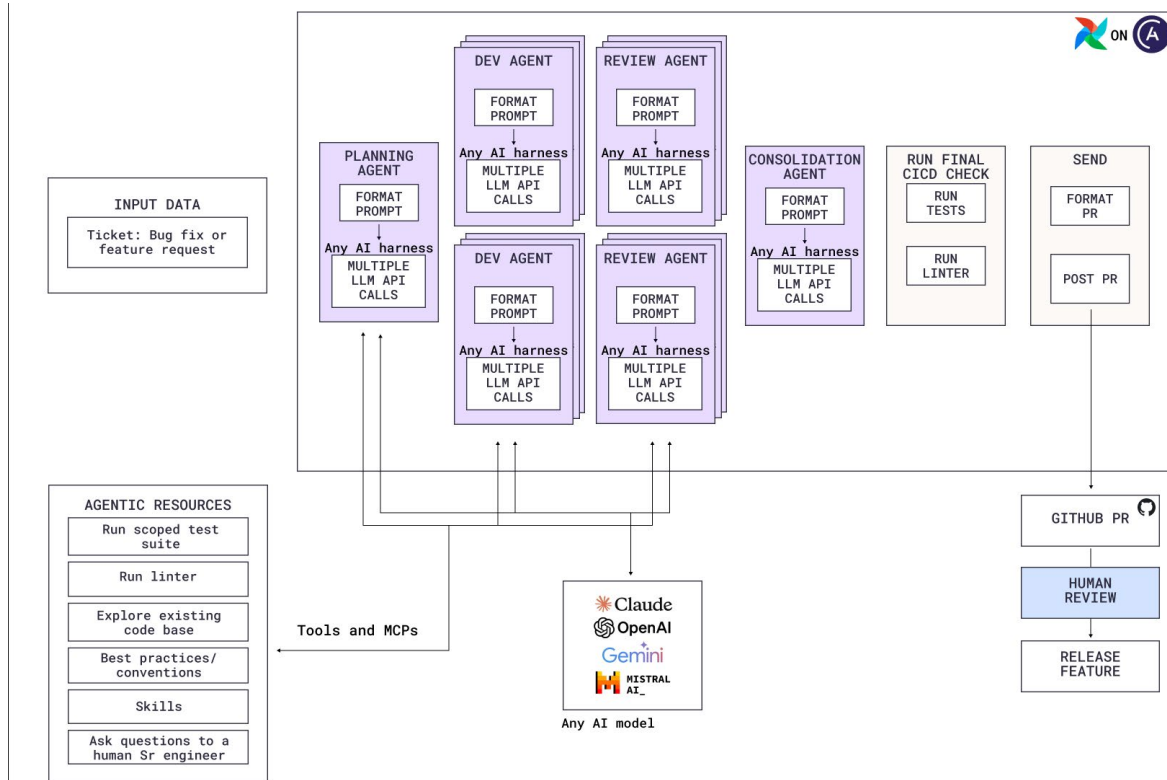


Figure: Architecture of a pipeline that pulls in a ticket about a bug or feature request, then uses groups of agents to develop, test and PR the code changes.

This example shows a possible way to structure your agent team of junior developers, starting with the familiar orchestrator-worker pattern where one agent plans out the individual steps for the agentic sprint to be given to developer agents. Downstream you have a set of reviewer agents as well as a consolidation agent that ensures the individual work of the developer agents is compatible.

One of the most important aspects for successful AI-driven code writing is to give agents clear success criteria and ways to unit test their code. Agentic resources like scoped test suites, linters, and documented best practices and conventions, which greatly improve the code quality. The full test suite is run by a deterministic task after the agentic tasks have completed and the final responsibility lies with the human reviewer who approves the pull request.

See the data engineering with AI learn guides (<https://www.astronomer.io/docs/learn/local-data-engineering-with-ai-overview>) for more information on how to improve your AI's code writing capabilities.

Context Engineering with Airflow

Every pattern in this book depends on the data given to each model call, the model’s context. Context engineering is the “delicate art and science of filling the context window with just the right information for the next step” (Karpathy responding to Lütke, June 2025, <https://x.com/karpathy/status/1937902205765607626>). ContextOps covers everything that keeps that context accurate over time, because context decays and something has to keep it current.

There are four main types of context engineering:

Pattern	Source of context	Refresh pattern
Manual	A human, ad hoc	One-shot, for example pasting a stack trace into a prompt
Traditional	Operational systems: product databases, object storage, APIs, RAG	Scheduled or event-driven pipelines
The context layer	Your data ecosystem itself: schemas, lineage, Dag structure, run history, asset events	Produced continuously by orchestration
Context graphs (See the section on Self-improving agents: context graphs and decision traces in the Agent Orchestration chapter)	Human decisions accumulated over time: approvals, rejections, rationale	Captured per decision, aggregated into a graph

Airflow plays three roles in context engineering:

- 1. Operational:** Airflow builds and maintains context units. Chunking source documents, generating embeddings in parallel, writing to vector stores or graph databases, updating metacontext fields as units change, and pausing for human input where a decision needs to be recorded.
- 2. As context itself:** Dag run history, task execution records, XCom values, asset events, and lineage are all context an agent can consume. This is the context layer, and you do not build pipelines for it. Airflow produces it as a side effect of running your Dags.

3. **As actionable context:** a Dag can be invoked as a tool through the Airflow REST API, executing a parameterized multi-step workflow and returning its result as context. Unlike ad-hoc agent-written code, it comes with retries, observability, and can be reused across different agents and agent sessions. See the Dag-as-a-Tool section in the Agent Orchestration chapter for more information.

For a comprehensive resource on context engineering check out our AI Context Engineering with Apache Airflow® eBook. (<https://www.astronomer.io/ebooks/ai-context-engineering-with-apache-airflow>)

RAG pipelines: Document embedding

RAG (retrieval augmented generation) is one of the most widely used patterns for giving an LLM access to information it wasn't originally trained on. To implement RAG you need an indexing pipeline, which fetches new content, chunks, and embeds it and loads it into a vector database. Information in the vector database can be accessed in various ways, including by giving an AI agent a similarity search tool, or adding an upstream task in an LLM orchestration pipeline that runs the search deterministically and attaches the output to the user prompt.

The diagram shows a knowledge base for an eCommerce store. A daily Dag embeds new information about customers and products and writes it to a vector database. A second Dag, triggered as soon as a customer completes a purchase, uses the products in that purchase and their descriptions to retrieve related items from the vector database, then gives that context to an LLM to generate a personalized recommendation shown on the customer's account page.

The indexing Dag has three steps, similar to an ETL pipeline, with the transformation step including chunking and embedding.

1. Product descriptions, catalog updates, and customer interaction data are extracted and chunked to fit inside the embedding call.
2. Each chunk is embedded using any embedding model.
3. Embeddings are written to the vector database, along with the source text, metadata like source record ID and last-updated timestamp.

