How Netflix Orchestrates Millions of Workflow Jobs with Maestro

BYTEBYTEGO
APR 15, 2025

WorkOS + MCP: Authentication for AI Agents (Sponsored)

Don’t reinvent auth for AI agents. WorkOS AuthKit delivers SSO, MFA, and user management with modern APIs and minimal config.

Gain confidence that MCP servers and AI agents stay within defined permissions, using scoped OAuth access.

Disclaimer: The details in this post have been derived from the articles written by the Netflix engineering team. All credit for the technical details goes to the Netflix Engineering Team. The links to the original articles and videos are present in the references section at the end of the post. We've attempted to analyze the details and provide our input about them. If you find any inaccuracies or omissions, please leave a comment, and we will do our best to fix them.

Netflix originally used an orchestrator named Meson to manage its growing number of data and machine learning (ML) workflows.

For several years, Meson served the company well. It was responsible for scheduling and managing a large volume of tasks, approximately 70,000 workflows and 500,000 daily jobs. These jobs supported various business functions, including data transformation, recommendation models, A/B tests, and other ML-based decision systems.

However, as Netflix's use of data grew, so did the pressure on Meson. The system began to show signs of strain, especially during periods of peak traffic, such as midnight UTC, when a large number of workflows were typically triggered. During these times, the orchestrator experienced slowdowns that led to increased operational overhead. Engineers on call had to closely monitor the system, especially during off-hours, to ensure it didn't fail under the load.

A key limitation of Meson was its architecture. It was built using a single-leader model, which meant there was one main node responsible for coordinating all activity. Although it had high availability, the team had to vertically scale (i.e., upgrade to more powerful machines) to keep up with the growing demand. Eventually, they approached the limits of what AWS instance types could offer, making it clear that vertical scaling was not a sustainable long-term solution.

At the same time, the rate of growth was dramatic. The number of workflows in the system was doubling year over year, which added to the urgency of finding a better solution.

This need led to the creation of Maestro: a next-generation workflow orchestrator designed to overcome Meson's bottlenecks.

Maestro was built with a distributed, scalable architecture from the ground up, capable of handling massive volumes of workflows and jobs while maintaining high reliability and low operational overhead. It was also designed to support a wide range of users and use cases, from engineers and data scientists to business analysts and content producers, making it a more versatile platform for the future of data and ML at Netflix.

In this article, we'll look at how Netflix designed Maestro and the challenges they faced along the way.

AI-assisted coding is faster. But is it safe? (Sponsored)

Tools like Cursor, Co-Pilot, and Claude are helping developers ship code faster than ever.

But with up to 40% of AI-generated code containing vulnerabilities—and 75% of developers believing it’s safer than human-written code—the real risk is what you can’t see.

In this guide, we explore how engineering teams can move quickly without compromising security or standards.

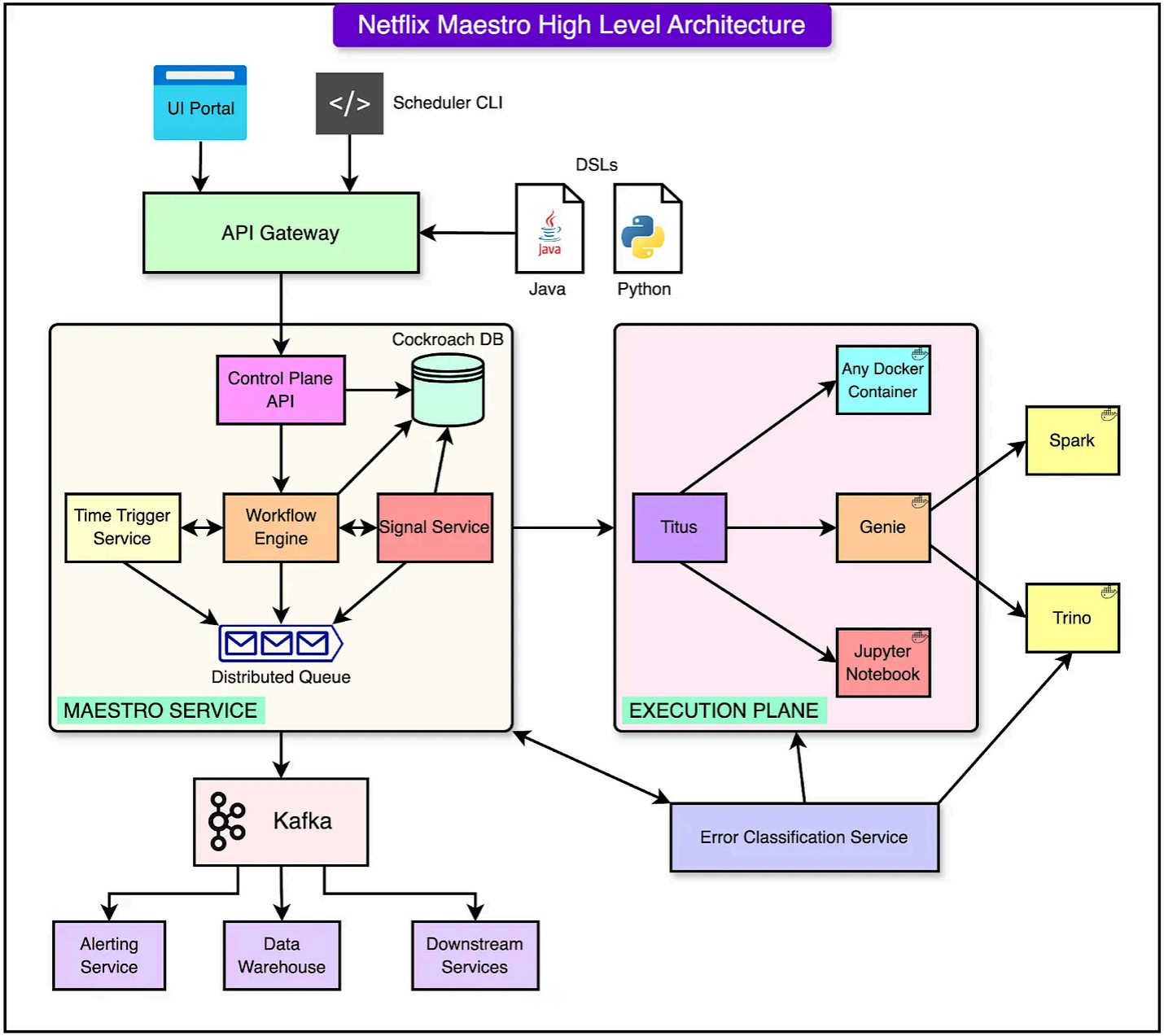
Learn how to set up automated guardrails, enforce service ownership, and catch risks before they spread—no bottlenecks required.

