

A brief summary of Netflix article

“Orchestrating Data/ML Workflows at Scale With Netflix Maestro”

At Netflix, Data and Machine Learning (ML) pipelines are widely used and have become central for the business, representing diverse use cases that go beyond recommendations, predictions and data transformations. A large number of batch workflows run daily to serve various business needs. These include ETL pipelines, ML model training workflows, batch jobs, etc. As Big data and ML became more prevalent and impactful, the scalability, reliability, and usability of the orchestrating ecosystem have increasingly become more important for our data scientists and the company.

Motivation:

- Our existing orchestrator (Meson) has worked well for several years. It schedules around 70 thousands of workflows and half a million jobs per day.
- facing issues such as :
 - Slowness during peak traffic moments like 12 AM UTC, leading to increased operational burden
 - Meson was based on a single leader architecture with high availability. As the usage increased, we had to vertically scale the system to keep up and were approaching AWS instance type limits.

Challenges in workflow orchestration:

- Scalability
 - schedule hundreds of thousands of workflows, millions of jobs every day
 - operate with a strict SLO of less than 1 minute of scheduler introduced delay
 - lot of our workflows are run around midnight UTC -> burst in traffic
 - in terms of size of workflow
 - ML model training workflows usually consist of tens of thousands (or even millions) of training jobs within a single workflow.
 - this create hotspots and overwhelm the orchestrator and downstream systems
- Usability
 - to focus on their business logic and let the orchestrator solve cross-cutting concerns like scheduling, processing, error handling, security etc.
 - It should also provide all the knobs for configuring their workflows to suit their needs.
 - be debuggable and surface all the errors for users to troubleshoot, as they improve the UX and reduce the operational burden.
 - Providing abstractions for the users is also needed to save valuable time on creating workflows and jobs

- rely on shared templates and reuse their workflow definitions across their team, saving time and effort on creating the same functionality

Introducing Maestro

- Maestro is the next generation Data Workflow Orchestration platform to meet the current and future needs of Netflix.
- It is a general-purpose workflow orchestrator that provides a fully managed workflow-as-a-service (WAAS) to the data platform at Netflix
- highly scalable and extensible to support existing and new use cases and offers enhanced usability to end users