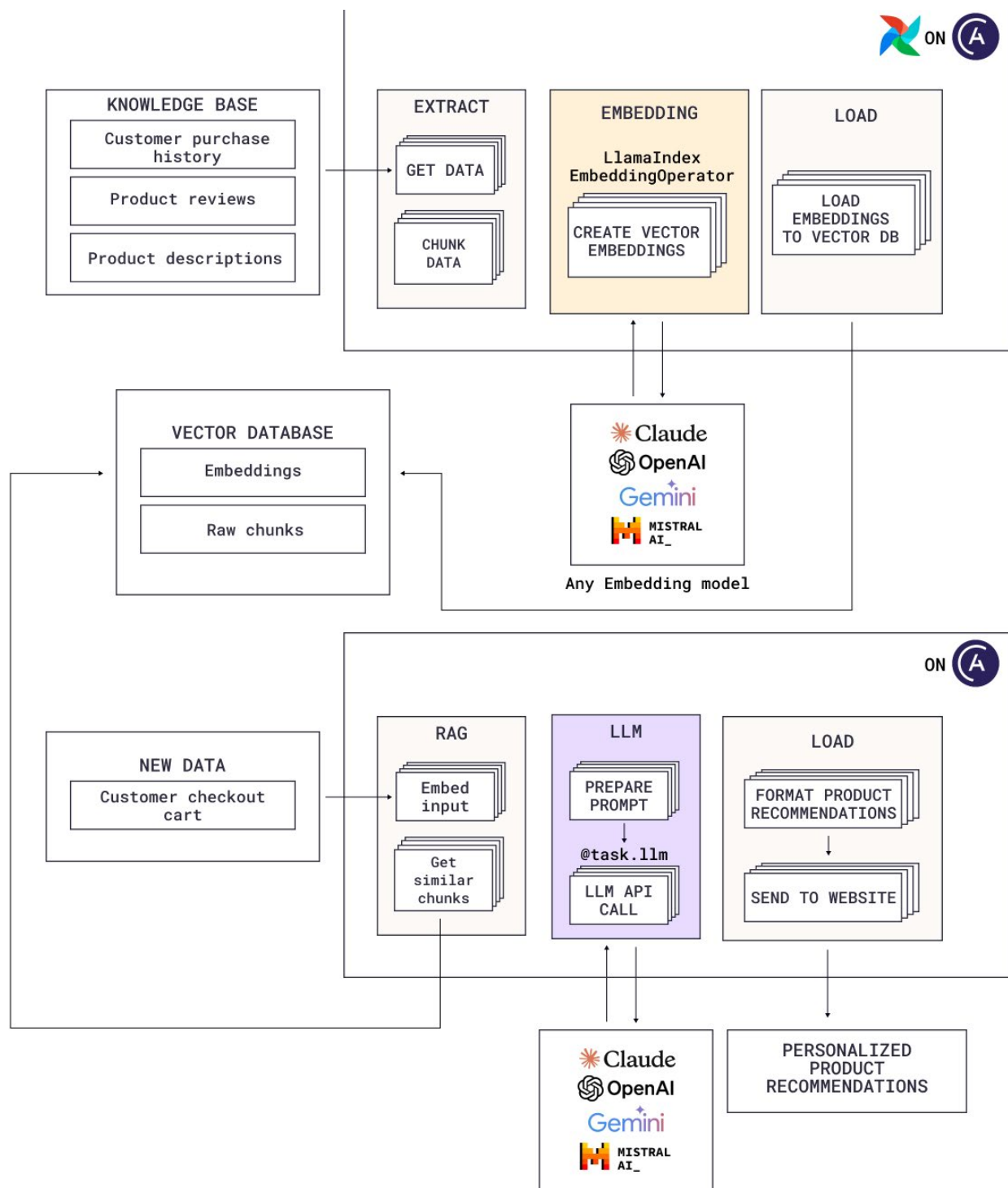


Figure: Example architecture for a RAG pipeline embedding text from a knowledge base and an LLM orchestration Dag that retrieves content based on embeddings to create personalized product recommendations for customers.

The retrieval Dag runs based on an event-driven schedule as soon as any customer has made a new purchase.

1. First, the purchased product's descriptions are embedded by the same model to be used as the query input.
2. A task upstream of the LLM runs the similarity search, retrieving similar information to the newly purchased products.
3. The retrieved items are formatted into context and passed to the LLM alongside the customer's purchase.
4. The LLM generates a short recommendation message based on the retrieved items, which is published to the customer's account dashboard.

For a hands-on introduction to building RAG pipelines with Airflow, see the Orchestrating Workflows for GenAI Applications course on DeepLearning.AI (<https://www.deeplearning.ai/courses/orchestrating-workflows-for-genai-applications>).

AI evals with Airflow

In previous sections you've already encountered two patterns of evaluating AI-generated assets: AI-as-a-judge as well as human-in-the-loop. The section on self-improving agents showed how human input can be gathered systematically in decision traces and made available to the agent as context graphs.

Both of those are evaluation signals. This chapter covers the orchestration pattern around all the other signals: where they come from, when they show up, and how to get them into one place. For more information on how to structure an evaluation pipeline, see the AI evals with Apache Airflow® guide (<https://www.astronomer.io/docs/learn/ai-orchestration-evals>).

Evaluation signals arrive at different times

Evaluating AI can be difficult because the signals that tell you whether your AI is any good appear at different moments, in different systems, long after the task that produced the output has finished.

Over time you can get AI eval signals:

- **Downstream in the same Dag:** deterministic checks against a source of truth, plus the agent-in-the-loop and human-in-the-loop steps covered earlier.
- **When the output arrives at the first stakeholder (human or agent):** approvals, rejections, edits, and, if you capture them, the reasons behind the edits.
- **When the output arrives at downstream stakeholders:** customer replies, clicks, conversions, explicit feedback.
- **Over the following months:** renewals, churn, repeat purchases, ticket volume, whether the stakeholder still uses or engages with the AI application.

None of those systems know about each other, and none of them know which Dag run, model setting, or prompt version produced the output they are reacting to.

Gathering these feedback signals on a schedule, often from many disjointed systems, and joining them back to the AI inference that caused them is ETL and a tool-agnostic orchestrator like Airflow is best suited to run your AI eval pipelines.

Evaluation pipelines are worth building even before you orchestrate any AI inside Airflow at all. Chat apps, coding assistants, and agents deployed in microservices all produce downstream effects you can measure, and measuring them is how you find out which use cases are worth moving into a pipeline in the first place.