Maestro Architecture

At the heart of Maestro's design is a microservices-based architecture that allows the system to scale efficiently.

WORKFLOW ENGINE, SIGNAL SERVICE, TIME TRIGGER SERVICE

The system is broken down into three primary services, each responsible for a different part of the workflow orchestration. The diagram below shows this architecture in detail:


Netflix Maestro High Level Architecture
UI Portal
</>
Scheduler CLI
DSLs
API Gateway
Java
Python
Cockroach DB
Control Plane API
Time Trigger Service
Workflow Engine
Signal Service
Distributed Queue
MAESTRO SERVICE
Any Docker Container
Titus
Genie
Jupyter Notebook
Spark
Trino
EXECUTION PLANE
Kafka
Error Classification Service
Alerting Service
Data Warehouse
Downstream Services

Workflow Engine

The Workflow Engine is the core component of Maestro. It manages the full lifecycle of workflows from the initial definition to step-by-step execution and completion.

In Maestro, a workflow is represented as a Directed Acyclic Graph (DAG) made up of individual units of work called steps (also referred to as jobs). These steps can include metadata, dependencies, runtime parameters, conditions, and branching logic.

A few key responsibilities of the Workflow Engine include:

- Managing workflow definitions, including versioning and metadata.
- Executing workflows by managing step instances, which are the runtime versions of each step.
- Supporting advanced data patterns like foreach loops, conditional branches, and subworkflows.
- Allowing steps to be reused across workflows through step templates.
- Enabling dynamic workflows by supporting code-based parameter injection through a secure custom expression language called SEL (Simple Expression Language).
- Providing a timeline view of workflow state changes to help with debugging and tracking.

Under the hood, the engine uses Netflix's open-source Conductor library to manage the workflow state machine.

Time-Based Scheduling

Maestro supports workflows that need to be triggered at regular time intervals through the time-based scheduling service. Users can define when workflows should run using cron expressions or predefined time intervals such as hourly or weekly schedules.

This service is lightweight and designed to scale, offering an at-least-once triggering guarantee.

However, the Maestro engine includes logic to deduplicate triggers, which means that even if the scheduler tries to trigger a workflow multiple times, it ensures that the workflow is only executed once. This provides exact-once execution in practice.

Signal Service

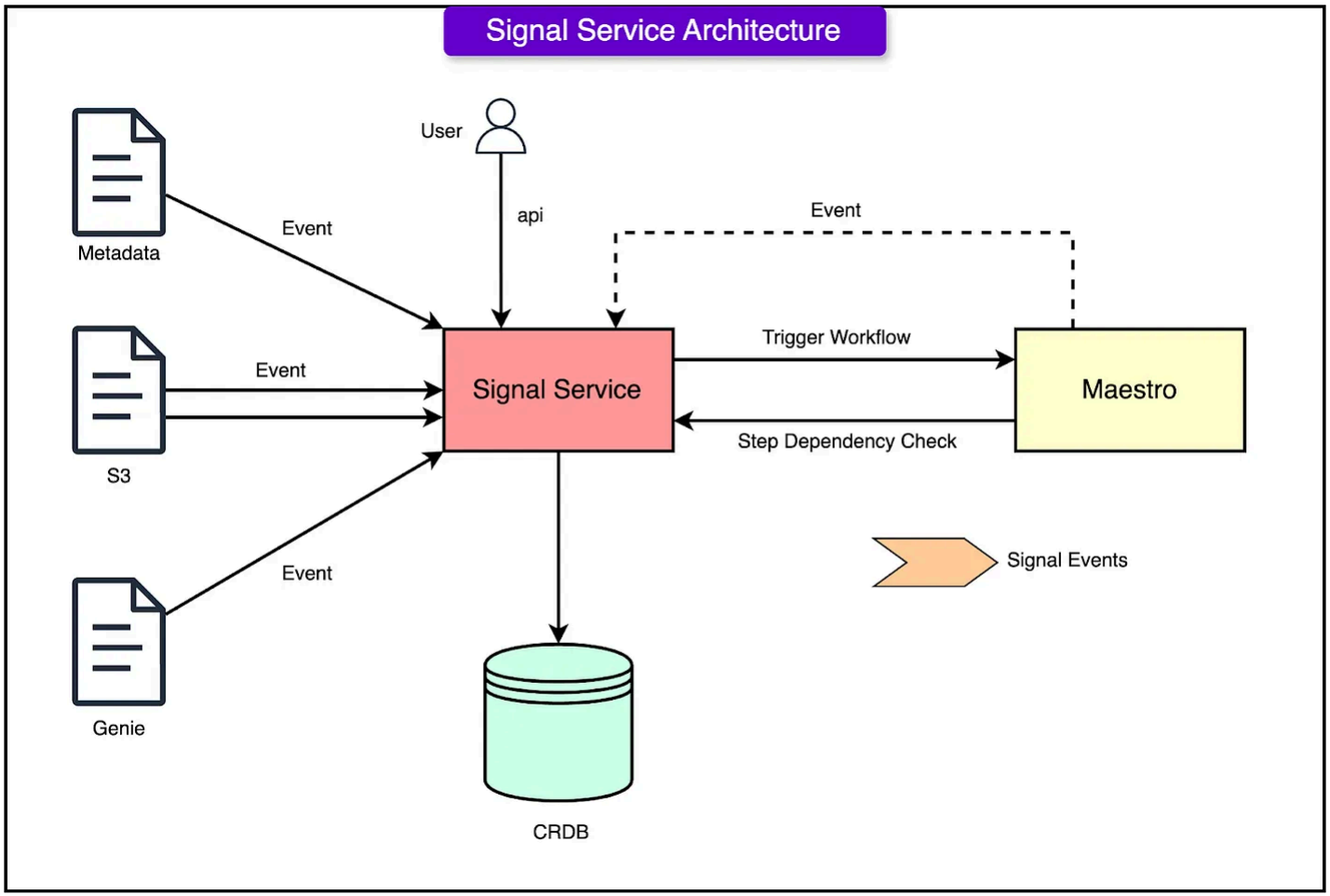
Time-based scheduling is especially useful for recurring tasks, but it may not always be the most efficient method. For instance, running a workflow at midnight might not make sense if the data it depends on isn't ready yet. To address that, Maestro includes another option for triggering workflows: the signal service.