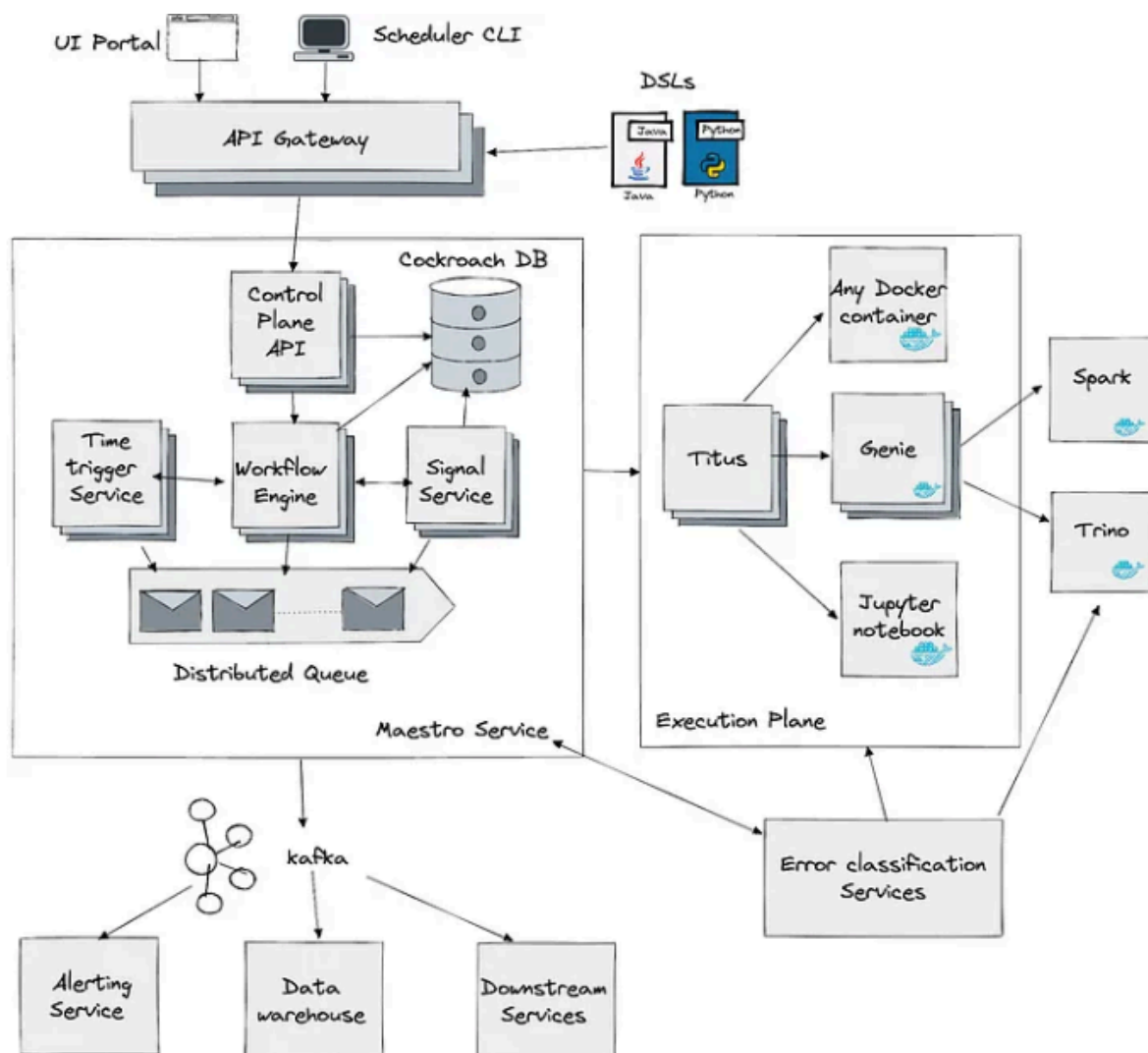


Figure 1. Maestro high level architecture

In Maestro, a workflow is a [DAG \(Directed acyclic graph\)](#) of individual units of job definition called Steps.

- Steps/jobs can have dependencies, triggers, workflow parameters, metadata, step parameters, configurations, and branches (conditional or unconditional)

- A workflow instance is an execution of a workflow
- an execution of a step is called a step instance

The system consists of 3 main micro services which we will expand upon in the following sections.

- Maestro launches a unit of work (a.k.a. Steps) in a container and ensures the container is launched with the users/applications identity.
- Launching with identity ensures the work is launched on-behalf-of the user/application, the identity is later used by the downstream systems to validate if an operation is allowed or not, for an example user/application identity is checked by the data warehouse to validate if a table read/write is allowed or not.

1.Workflow Engine

- Workflow engine is the core component, which manages workflow definitions, the lifecycle of workflow instances, and step instances
- Provides support to
 - Any valid DAG patterns
 - data flow constructs such as subflow, foreach loop, conditional branching etc.
 - multiple failure mode to handle/retry failure with different failure policy
 - Flexible concurrency control to throttle the number of executions at workflow/step level
 - Job templates for common job patterns like running a Spark query or writing to google sheets etc.
 - workflow definition, timeline, management

2.Time-Based Scheduling Service

- Time-based scheduling service starts new workflow instances at the scheduled time specified in workflow definitions
- This service is lightweight and provides an at-least-once scheduling guarantee.
- Maestro engine service will deduplicate the triggering requests to achieve an exact-once guarantee

But sometimes, it is not efficient. For example, the daily workflow should process the data when the data partition is ready, not always at midnight.

Therefore,

3.Signal Service

- Maestro supports event-driven triggering over signals
- Signal triggering is efficient and accurate because we don't waste resources checking if the workflow is ready to run, instead we only execute the workflow when a condition is met.

- Signals are used in two ways:
 - A trigger to start new workflow instances
 - A gating function to conditionally start a job
- Signal service goals are to
 - Collect and index signals
 - Register and handle workflow trigger subscriptions
 - Register and handle the step gating functions
 - Captures the lineage of workflows triggers and steps unblocked by a signal