Gathering the signals

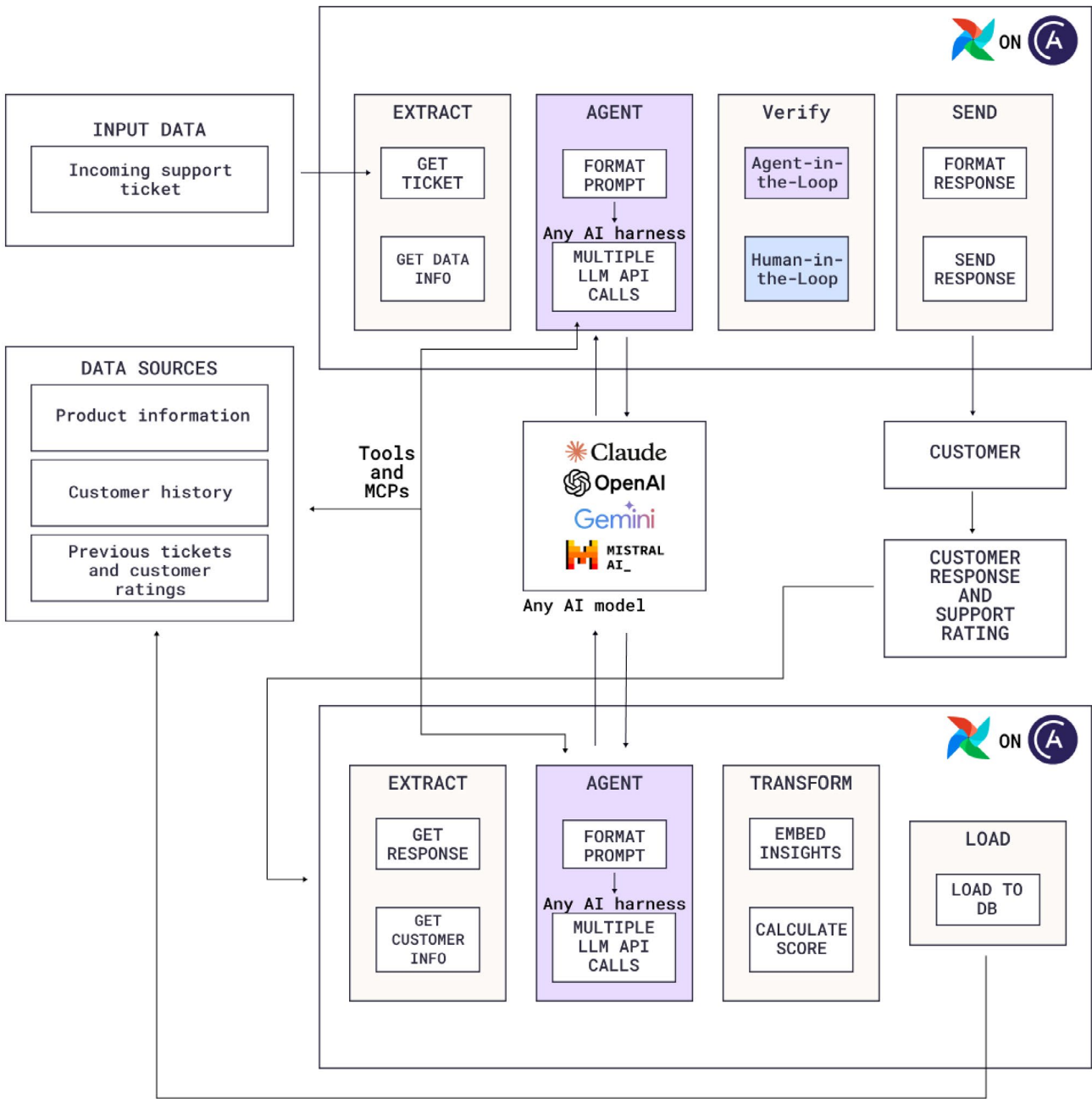


Figure 27: Architecture diagram of a version of the support ticket agent that utilizes three patterns of AI evaluation: AI-as-a-judge, human-in-the-loop as well as gathering of the customer response and support rating.

How an evaluation pipeline is structured depends on the metrics you are gathering. Data like web traffic or numeric customer satisfaction scores is gathered by deterministic pipelines. Often the feedback for AI-generated assets is less structured.

In the example of the support ticket agent, the highest signal comes from the customer's response. In order to get actionable data you can process the response, using AI to score it for sentiment and signals of whether the response was helpful. In more complex support cases it can make sense to use an agent that can evaluate a customer reply in context of previous tickets, customer tier, and other customer signals like product usage. A "This is finally working!" reply shows a different signal at the end of debugging a complex issue with a grateful customer than if sent from an account after a frustrated interaction.

Whatever you gather, version everything. An AI task has a lot of dials: the system prompt, the model, model settings, the tools the agent can use, and the context the upstream pipeline assembled. If several of those change between runs, your evaluation data tells you that something got better or worse, but it can be hard to attribute the changed outcome to changes in the pipeline.

What you measure is what the system optimizes for. Imagine the following: A pipeline drafts outreach messages for the top 100 accounts every night. A few months in, the reps using the AI-generated messages are closing significantly more deals. Success. A year later most of those accounts have churned, and overall new deals are down. What happened? The AI had been promising features that don't exist, support burned out handling the fallout, and the complaints made it to social media.

An evaluation that only counts deals closed will happily lead to the system optimizing for exactly and only that. Define the outcome you actually want over the full lifespan of the AI-generated data product, including its accuracy and downstream effects, and keep measuring it over time.

Agent harnesses, micro vs macro orchestration

The previous chapters covered patterns of macro orchestration, creating the AI prompt, determining the model and available tools for agents, and controlling when the AI task runs in the context of the larger data ecosystem. Macro orchestration also handles retrying of the AI task (optionally with cached state available through the task state store), as well as dynamically creating many AI tasks in parallel, and processing the AI's output. Macro orchestration is the domain of Airflow.

Within agentic tasks, the agent harness itself performs actions which are often also referred to as orchestration. This micro orchestration happens at the level of individual AI steps controlling tool and API calls by the agent, as well as tracking metadata such as tool use, and token costs.

The previous chapters used the Airflow Common AI provider for micro orchestration, an Airflow provider built on top of Pydantic AI. Because Airflow is tool agnostic, you can switch out the `@task.llm` and `@task.agent` tasks in the previous chapters for a general `@task` using any micro orchestration framework you prefer.

This chapter shows code examples for three of the most commonly used micro orchestration tools at the time of writing: LangChain/LangGraph, Temporal, and CrewAI. They are all based on the support ticket agent example and you can view them as full Dags in the book's companion GitHub repository (<https://github.com/astronomer/ai-orchestration-examples>).

USE LANGCHAIN/LANGGRAPH WITH AIRFLOW

LangChain (<https://docs.langchain.com/>) is a library of LLM integrations and abstractions, and LangGraph is its graph-based framework for defining agents and multi-step workflows on top of those integrations. Together they offer prebuilt agent constructors like `create_react_agent`, a large catalog of tool and integration wrappers, and a graph-based way to define control flow between steps.

```
@task
def generate_ai_response(ticket: dict) -> dict:
    from langchain_core.tools import tool as lc_tool
    from langchain_openai import ChatOpenAI
    from langgraph.prebuilt import create_react_agent

    agent = create_react_agent(
        model=ChatOpenAI(model="gpt-5-mini"),
        tools=[lc_tool(lookup_shipment)],
```

```

        prompt=SYSTEM_PROMPT,
        response_format=TicketResponse,
    )
    result = agent.invoke(
        {"messages": [{"role": "user", "content": build_user_prompt(ticket)}}
    )
    return result["structured_response"].model_dump()

_generate_ai_response = generate_ai_response(_fetch_pending_ticket)

```

USE TEMPORAL WITH AIRFLOW

Temporal (<https://docs.temporal.io/>) is a general-purpose durable execution engine. It lets you create workflows with state persisted automatically so a workflow can resume after a crash without redoing steps that have already completed.

In Airflow 3.3 you can implement durable execution for any task by using the task state store feature (<https://www.astronomer.io/docs/learn/airflow-task-state-store>). For agentic tasks using the `@task.agent` decorator of the Common AI provider you can simply set `durable=True` to switch to durable execution using the task state store under the hood.

In this example you can see the agent loop written out by hand: `call_llm` and `lookup_shipment_activity` are separate activities, each independently retryable with its own timeout, and `TicketResponseWorkflow` loops between them until the model returns a final answer.

Note that this example starts a Temporal `Worker` inside the Airflow task in order to keep it self-contained; in a production setup the worker runs as its own long-lived service that the Airflow task connects to. Because a running workflow stays addressable by its ID, several Airflow tasks can interact with the same long-running agentic process, submitting new information via Temporal signals or reading its current state via queries, provided the workflow defines handlers for them.

```

from temporalio import activity, workflow
from temporalio.client import Client

```

```

from temporalio.worker import UnsandboxedWorkflowRunner, Worker

TEMPORAL_ADDRESS = "temporal:7233"
TEMPORAL_TASK_QUEUE = "support-ticket-queue"

@activity.defn
async def call_llm(messages: list[dict]) -> dict:
    from openai import AsyncOpenAI

    client = AsyncOpenAI()
    completion = await client.chat.completions.parse(
        model="gpt-5-mini",
        messages=messages,
        tools=OPENAI_TOOLS,
        response_format=TicketResponse,
    )
    message = completion.choices[0].message
    return {
        "content": message.content,
        "tool_calls": [
            {
                "id": tc.id,
                "type": "function",
                "function": {
                    "name": tc.function.name,
                    "arguments": tc.function.arguments,
                },
            }
            for tc in message.tool_calls or []
        ],
        "parsed": message.parsed.model_dump() if message.parsed else None,
    }

@activity.defn
async def lookup_shipment_activity(shipment_id: str) -> dict:
    return lookup_shipment(shipment_id)

```

```

@workflow.defn
class TicketResponseWorkflow:
    @workflow.run
    async def run(self, ticket: dict) -> dict:
        messages: list[dict] = [
            {"role": "system", "content": SYSTEM_PROMPT},
            {"role": "user", "content": build_user_prompt(ticket)},
        ]
        while True:
            message = await workflow.execute_activity(
                call_llm,
                messages,
                start_to_close_timeout=timedelta(minutes=2),
            )
            if not message["tool_calls"]:
                return message["parsed"] or json.loads(message["content"])
            messages.append(
                {
                    "role": "assistant",
                    "content": message["content"],
                    "tool_calls": message["tool_calls"],
                }
            )
            for tool_call in message["tool_calls"]:
                arguments = json.loads(tool_call["function"]["arguments"])
                result = await workflow.execute_activity(
                    lookup_shipment_activity,
                    arguments["shipment_id"],
                    start_to_close_timeout=timedelta(seconds=30),
                )
                messages.append(
                    {
                        "role": "tool",
                        "tool_call_id": tool_call["id"],
                        "content": json.dumps(result),
                    }
                )