The signal service provides support for event-driven orchestration, which complements time-based scheduling by triggering workflows or steps based on real-time events instead of a fixed time.

A signal is essentially a small message that carries information (such as metadata or parameter values) related to a triggering event. This service plays a crucial role in enabling conditional, responsive execution. Signals can come from:

- Internal sources, such as other workflows.
- External systems, such as S3 events, data warehouse table updates, or other infrastructure changes.

See the diagram below:



The Signal Service is used in two primary ways:

- Triggering entire workflows when a signal matches a subscribed condition.

- Gating individual steps within a workflow, where steps can wait for a signal to arrive before they begin execution.

In addition to these capabilities, the Signal Service also tracks signal lineage. This means it can trace which workflows were triggered by which signals, creating a dependency map between upstream and downstream workflows.

Scalability Techniques for Maestro

One of the core strengths of Maestro is its ability to scale horizontally and handle very large workloads without compromising reliability or performance.

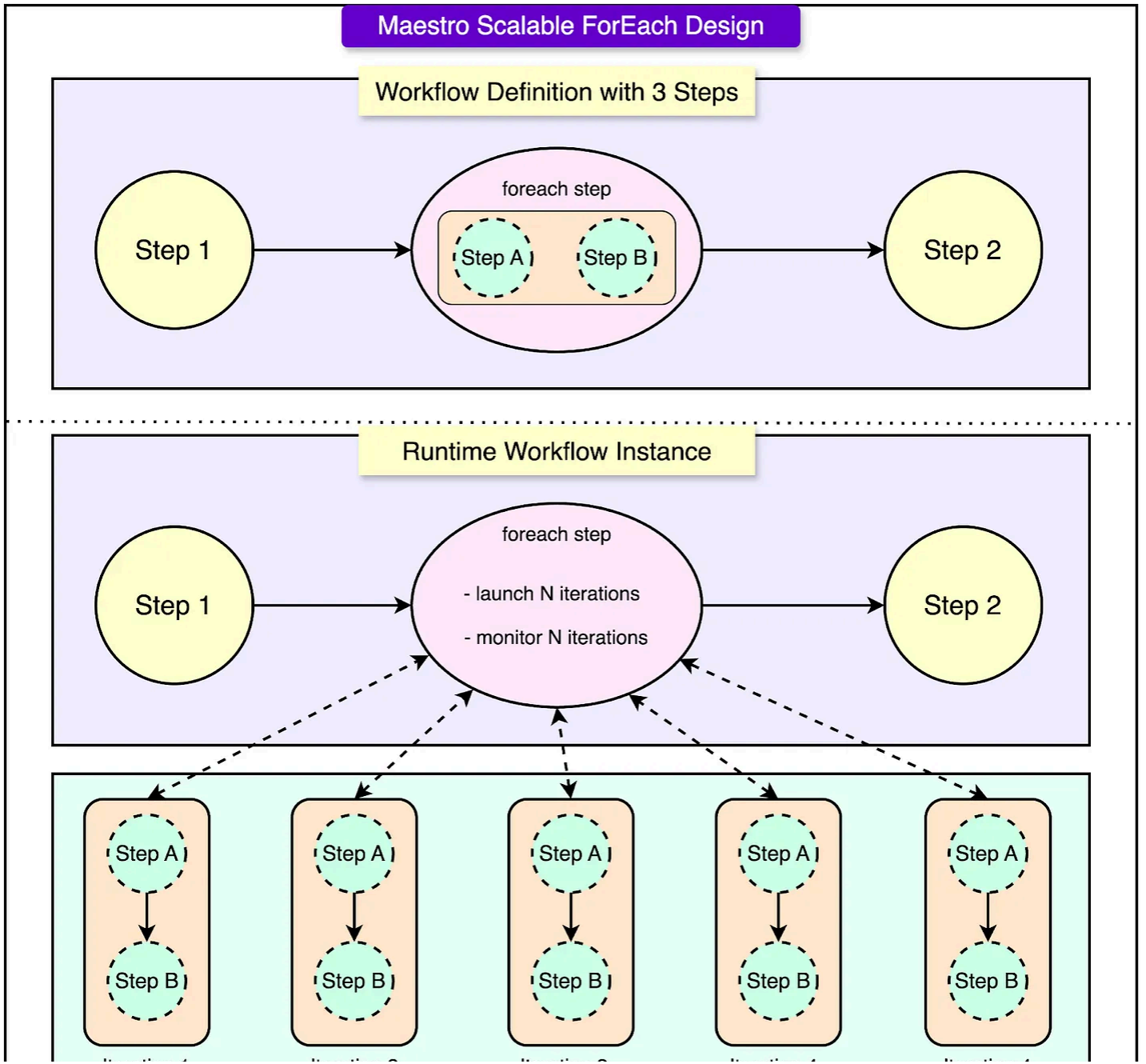
There are a few ways Maestro achieves this:

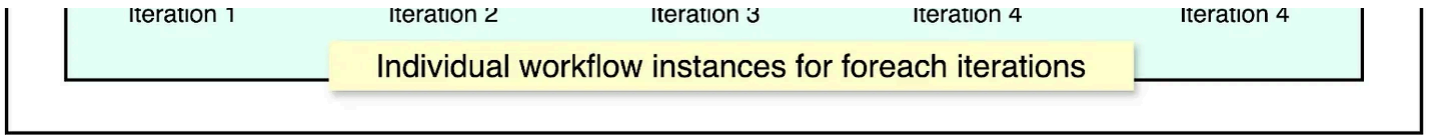
- At the architectural level, all of Maestro's services are built as stateless microservices. This means that any instance of a service can handle any request, and there is no need to store session or workflow data within the service itself. Instead, all state information is stored in an external system. This design makes it easy to scale out services by simply adding more instances.
- To handle communication between components and coordinate workflow execution, Maestro uses distributed queues. These queues decouple the services from each other, so each one can process requests at its own pace. They also allow

for reliable message passing across the system, even when traffic spikes or failures occur in individual components.

- For persistent storage of workflow definitions, step instances, and state data, Maestro uses CockroachDB. CockroachDB is a distributed SQL database that provides strong consistency guarantees. It's well-suited for Maestro because it can scale horizontally and maintain high availability without much operational complexity. This ensures that all workflow and step data is stored reliably, even at a very large scale.
- Another important aspect of Maestro's scalability comes from how it handles workflows with massive numbers of steps. In many use cases, a single workflow might need to run the same job tens or even hundreds of thousands of times. For example, to backfill data over a long historical period or to train models across different parameters. Writing out all of these steps manually in a workflow definition would be impractical and hard to manage.
- Moreover, Maestro supports nested foreach loops, meaning that one foreach block can contain another. This allows users to construct complex, large-scale workflows that execute millions or even billions of individual steps within a single workflow instance, all while remaining manageable and performant.

See the diagram below that shows the forEach design for Maestro.


Maestro Scalable ForEach Design
Workflow Definition with 3 Steps
Step 1
foreach step
Step A
Step B
Step 2
Runtime Workflow Instance
Step 1
foreach step
- launch N iterations
- monitor N iterations
Step 2
Step A
Step B
Step A
Step B
Step A
Step B
Step A
Step B
Step A
Step B



Maestro Execution Abstractions

Maestro is designed to serve a wide range of users from beginner data analysts to advanced engineers. Therefore, it provides multiple execution abstractions to make it easier to define and run workflows.

Some abstractions are as follows:

Step Types

Maestro includes a set of predefined step types that encapsulate common tasks.

These step types represent standard operations, such as running a Spark job, moving data between systems (like from a database to a spreadsheet), or executing a SQL query.

When a user chooses a step type, they only need to supply the required parameters (such as the query string, memory allocation, or table name), and Maestro handles everything else behind the scenes.

For example, if a user selects the "Spark" step type, they can simply provide the Spark SQL query, along with resource requirements, and Maestro will schedule and execute the job accordingly. If the internal logic for Spark jobs ever needs to be updated (such as changing how jobs are submitted), the Maestro team can make that change once, and it will automatically apply to all workflows that use that step type.

Notebook Execution

Maestro also supports direct execution of Jupyter notebooks, which is useful for users who prefer working in a notebook environment or want to run code that has already been developed interactively.

These users can pass their notebooks to Maestro, and the system will schedule and run them as part of a workflow. This is especially helpful for data scientists or analysts who may not want to write workflow logic from scratch but still need to run periodic jobs.

Notebook execution integrates with other workflow steps, allowing users to include notebooks alongside other jobs within a larger DAG. Parameters can also be passed into notebooks dynamically, making them flexible components in a pipeline.

Docker Jobs

For more advanced users who need complete control over the execution environment, Maestro supports running Docker containers. These users can package their custom business logic into a Docker image, define the necessary input parameters, and let Maestro schedule and manage the container's execution.

This option allows users to run any logic, regardless of programming language or dependencies, as long as it is packaged inside a container.

Maestro DSL Interface

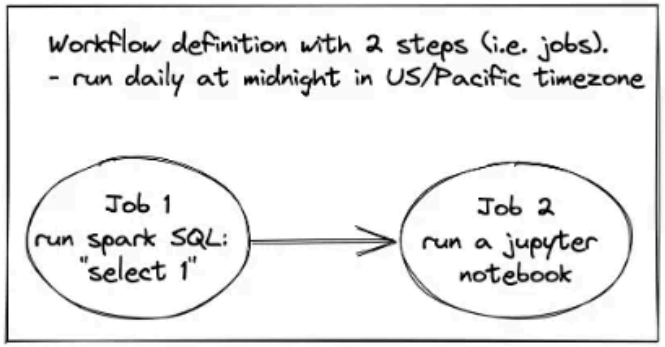
To make it easy for users with different technical backgrounds to define workflows, Maestro supports multiple ways to create and manage them. Users can choose from a set of Domain Specific Languages (DSLs), graphical interfaces, or programmatic APIs, depending on what works best for their needs.

Maestro supports the following DSLs:

- **YAML:** This is the most popular option because it is simple, human-readable, and easy to write. YAML is ideal for defining the structure of workflows, including steps, parameters, and dependencies, without needing to write actual code.
- **Python:** For users who prefer scripting and want to dynamically generate workflows, Python DSL provides a more programmable approach. It's useful when logic needs to be calculated or generated on the fly.