```

```

# ...

@task
def generate_ai_response(ticket: dict) -> dict:
    async def _run() -> dict:
        client = await asyncio.wait_for(
            Client.connect(TEMPORAL_ADDRESS), timeout=10
        )
        async with Worker(
            client,
            task_queue=TEMPORAL_TASK_QUEUE,
            workflows=[TicketResponseWorkflow],
            activities=[call_llm, lookup_shipment_activity],
            workflow_runner=UnsandboxedWorkflowRunner(),
        ):
            return await client.execute_workflow(
                TicketResponseWorkflow.run,
                ticket,

            id=f"support-ticket-{ticket['ticket_id']}-{uuid.uuid4().hex[:8]}",
                task_queue=TEMPORAL_TASK_QUEUE,
            )

    result = asyncio.run(_run())
    return TicketResponse.model_validate(result).model_dump()

_generate_ai_response = generate_ai_response(_fetch_pending_ticket)

```

USE CREWAI WITH AIRFLOW

CrewAI (<https://docs.crewai.com/>) is a multi-agent framework built around agents defined by a role, goal, and backstory. It provides **Agent** and **Task** primitives, a **Crew** object that coordinates multiple agents on a set of tasks, and structured output via **output_pydantic**.

```
@task
```

```

def generate_ai_response(ticket: dict) -> dict:
    import os

    os.environ.setdefault("CREWAI_DISABLE_TELEMETRY", "true")

    from crewai import Agent, Crew
    from crewai import Task as CrewTask
    from crewai.tools import tool as crew_tool

    @crew_tool("lookup_shipment")
    def lookup_shipment_tool(shipment_id: str) -> dict:
        return lookup_shipment(shipment_id)

    support_agent = Agent(
        role="Support agent",
        goal=SYSTEM_PROMPT,
        backstory=(
            "Experienced customer support agent."
        ),
        llm="openai/gpt-5-mini",
        tools=[lookup_shipment_tool],
    )
    draft = CrewTask(
        description=build_user_prompt(ticket),
        expected_output=(
            "A TicketResponse with summary, response, priority, "
            "confidence_score and suggested_tags."
        ),
        output_pydantic=TicketResponse,
        agent=support_agent,
    )
    crew = Crew(agents=[support_agent], tasks=[draft])
    result = crew.kickoff()
    return result.pydantic.model_dump()

_generate_ai_response = generate_ai_response(_fetch_pending_ticket)

```

PART 2

AIRFLOW FEATURES FOR AI

This chapter contains a selection of Airflow features and provider packages that are highly relevant for AI orchestration pipelines.

- **Common AI provider:** A provider package built to interact with AI Agents.
- **Human-In-The-Loop:** The HITLOperator allows you to add steps in an Airflow Dag that stop and wait for human decisions and input before continuing task execution.
- **Scheduling:** Airflow comes with many scheduling options. This section gives an overview and explains when to choose different types of scheduling.

Of course other Airflow features can be helpful for AI orchestration pipelines as well. For broader coverage of Airflow features, see Astronomer's Airflow 3 Dags: The Definitive Guide (<https://www.astronomer.io/ebooks/airflow-3-dags-definitive-guide>).

Airflow frequently releases new features focussing on AI orchestration. An exciting planned addition for an upcoming Airflow minor release is to allow you to rerun a set of tasks, including LLM and agentic tasks until exit criteria are met (<https://cwiki.apache.org/confluence/spaces/AIRFLOW/pages/440303747/AIP-111+Task+Loops>)

Additionally, an early stage proposal exists for dynamic task generation at Dag runtime: (<https://cwiki.apache.org/confluence/spaces/AIRFLOW/pages/440304823/AIP-113+Dynamic+Task+Groups>).

We recommend joining the Airflow Slack channel to stay current on new features and releases (<https://apache-airflow-slack.herokuapp.com/>).

Common AI provider

The Airflow Common AI provider (<https://airflow.apache.org/docs/apache-airflow-providers-common-ai/stable/index.html>) contains several operators and other modules to add AI-based tasks to your Dags, from simple LLM calls to AI agents with access to Airflow-based tools. This section explains the use of the `@task.llm` and `@task.agent` decorator.

The Common AI provider contains additional decorators and operators, which are explained in depth in the Orchestrate AI tasks with Apache Airflow® and the Common AI provider guide (<https://www.astronomer.io/docs/learn/airflow-common-ai-provider>).

Keep in mind that LLMs and AI agents are not deterministic and therefore tasks using these models are not fully idempotent, meaning that the exact output of the AI task will likely differ in a rerun or backfill of the Dag even if the inputs were the exact same.

What you can achieve when orchestrating AI with Airflow is pipelines where each task produces the same end state, even if the exact content differs. For example an AI summarizing an email might return a slightly different summary on a rerun, but it will still return one summary for one email.

These pipelines where each AI task performs as small of an action as possible, is as atomic as possible, are the most robust way to use AI. Keep all non-AI-based tasks idempotent and, if rerunning or backfilling of AI-based tasks is needed, be aware that outputs can differ.

This chapter was written using the Common AI provider version 0.7.0. If you are using a different version there might have been changes to the described features. See the provider documentation (<https://airflow.apache.org/docs/apache-airflow-providers-common-ai/stable/index.html>) and Common AI provider guide (<https://www.astronomer.io/docs/learn/airflow-common-ai-provider>) for more information.

Installation

This provider package is built on top of Pydantic AI and can be used with any compatible model provider, including OpenAI, Anthropic, Gemini, AWS Bedrock, HuggingFace, and more (<https://pydantic.dev/docs/ai/models/overview/>).

To use the Common AI provider, you need to be at least on Airflow 3.0 and install it by adding it to your `requirements.txt` file. Make sure to add the extra for your model provider, for example `[openai]` and pin the latest version.

```
apache-airflow-providers-common-ai[<your-model-provider-extra>]==<version>
```

Most operators in the Common AI provider depend on modules from the Airflow standard provider, which is pre-installed when using the Astro CLI. Additionally, you'll need to install the Airflow Common SQL provider (<https://airflow.apache.org/docs/apache-airflow-providers-common-sql>) when using `SQLToolset`, `@task.llm_sql`, `@task.llm_schema_compare`, or other features that read database metadata through a `DbApiHook`.

```
apache-airflow-providers-standard==<version>
apache-airflow-providers-common-sql==<version>
```

The Common AI provider uses the same operators and decorators across many model providers by interacting with them through Pydantic AI. You can switch providers by updating the Airflow connection (<https://www.astronomer.io/docs/learn/connections>).

The basic connection has the following format:

```
AIRFLOW_CONN_PYDANTICAI_DEFAULT='{
    "conn_type": "pydanticai",
    "host": "<your_host>",
    "password": "<your_api_key>",
    "extra": {
        "model": "<your_provider>:<your_model>"
    }
}'
```

You can set a default model for the connection by specifying `model` in the connection `extra` field in the format `your_provider:your_model` (for example `anthropic:claude-sonnet-5`). Models can be overridden at the task level, as long as the model is accessible through the provided Airflow connection. See Pydantic AI connection (https://airflow.apache.org/docs/apache-airflow-providers-common-ai/stable/connections/pydantic_ai.html) for more information.

@task.llm

The `@task.llm` decorator and `LLMOperator` make a single turn LLM call and return the model output.

```
from pendulum import duration
from airflow.sdk import task
from pydantic_ai.usage import UsageLimits
from pydantic import BaseModel
from typing import Literal

class MyOutput(BaseModel):
    information_density: Literal["low", "medium", "high"]
    summary: str

@task.llm(
    llm_conn_id="pydanticai_default", # required: your Airflow connection ID
    # Optional parameters see descriptions below
    model_id="<your_provider>:<your_model>",
    system_prompt="You are a helpful AI.",
    output_type=MyOutput,
    agent_params={},
```

```

usage_limits=UsageLimits(total_tokens_limit=4000),
require_approval=True,
approval_timeout=duration(minutes=10),
allow_modifications=False,
serialize_output=False,
)
def my_llm_task(user_context: str) -> str:
    user_prompt = f"Summarize this document: {user_context}" # any transformation
    return user_prompt

my_llm_task(user_context=my_document) # User context, often prepared by upstream
tasks is passed to the LLM task as arguments, the LLM response is pushed to XCom in
the format defined by output_type

```

The string your `@task.llm` callable returns is the user prompt sent to the model. When you use `LLMOperator` directly, the `prompt` argument is required.

Optional parameters:

- `model_id`: Overrides the model in the connection `extra` (format: `<provider>:<model>`).
- `system_prompt`: System instructions loaded before the user prompt.
- `output_type`: Return type for the run. Defaults to `str`. For structured JSON, set a subclass of Pydantic BaseModel (https://pydantic.dev/docs/validation/latest/api/pydantic/base_model/).
- `agent_params`: Extra keyword arguments for the Pydantic AI `Agent` constructor (for example `model_settings`). See the Pydantic AI Agent parameters (<https://pydantic.dev/docs/ai/api/pydantic-ai/agent/#parameters>). Despite performing a single turn LLM call, you can still pass arguments to the agent constructor when using `@task.llm`.
- `usage_limits`: Optional override of default Pydantic AI UsageLimits enforced on the run. By default current Pydantic AI versions set `request_limit=50`, with no other limits. You can define additional limits, for example on token or tool usage (<https://pydantic.dev/docs/ai/api/pydantic-ai/usage/>).
- `require_approval`: When `True`, the task goes into `awaiting_input` mode until a human approves or rejects through the approval UI. Defaults to `False`.

- `approval_timeout`: Time limit to wait for a review when `require_approval` is `True`. When it is exceeded, the task fails. Defaults to `None`.
- `allow_modifications`: When `True` with approval enabled, the reviewer can edit the generated text before approval; that edited value becomes the task result. Defaults to `False`.
- `serialize_output`: When `True` and using an `output_type` that is a Pydantic BaseModel, the model output will be converted into a `dict` before being pushed to XCom.

When `require_approval` is `True`, the task enters `awaiting_input` mode after the model returns its output. In the Airflow UI, open the task instance, select the **Required Action** tab, read the generated text, edit the output if needed, and then click **Approve** or **Reject**.

In the optional approval step, the reviewer can edit the model output before approval when `allow_modifications` is `True`.

The screenshot shows the Airflow web interface for a task instance named 'summarize' (ID: manual__2026-07-22T14:16:40.585550+00:00). The task is in the 'Awaiting Input' state. The 'Required Action' tab is active, showing a long text block of generated content. Below the text, there is an 'output' field with a placeholder '[object Object]' and a note 'Edit the output before approving (optional)'. At the bottom right, there are 'Approve' and 'Reject' buttons.

Figure: Screenshot of the Airflow UI showing the Required Action waiting for human input.

@task.agent

The `@task.agent` decorator and `AgentOperator` let you run a Pydantic AI agent as a task in your Airflow Dag.

```
from datetime import timedelta
from airflow.sdk import task
from pydantic import BaseModel
from pydantic_ai import FunctionToolset, UsageLimits

class MyToolset(FunctionToolset): ... # your custom toolset

class MyOutputClass(BaseModel): ... # your custom output class

@task.agent(
    llm_conn_id="pydanticai_default",
    model_id=None,
    system_prompt="You are a software engineer",
    output_type=MyOutputClass,
    toolsets=[MyToolset()],
    enable_tool_logging=True,
    agent_params={},
    usage_limits=UsageLimits(
        request_limit=15,
        tool_calls_limit=8,
        total_tokens_limit=50_000,
    ),
    durable=False,
    code_mode=False,
    message_history="[]",
    enable_hitl_review=True,
    max_hitl_iterations=3,
    hitl_timeout=timedelta(minutes=5),
    hitl_poll_interval=5.0,
    serialize_output=True,
)
```

```
def my_agentic_task(my_input: str) -> str:
    return f"This is the user prompt! {my_input}"

my_agentic_task(my_input="Say hello!")
```

The string returned by the `@task.agent` decorated function is the user prompt input to the agent. When using the `AgentOperator` directly, providing a user prompt with the `prompt` parameter is required.

The only other mandatory parameter is `llm_conn_id`, the Airflow connection to use for the LLM.

The following optional parameters are available for the `@task.agent` decorator:

- `model_id`: `<your_provider>:<your_model>`, for example `anthropic:claude-sonnet-5`.
- `system_prompt`: The system prompt to use for the agent, which is loaded into context before the user prompt. It is common to give general instructions to the agent in the system prompt about their role and task, as well as information about available tools. Note that instructions given in the prompt are not guaranteed to be followed.
- `output_type`: The format for the output from the agent. You can use primitive types like `str`, `int`, `float`, `bool`, or a subclass of Pydantic `BaseModel` (https://pydantic.dev/docs/validation/latest/api/pydantic/base_model/) for more complex structured outputs. Defaults to `str`. The `output_type` format is enforced, in contrast to instructions in the prompt.
- `toolsets`: The toolsets the agent has access to as a list of `pydantic_ai.toolset` instances (<https://pydantic.dev/docs/ai/api/pydantic-ai/toolsets>). Defaults to `None`.
- `enable_tool_logging`: If `True`, every toolset is wrapped in a `LoggingToolset` that logs tool calls with timing at INFO level and arguments at DEBUG level. Defaults to `True`.
- `agent_params`: Additional keyword arguments passed to the agent constructor. See the Pydantic AI agent documentation (<https://pydantic.dev/docs/ai/api/pydantic-ai/agent/>) for a list of available parameters.
- `durable`: Whether to enable step-level caching of model responses and tool results. Defaults to `False`. Cannot be used with human-in-the-loop review. See Durable execution.
- `code_mode`: Mutually exclusive with `durable=True`. Requires the `code-mode` extra from the Pydantic AI harness (<https://pydantic.dev/docs/ai/harness/code-mode/>).

- `message_history`: “[]” starts a tracked session: the full transcript gets pushed to this task’s XCom under key `message_history` for a follow-up turn to resume. The default, `None`, is a single-turn run with no push.
- `enable_hitl_review`: Whether to enable human-in-the-loop review through an Airflow plugin. Defaults to `False`. Needs Airflow 3.1+ and cannot be used with durable execution.
- `max_hitl_iterations`: The maximum number of iterations of human-in-the-loop review. Defaults to 5.
- `hitl_timeout`: The timeout for the human-in-the-loop review as a `timedelta` object. Defaults to `None`.
- `hitl_poll_interval`: The interval between polling for human-in-the-loop review. Defaults to 10.0.
- `serialize_output`: When `True` and using an `output_type` that is a Pydantic BaseModel, the model output will be converted into a `dict` before being pushed to XCom.

TOOLSETS

AI agents can use tools to perform actions in another system. Tools are typically implemented as functions that can be called by the agent. A toolset is a collection of such functions in a class.

When using the Common AI provider, your agents can use three types of toolsets:

- Pre-built toolsets included in the Common AI provider: `SQLToolset` and `MCPToolset`.
- Use any Airflow hook (<https://www.astronomer.io/docs/learn/what-is-a-hook>) as a toolset by wrapping it in the `HookToolset` class; the hook methods are the tools the agent can call.
- Custom toolsets you can implement yourself by subclassing a `pydantic_ai.toolset` class.

As soon as tools are added to an AI agent, the agent can independently call that tool and perform any action that is possible through the tool. If you are adding tools, especially custom tooling with access to production systems, make sure to scope the access AI agents have to prevent destructive actions like dropping a table or deleting a file. **Never rely on instructions in a prompt** to restrict your agent from performing actions, always enforce access control programmatically.