- **Java:** Java DSL is available for engineers who work within Java-based systems or want to integrate workflows more closely with Java applications.

Workflow Definition



YAML DSL

```yaml
Trigger:
  cron: '@daily'
  tz: US/Pacific
Workflow:
  id: demo.pipeline
  jobs:
    - job:
        id: job1
        type: Spark
        spark:
          script: SELECT 1;
    - job:
        id: job2
        type: Notebook
        notebook:
          input_path: s3://path/to/notebook.ipynb
```

Python DSL

```python
wf = Workflow('demo.pipeline') \
 .cron_trigger(cron='0 0 * * *', tz='US/Pacific') \
 .job(
   Spark('job1')
     .spark_script('SELECT 1;')
 ) \
 .job(
   NotebookJob('job2')
 .notebook_input_path('s3://path/to/notebook.ipynb')
 )
```

Java DSL

```java
Workflow wf = Workflow.builder("demo.pipeline")
 .cronTrigger("0 0 * * *", "US/Pacific")
 .job(
   Spark.builder("job1")
     .spark_script("SELECT 1;")
 )
 .job(
   NotebookJob.builder("job2")
     .notebookInputPath("s3://path/to/notebook.ipynb")
 ).build();
```

Source: Netflix Tech Blog

Apart from these DSLs, users can also define workflows in the following ways:

- **API (JSON):** Maestro exposes a set of APIs that allow users to submit workflow definitions directly in JSON format. This is helpful for automation or integration with other systems.
- **Maestro UI**: A graphical user interface is provided for users who prefer not to write code. The UI allows users to visually create workflows by selecting and configuring steps through clicks and forms. This is especially useful for non-engineers or users who want a quick and easy way to build workflows without touching code.
- **Metaflow Integration:** Maestro integrates with Metaflow, a library developed by Netflix that allows users to define DAGs in a Pythonic way. Users can write regular Python code using Metaflow and then run those workflows through Maestro. This bridges the gap between experimentation and production for data scientists.

Parameterized Workflow Example

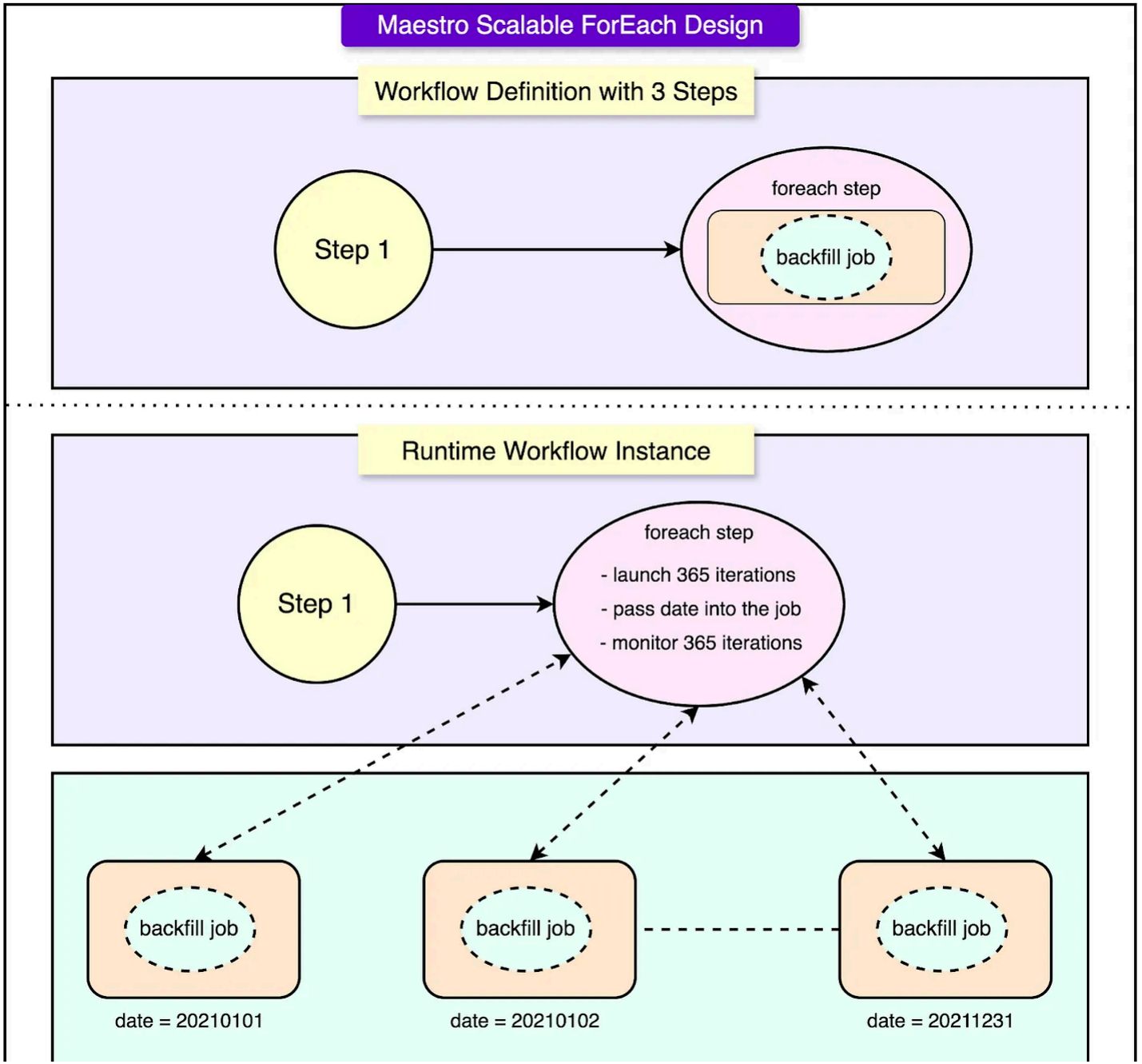
A key feature of Maestro is its support for parameterized workflows, which allow workflows to adapt dynamically at runtime based on input values. This reduces duplication, simplifies maintenance, and enables powerful use cases.