PRE-BUILT TOOLSETS

The SQLToolset (https://airflow.apache.org/docs/apache-airflow-providers-common-ai/stable/_api/airflow/providers/common/ai/toolsets/sql/index.html) is a curated toolset that gives your agent access to a SQL database, only allowing the following methods: `list_tables`, `get_schema`, `query` (`SELECT` only by default), and `check_query`. It can be used with any SQL database that is supported by the `DbApiHook`.

```
from airflow.providers.common.ai.toolsets.sql import SQLToolset

toolsets=[
    SQLToolset(
        db_conn_id="my_sql_conn",
        allowed_tables=["my_table"], # restrict which tables the agent can access
        # through list_tables and get_schema
        schema="my_schema", # Database schema/namespace for table listing and
        # introspection.
        allow_writes=False, # If True, modify operations are allowed (insert,
        # update, delete). Default: False.
        max_rows=1000 # Maximum number of rows to return from a query. Default:
        50.
    )
],
```

The `allowed_tables` parameter is enforced in Common AI 0.5.0+ using `sqlglot`. This still has some limitations in Common AI 0.6.0: `sqlglot` cannot evaluate data retrieved through functions or `COPY` statements. This means that in some cases an LLM can still query tables outside this list if the name is retrieved through these methods. Common AI 0.7.0 closed these gaps and added an `allowed_functions` parameter.

To be fully safe: for query-level restrictions, use database-level permissions (for example a read-only role with grants limited to specific tables).

Setting `allow_writes=True` allows the agent to perform modify operations (insert, update, delete) on the database. Use at your own risk.

The MCPToolset (https://airflow.apache.org/docs/apache-airflow-providers-common-ai/stable/_api/airflow/providers/common/ai/toolsets/mcp/index.html) is a toolset that gives your agent access to an MCP server. It uses the `MCPHook` to retrieve credentials from an Airflow connection and creates a Pydantic AI MCP server instance (<https://pydantic.dev/docs/ai/api/pydantic-ai/mcp/>).

```
from airflow.providers.common.ai.toolsets.mcp import MCPToolset

toolsets=[MCPToolset(mcp_conn_id="mcp_default")],
```

Set the MCP server connection (<https://airflow.apache.org/docs/apache-airflow-providers-common-ai/stable/connections/mcp.html>) as an Airflow connection. There are two main types of transports for MCP servers:

- **Streamable HTTP:** Uses HTTP to stream responses, recommended for remote servers. If your `host` does not require authentication, you can omit the `password` field.

```
AIRFLOW_CONN_MCP_DEFAULT=' {
    "conn_type": "mcp",
    "host": "http://localhost:3001/mcp",
    "password": "<your_auth_token>",
    "extra": {
        "transport": "http"
    }
}'
```

- **stdio:** Runs the MCP server as a subprocess communicating over stdin/stdout.

```
AIRFLOW_CONN_MCP_DEFAULT=' {
    "conn_type": "mcp",
    "extra": {
        "transport": "stdio",
        "command": "uvx",
        "args": ["mcp_server"]
    }
}'
```

USING AIRFLOW HOOKS AS TOOLSETS

You can use any Airflow hook (<https://www.astronomer.io/docs/learn/what-is-a-hook>) as a toolset by wrapping it in the `HookToolset` class. The hook methods are the tools the agent can call. The `allowed_methods` parameter is required to make methods available to the agent, auto-discovery is intentionally disabled for safety purposes.

```
from airflow.providers.common.ai.toolsets.hook import HookToolset

toolsets=[
    HookToolset(
        hook=MyHook(conn_id="my_conn"),
        allowed_methods=["method1", "method2"],
        tool_name_prefix="my_prefix_", # optional, default: ""
    )
],
```

You can explore available hooks in the Airflow registry (<https://airflow.apache.org/registry/>).

HUMAN-IN-THE-LOOP REVIEW

Human-in-the-loop review lets you iterate on the output of an agentic task by reviewing the agent output and providing feedback. An Airflow plugin adds a **HITL Review** tab to the task instance page in the Airflow UI. This is different from human-in-the-loop operators, which surface decisions under **Required Actions**.

```
from airflow.sdk import task
from datetime import timedelta

@task.agent(
    enable_hitl_review=True,
    max_hitl_iterations=5,
    hitl_timeout=timedelta(minutes=5),
    hitl_poll_interval=10.0,
)
```

```
def my_agentic_task(my_input: str) -> str:
    return f"This is the user prompt! {my_input}"

my_agentic_task(my_input="Say hello!")
```

On each iteration you have three options:

- **Approve:** The task succeeds with the current assistant output and the HITL review ends.
- Provide feedback to the agent and click **Send**: The agent will take the feedback as its user prompt and regenerate the output. The maximum number of iterations is controlled by the `max_hitl_iterations` parameter.
- **Reject:** The task fails and the next iteration is not started.

The `hitl_timeout` parameter provides an optional time limit for the entire HITL review phase, that is, all review rounds combined. When that limit is exceeded, the task fails. The `hitl_poll_interval` parameter is the number of seconds to wait between polls for the reviewer action on the human-action XCom key.

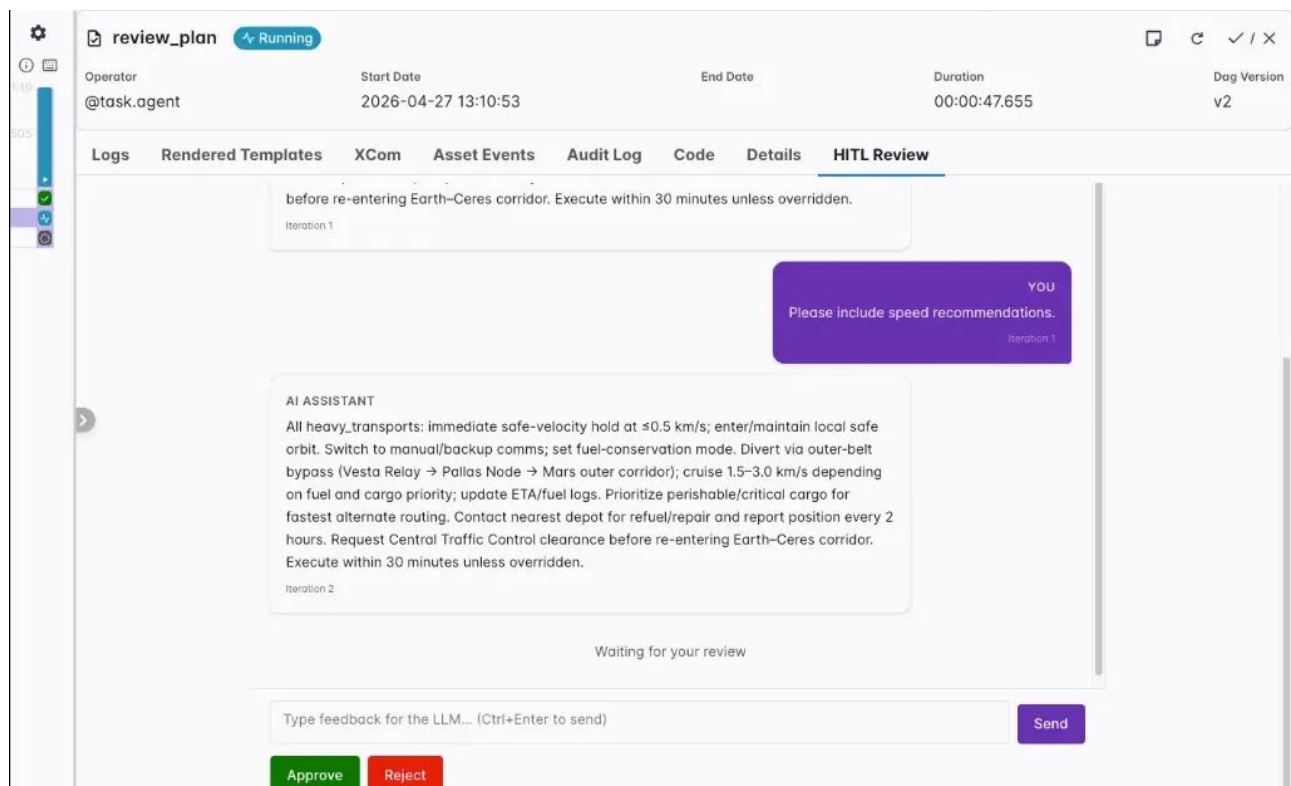


Figure: A screenshot of the HITL Review tab which is added for `@task.agent` tasks that enable human-in-the-loop review.