One example is a backfill window, which is commonly used to process historical data across a range of dates.

Here's how it works:

- **Step 1 – Compute the Date Range:** The first step of the workflow uses a simple job (called a NoOp job) to calculate the list of dates that need to be backfilled. For example, it might generate all dates between 2021-01-01 and 2022-01-01. This list of dates is stored as an output parameter.
- **Step 2 – Foreach Loop Over Dates:** The workflow then enters a foreach step that iterates over each date in the list produced by Step 1. For every date, it creates an iteration that runs the same set of jobs, each with a different date value as input.
- **Step 3 – Run the Backfill Job for Each Date:** Inside the foreach, a job is defined to perform the actual backfill operation. In this example, it's a notebook job that runs a Jupyter notebook. The date from the current iteration is passed into the notebook as a parameter, so each run processes data for a specific day.

See the diagram below:


Maestro Scalable ForEach Design
Workflow Definition with 3 Steps
Step 1
foreach step
backfill job
Runtime Workflow Instance
Step 1
foreach step
- launch 365 iterations
- pass date into the job
- monitor 365 iterations
backfill job
backfill job
backfill job
date = 20210101
date = 20210102
date = 20211231

This setup allows users to create a single workflow definition that can dynamically expand into hundreds or thousands of job runs, each tailored to a different date, without having to manually write out every step.

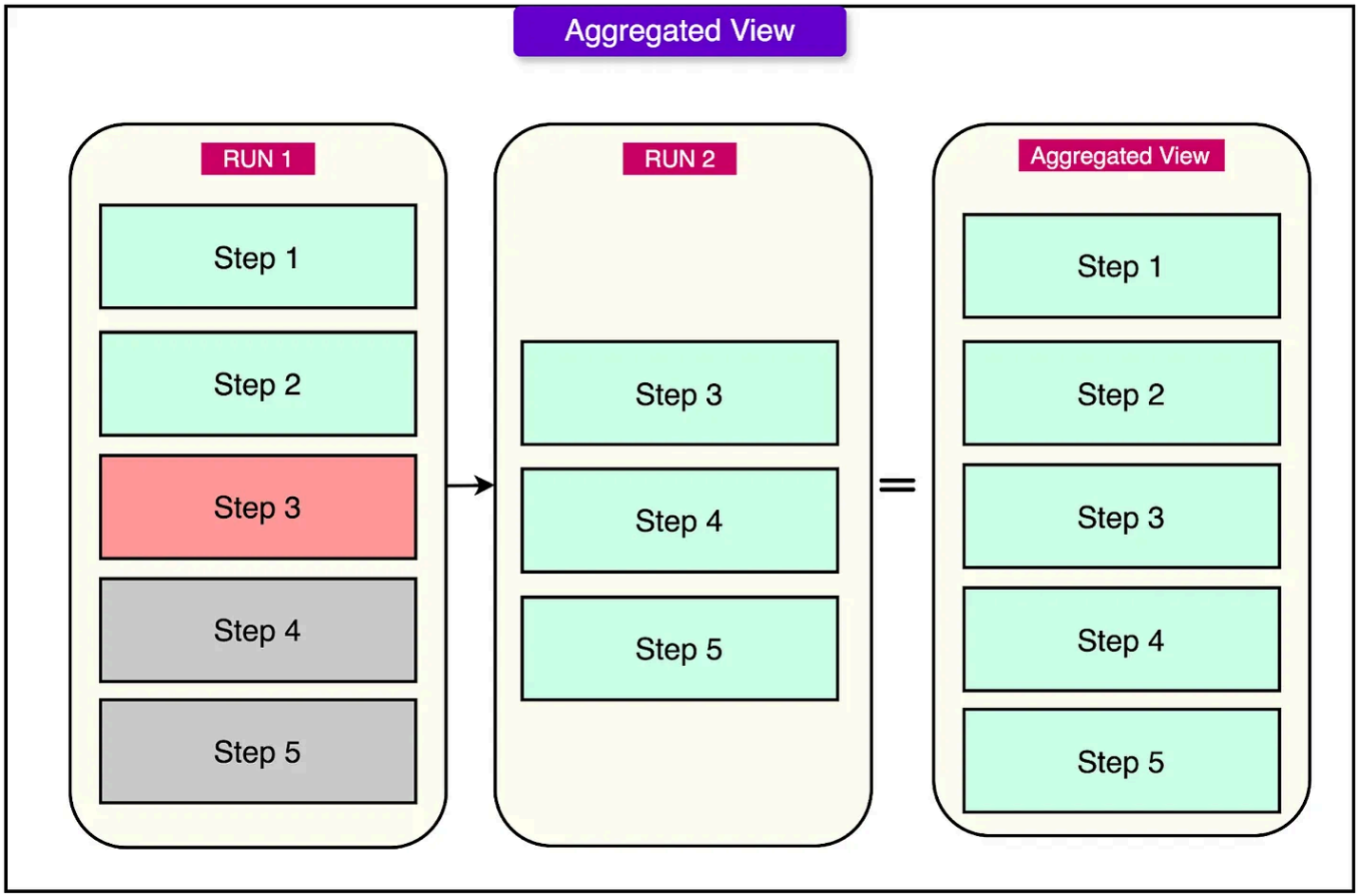
Aggregated View and Rollup

When dealing with large-scale workflows, especially those that include subworkflows, foreach loops, or deeply nested execution paths, it can become difficult for users to keep track of what's happening during and after execution. To solve this, Maestro provides two helpful features: Aggregated View and Rollup.

Aggregated Views

The aggregated view is designed to give users a high-level summary of how a workflow has performed across multiple runs.

See the diagram below:



Instead of manually going through every instance of a workflow execution, users can see a combined overview of statuses across all runs. This includes information such as how many instances:

- Succeeded

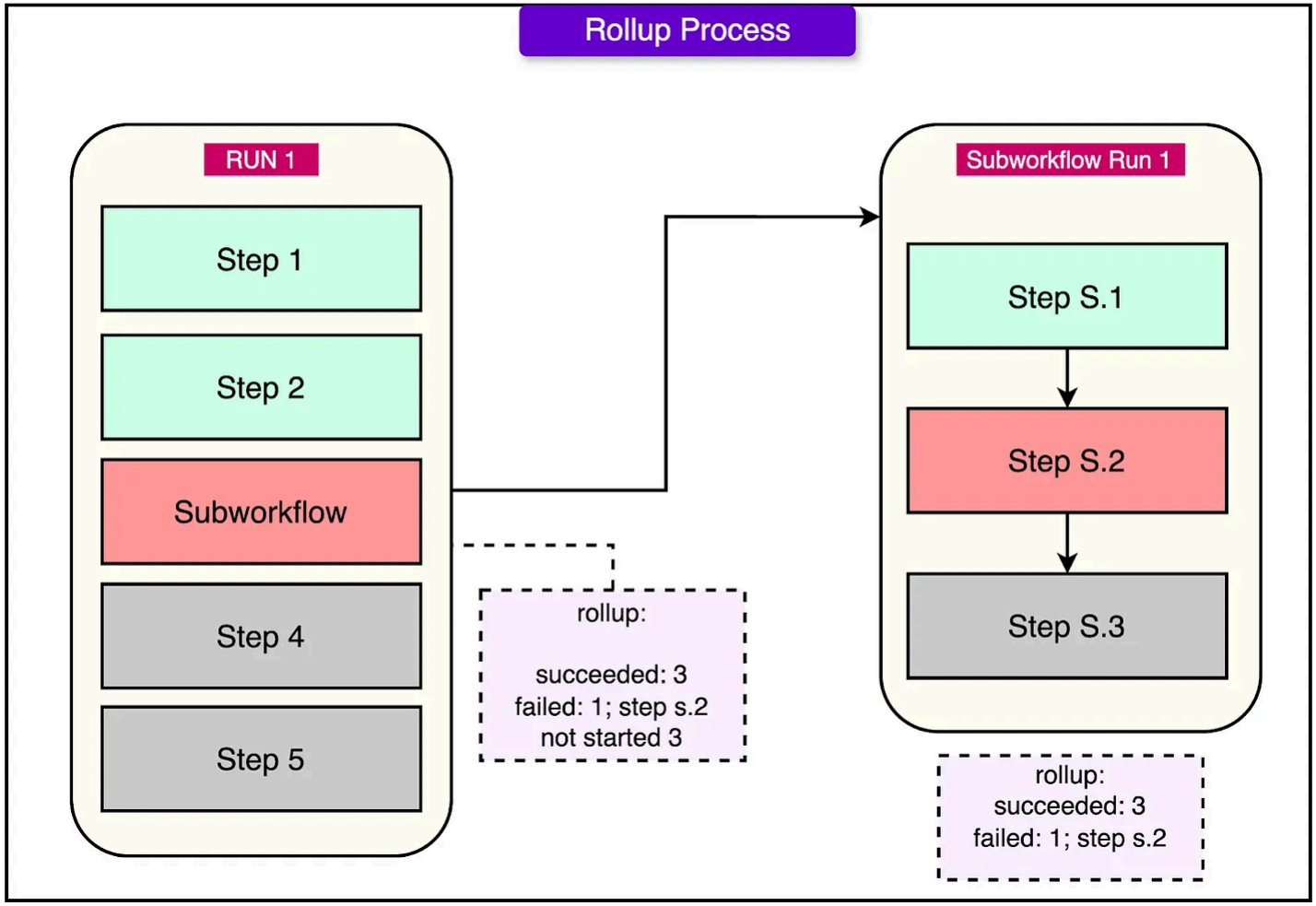
- Failed
- Were skipped
- Are still running

This helps users quickly understand the overall behavior and health of a workflow over time, making it easier to spot patterns or recurring issues.

Rollup

The rollup feature goes a step further by providing a flattened summary of all the steps in a workflow, even when the workflow contains subworkflows, foreach iterations, or nested loops.

In large workflows, a single step can itself be a foreach loop or a call to another workflow, which then includes its steps. Rollup pulls all of this structure together into a unified view.



With rollup, users can:

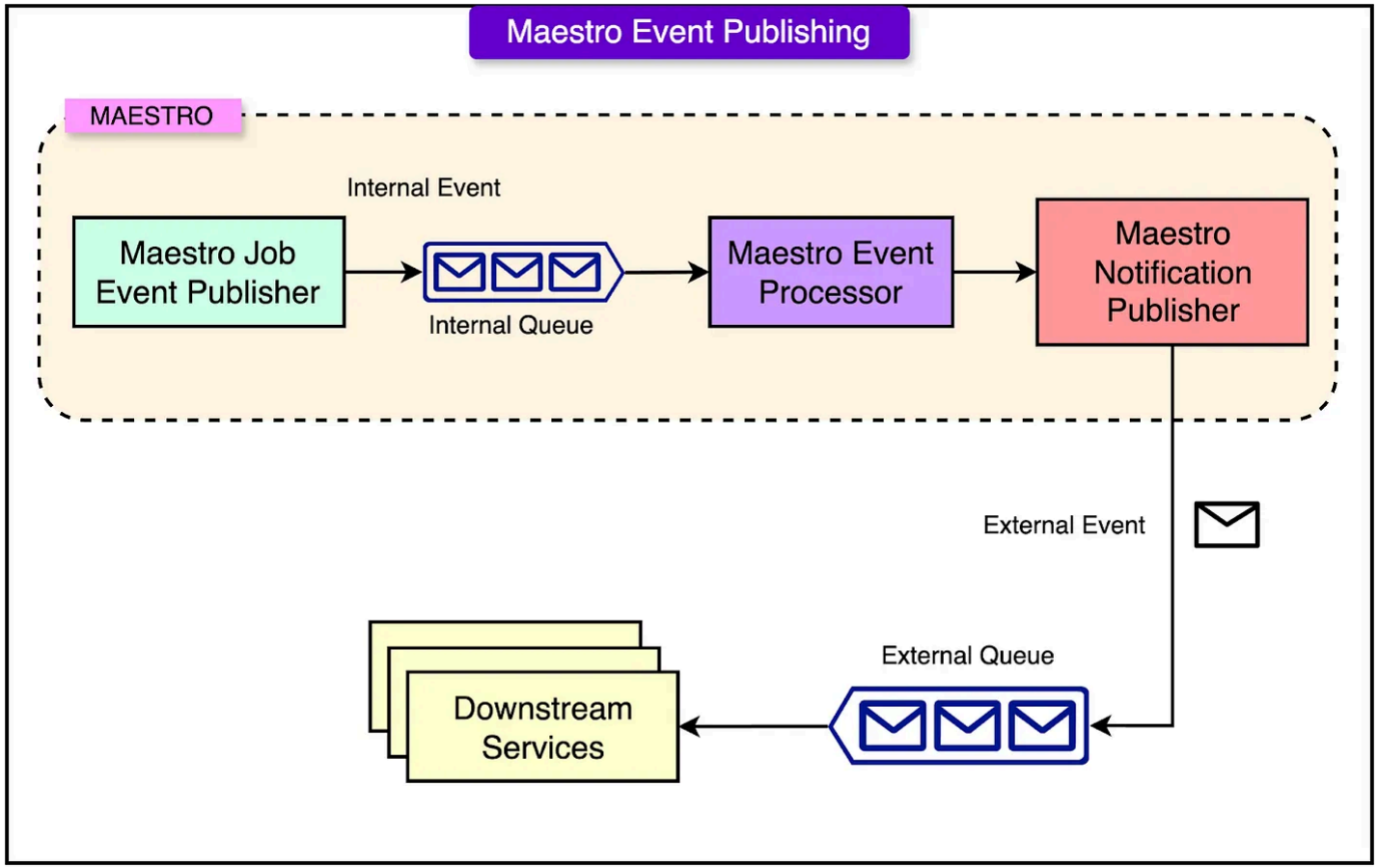
- See the total count of steps across all levels of the workflow.
- Track how many of those steps succeeded, failed, were skipped, or were retried.

- Get recursive visibility, meaning it looks deep into all execution layers to give a complete picture.

Event Publishing

Maestro is designed not just to run workflows but also to integrate smoothly with other systems at Netflix. One of the key ways it achieves this is through a robust event publishing system, which keeps various services informed about what is happening inside the orchestration platform in real time.

See the diagram below:



There are two main types of events that Maestro generates.

1 - Internal Events

These are events that describe the internal lifecycle of a workflow. They include things like:

- When a workflow starts
- When a step begins execution
- When a step or workflow completes, fails, or is retried
- When a workflow is paused, resumed, or cancelled

These events are used within Maestro itself and by other internal Netflix systems to track execution status, handle retries, and support user notifications or UI updates. They help ensure that the orchestration engine behaves correctly and reliably, especially under high workloads.

2 - External Events

Maestro can also publish events to external messaging systems such as Kafka or SNS (Simple Notification Service).

These events are typically consumed by other systems that need to react to changes in workflow state. For example:

- A data warehouse system might start preparing data once it knows a data pipeline workflow has completed.
- A monitoring system might trigger alerts if a critical workflow fails.

- A downstream job might begin execution only after receiving confirmation that the upstream job has succeeded.

Some examples of events published externally include:

- Workflow definition changes, such as when a workflow is updated, versioned, or deleted.
- Workflow instance state transitions, like when an instance moves from “scheduled” to “running” or “failed” to “succeeded.”

By publishing these events, Maestro allows other services in the Netflix ecosystem to stay in sync with what's happening in the orchestration platform.

Conclusion

Maestro is Netflix’s answer to the growing complexity and scale of modern data and machine learning workflows.

Designed as a horizontally scalable, cloud-native orchestrator, it addresses the limitations of its predecessor, Meson, by providing a robust architecture built on stateless microservices, distributed queues, and a strong consistency layer via CockroachDB.

Its support for high-level execution abstractions, dynamic parameterization, and deep modularity through features like foreach, subworkflows, and reusable templates allows users to build scalable, maintainable pipelines.

With multiple DSLs, a visual UI, and integrations like Metaflow, Maestro is accessible to a broad range of users, from engineers to analysts. Its advanced monitoring features, such as signal lineage, rollups, and event publishing, ensure transparency and real-time observability across workflows.

As Netflix continues to grow its data infrastructure, Maestro lays a strong, flexible foundation for the future of intelligent workflow orchestration. It has also been made open-source.

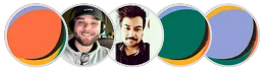
References:

- Orchestrating Data/ML Workflows at Scale with Netflix Maestro
- Maestro: Data/ML Workflow Orchestrator at Netflix
- Maestro: Netflix's Workflow Orchestrator

Our newsletter puts your products and services directly in front of an audience that matters - hundreds of thousands of engineering leaders and senior engineers - who have influence over significant tech decisions and big purchases.

Space Fills Up Fast - Reserve Today

Ad spots typically sell out about 4 weeks in advance. To ensure your ad reaches this influential audience, reserve your space now by emailing sponsorship@bytebytego.com.



81 Likes · 9 Restacks

Discussion about this post

Comments Restacks

Write a comment...