DURABLE EXECUTION

The `durable` parameter allows you to enable step-level caching of model responses and tool results. When a task is retried, steps that have already finished do not run again; the task reads their results from the cache instead.

As of Common AI provider 0.7.0+, the cached values are stored using the Airflow 3.3 task state store feature (<https://www.astronomer.io/docs/learn/airflow-task-state-store>).

To run an agent in durable mode set `durable=True`.

```
from airflow.sdk import task

@task.agent(
    durable=True,
)
```

When a durable agent retries and uses previously cached results you'll see a log message like this:

```
[2026-04-26 20:37:33] INFO - Durable: replayed 3 cached steps (2 model, 1 tool),
executed 4 new steps (2 model, 2 tool)
```

Note that you can only use `durable=True` if `enable_hitl_review=False`.

OUTPUT TYPE

Pass `output_type` to set the format of the agent output. It can be a primitive type like `str`, `int`, `float`, `bool`, or a subclass of Pydantic BaseModel (https://pydantic.dev/docs/validation/latest/api/pydantic/base_model) for more complex structured outputs. Use the `Field` class to add descriptions to the fields for the agent to use.

```

from typing import Literal

from pydantic import BaseModel, Field

class MyOutputClass(BaseModel):
    my_field_one: Literal["A", "B", "C", "D", "F"] = Field(
        description=(
            "Letter grade for xyz..."
        )
    )
    my_field_two: str = Field(
        description="String that describes xyz..."
    )
    my_field_three: list[str] = Field(
        description=(
            "List of strings that describe xyz..."
        )
    )
    my_field_four: int = Field(
        description=(
            "Number that describes xyz..."
        )
    )

```

Using the above `MyOutputClass` as the `output_type` parameter the agentic task will always produce a JSON output with the fields and their values.

```

{
    "my_field_one": "A",
    "my_field_two": "In summary...",
    "my_field_three": ["In detail...", "In detail..."],
    "my_field_four": 10
}

```

Human-In-The-Loop

Human-in-the-loop workflows are processes that require human intervention. This is especially relevant in AI orchestration, where you might need to approve or reject an AI-generated asset or make a decision based on AI output.

Airflow 3.1 contains a set of operators to create tasks that will wait for human input, either in the Airflow UI or through the Airflow REST API. These operators are part of the Airflow standard provider.

This chapter was written using the Airflow standard provider version 1.16.0 and Airflow 3.3. If you are using different versions there might have been changes to the described features. See the changelog for more information (<https://airflow.apache.org/docs/apache-airflow-providers-standard/stable/changelog.html>).

With the **HITLOperator** you can display information, let the user choose one or more from a list of options and accept additional input via a form based on Airflow params. In Airflow 3.1 and 3.2, human-in-the-loop operators are implemented as deferrable operators, meaning that they run an asynchronous trigger process in the Triggerer component while waiting for the human input. In Airflow 3.3+, human-in-the-loop operators enter the **awaiting_input** task state while waiting for a required action to receive a response. The **awaiting_input** state does not require a running triggerer or worker.

Two parameters are mandatory when instantiating the **HITLOperator**:

- **subject** (required): The subject of the templated action which is displayed as the title in the **Required Actions** tab. This field is templatable, which means you can use Jinja templates to render information at runtime, including information computed by an upstream task.
- **options** (required): A list of strings that are rendered as response options at the bottom of the required action form.

There are also several optional parameters that you can use to further configure the behavior of the `HITLOperator`:

- `body`: The main text body. This field is templatable as well and supports markdown formatting.
- `defaults`: Optionally, you can provide a list of one or more options that are selected by default if the task times out before a human responds. All default options need to be in the `options` list.
- `multiple`: If set to `True`, the user can select multiple options. Default is `False`.
- `params`: With these parameters you can create form fields for any user input to the required action using Airflow params. Note that not all param functionality is supported for human-in-the-loop operators.
- `response_timeout`: This parameter times out the task after a specified duration provided as a `datetime.timedelta` or `pendulum.duration` object. Default is `None`. After the timeout has been reached the behavior depends on whether you provided a `defaults` list or not:
 - If you provided a `defaults` list, the default(s) is/are chosen as the response and the task succeeds.
 - If you did not provide a `defaults` list, the task fails.
- `assigned_users`: A list of all users as `HITLUser` dictionaries who are allowed to respond to the required action.
- `notifiers`: A list of notifiers of which to execute the `.notify()` method when the task starts running.

The following example shows a simple use of the `HITLOperator` where the output of an upstream task is fetched from XComs and the user is given 3 response options to choose from. One additional input field for the `expense_amount` is rendered in the Airflow UI using an Airflow param. After 5 minutes (`response_timeout`) the task times out and chooses the default option: `ACH Transfer` as the response and `10000` as the expense amount.

The downstream `print_result` task prints out the information pushed to XComs by the `HITLOperator` task: the `chosen_options` and the `params_input` values.

```

from airflow.providers.standard.operators.hitl import HITLOperator
from airflow.sdk import dag, task, chain, Param

@dag
def HITLOperator_syntax_example():

    @task
    def upstream_task():
        return "Review expense report and approve vendor payment method."

    _upstream_task = upstream_task()

    _hitl_task = HITLOperator(
        task_id="hitl_task",
        subject="Expense Approval Required", # templatable
        body="{{ ti.xcom_pull(task_ids='upstream_task') }}", # templatable
        options=["ACH Transfer", "Wire Transfer", "Corporate Check"],
        defaults=["ACH Transfer"],
        multiple=False, # default: False
        params={
            "expense_amount": Param(
                10000,
                type="number",
            )
        },
        response_timeout=timedelta(minutes=5), # default: None
    )

    @task
    def print_result(hitl_output):
        print(f"Expense amount: ${hitl_output['params_input']['expense_amount']}")
        print(f"Payment method: {hitl_output['chosen_options']}")

    _print_result = print_result(_hitl_task.output)
    chain(_upstream_task, _hitl_task)

HITLOperator_syntax_example()

```

The open required action form is displayed under Browse -> Required Actions.

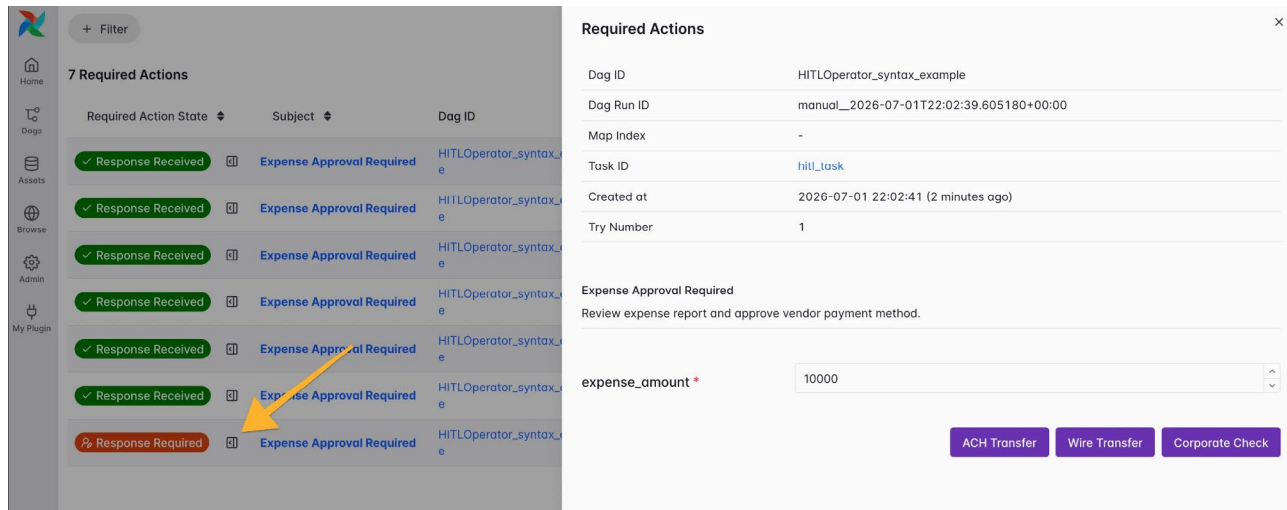


Figure: Screenshot of an open Required Action entry in the Airflow UI.

There are 3 other more specialized human-in-the-loop operators, which are covered in the Human-in-the-loop workflows with Airflow guide (<https://www.astronomer.io/docs/learn/airflow-human-in-the-loop>).

Dynamic Task Mapping

Many AI workflows process sets of similar data, for example analysing many customer tickets to prioritize roadmap items or generating copy for several products. The dynamic task mapping feature in Airflow allows you to create a variable number of parallel “copies” of a task processing similar data. The number of copies can change at runtime based on upstream input, so, for example, you could process 50 customer tickets in one run and 1000 in the next.

Airflow tasks have two main methods available to implement dynamic task mapping. For the task you want to map, you must pass all operator parameters through one of the following functions.

- `.expand()`: This method passes the parameters that you want to map. A separate parallel task is created for each input. For some instances of mapping over multiple parameters, `.expand_kwargs()` is used instead.
- `.partial()`: This method passes any parameters that remain constant across all mapped tasks which are generated by `.expand()`.

In the following example, the add task uses both, `.partial()` and `.expand()`, to dynamically generate three task runs.

Example using the `@task` decorator:

```
# from airflow.sdk import task

@task(
    # optionally, you can set a custom index to display in the UI
    map_index_template="{{ my_custom_map_index }}"
)
def add(x: int, y: int):

    # get the current context and define the custom map index variable
    from airflow.sdk import get_current_context

    context = get_current_context()
    context["my_custom_map_index"] = "Input x=" + str(x)

    return x + y

added_values = add.partial(y=10).expand(x=[1, 2, 3])
```

See the dynamic task mapping guide for more information: <https://www.astronomer.io/docs/learn/dynamic-tasks>.

Scheduling

Airflow has many options for scheduling Dags, much more than fancy cron! The table below gives an overview of the scheduling options.

A good start for most use cases is to run initial dags based on cron schedules and run subsequent dags that depend on the output of earlier Dags on a data-aware schedule, as soon as the upstream dags have completed successfully.

Name	Explanation	Example
Data-aware scheduling	Data-aware scheduling refers to scheduling a Dag based on updates to one or more Assets. In basic data-aware scheduling, a Dag runs based on updates to one singular Asset. Advanced data-aware scheduling refers to using conditional Asset schedules and/or combining time and Asset schedules. A special case of data-aware scheduling is event-driven scheduling (see below). Use when your dag should run based on updates to a (set of) Airflow Asset(s) in your Airflow environment(s). Updates to Assets can occur when tasks complete successfully, when manual updates are created in the Airflow UI, or via a call to the Airflow REST API.	https://www.astronomer.io/docs/learn/airflow-datasets
Event-driven scheduling	Event-driven scheduling, which is new in Airflow 3, allows users to schedule pipelines whenever a message appears in a message queue like Apache Kafka. It enables users to trigger a dag run at any time, dependent on events external to Airflow that post a message in a message queue. Use when your dag should run based on detecting a message in a message queue (poll pattern).	https://www.astronomer.io/docs/learn/airflow-event-driven-scheduling
Trigger via API	Airflow dags can be triggered from within any external system by sending a POST request to the Airflow REST API. Use when your Dag should run based on an event in an external system using a push pattern.	https://airflow.apache.org/docs/apache-airflow/stable/stable-rest-api-ref.html

Name	Explanation	Example
cron	Cron is a simple time-based job scheduling package for use directly in the command-line interface (CLI). A good resource for writing cron schedules is crontab guru. For many common cron schedules, Airflow has built-in shorthand notation. To run a Dag every day at midnight, for example, you can use “@daily” instead of “0 0 * * *”. For a full list, see the Airflow documentation Use when your Dag should run regularly at the same point(s) in time. For more complex point-in-time schedules that can’t be represented by a cron string, use timetables (discussed later in this table).	<code>schedule=</code> <code>“0 0 * * *”</code> <code>schedule=</code> <code>“@daily”</code>
Timedelta	A timedelta schedule runs a Dag on a regular cadence, such as every 35 minutes and 30 seconds, after it is first activated. You implement it by providing a timedelta object to the schedule parameter and specifying the desired cadence. Use when your Dag should run regularly every X time period after it is activated.	<code>schedule=</code> <code>timedelta(</code> <code>minutes=35,</code> <code>seconds=30</code> <code>)</code>
Timetable	Timetables are Airflow classes that can achieve more-complex time-based schedules. Airflow comes with a couple of built-in timetables, and users can define their own. Use when your Dag should run on a time-based schedule that cannot be represented by a cron or timedelta schedule.	https://airflow.apache.org/docs/apache-airflow/stable/authoring-and-scheduling/timetable.html
Run continuously	The <code>ContinuousTimetable</code> starts a new dag run as soon as the old one finishes. Aside from defining the schedule parameter, you need to set <code>max_active_runs=1</code> when using this timetable. Use when your Dag should always have one active run. Caution: This feature is not intended for streaming processes.	<code>schedule=</code> <code>“@continuous”</code>

NEXT STEPS

No matter your use case, the patterns in this book share a core checklist for implementing AI orchestration pipelines:

- Choose the right pattern for your use case. Plenty of use cases only need one LLM call with a structured output class, surrounded by deterministic tasks.
- Add guardrails. Scoped Airflow connections limit what an agent can do; `output_type` limits the structure of what it returns.
- Keep AI tasks as atomic as possible.
- Add a human review step for high-stakes AI actions and outputs.
- Record what your reviewers decide, as well as their reasoning and feedback, and make this information available to your model in future runs.

Where to start depends on the current state of your data. If your data pipelines are in good shape but nothing downstream of your models is measured, start with AI evals. You can do that before you orchestrate a single model call inside Airflow, and it tells you which use cases are worth building out. Next, start moving your model API call into your pipelines for batch and event-driven use cases with an `@task.llm` task or another AI harness. Agents, multi-agent patterns, and Dag-as-a-Tool follow for more complex use cases.

RESOURCES

CODE AND HANDS-ON

AI orchestration examples repository, the companion repo for this book:

<https://github.com/astronomer/ai-orchestration-examples>

Start a free Astro trial to get access to Otto, Astronomer's data engineering agent:

<https://www.astronomer.io/lp/signup/>

Orchestrating Workflows for GenAI Applications, free course on DeepLearning.AI:

<https://www.deeplearning.ai/courses/orchestrating-workflows-for-genai-applications>

EBOOKS

AI Context Engineering with Apache Airflow®:

<https://www.astronomer.io/ebooks/ai-context-engineering-with-apache-airflow>

Airflow 3 Dags: The Definitive Guide:

<https://www.astronomer.io/ebooks/airflow-3-dags-definitive-guide>

Apache Airflow® 3 Best Practices for ETL and ELT Pipelines:

<https://www.astronomer.io/ebooks/apache-airflow-3-best-practices-etl-elt-pipelines>

RELEVANT GUIDES AND DOCUMENTATION

Orchestrate AI tasks with Apache Airflow® and the Common AI provider:

<https://www.astronomer.io/docs/learn/airflow-common-ai-provider>

AI evals with Airflow:

<https://www.astronomer.io/docs/learn/ai-orchestration-evals>

Human-in-the-loop workflows with Airflow:

<https://www.astronomer.io/docs/learn/airflow-human-in-the-loop>

Use Otto to investigate Dag failures and PR a fix:

<https://www.astronomer.io/docs/learn/airflow-3/airflow-otto-rca-auto-fix>

Data engineering with AI:

<https://www.astronomer.io/docs/learn/local-data-engineering-with-ai-overview>

Common AI provider documentation:

<https://airflow.apache.org/docs/apache-airflow-providers-common-ai/stable/index.html>

Airflow registry:

<https://airflow.apache.org/registry/>