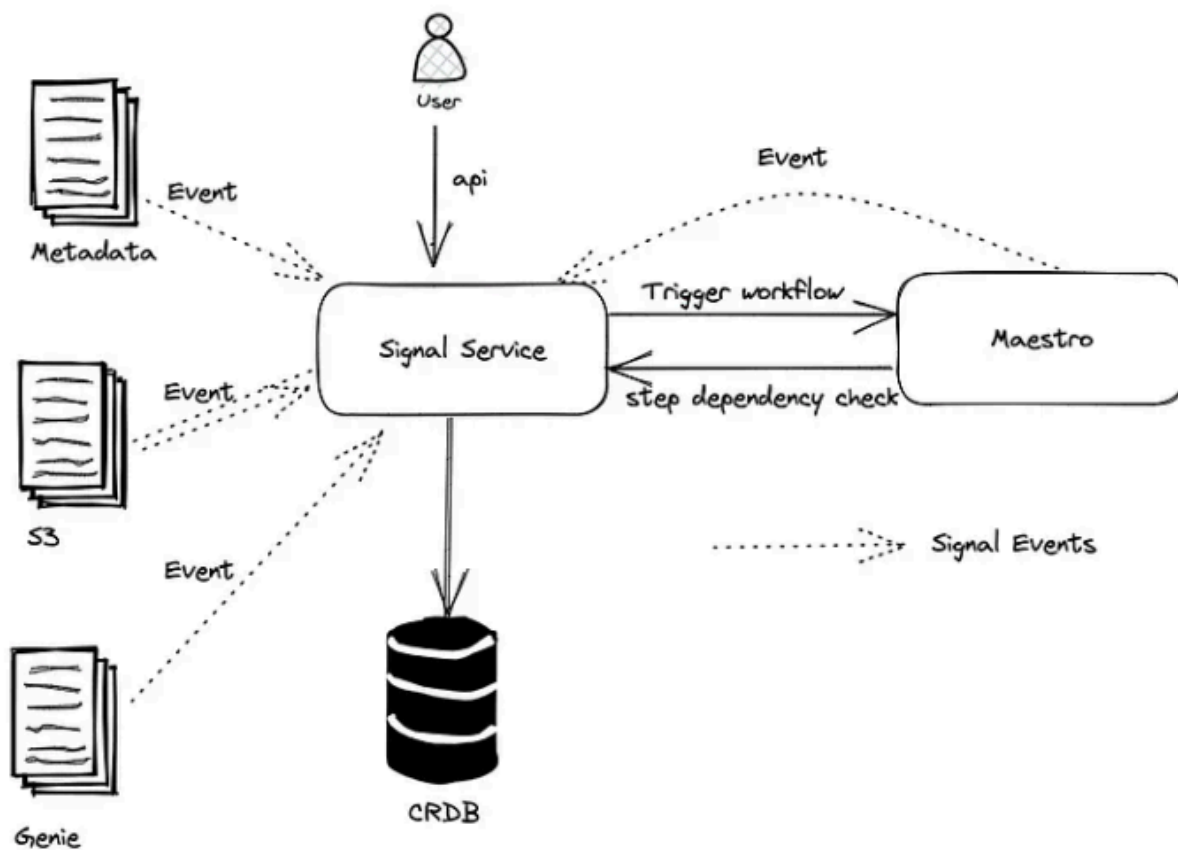


Figure 2. Signal service high level architecture

- Maestro signal service
 - consumes all the signals from different sources
 - generates the corresponding triggers by correlating a signal with its subscribed workflows
 - provides the signal lineage
 - a table updated by a workflow could lead to a chain of downstream workflow executions.
 - Most of the time the workflows are owned by different teams, the signal lineage helps the upstream and downstream workflow owners to see who depends on whom.

Orchestration at Scale

- All services in the Maestro system are stateless and can be horizontally scaled out.
- All the requests are processed via distributed queues for message passing
- CockroachDB is used for persisting workflow definitions and instance state
 - provides strong consistency guarantees that can be scaled horizontally without much operational overhead.

Concept of sub-workflow and use of foreach

- Its not possible to render a workflow on UI which consists of thousands or millions of Jobs.
- There is the need to enforce some constraints and support valid use cases
- **Therefore, we enforce a workflow DAG size limit (e.g. 1K) and we provide a foreach pattern that allows users to define a sub DAG within a foreach block and iterate the sub DAG with a larger limit (e.g. 100K).**

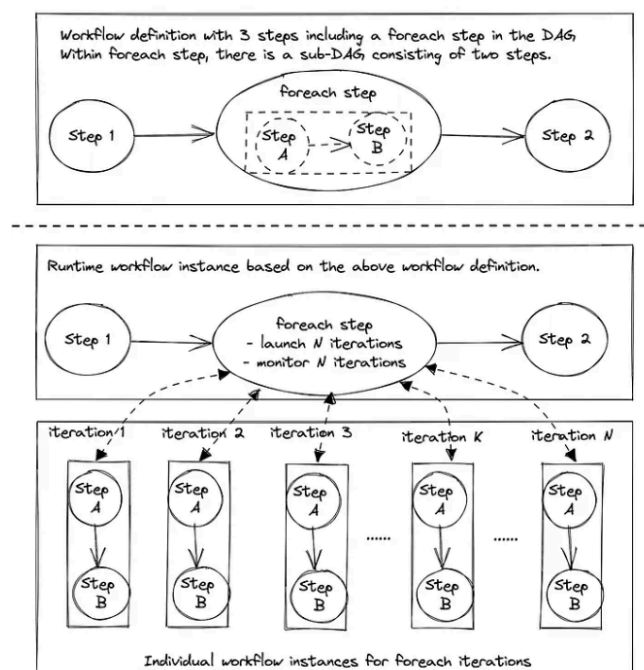


Figure 3. Maestro's scalable foreach design to support super large iterations

With this design, foreach pattern supports sequential loop and nested loop with high scalability.

Workflow Platform for Everyone

- The users can bring their business logic in multiple ways, including but not limited to,
 - a bash script, a Jupyter notebook, a Java jar, a docker image, a SQL statement, or a few clicks in the UI using workflow templates.
- Maestro provides multiple domain specific languages (DSLs) including YAML, Python, and Java, for end users to define their workflows, which are decoupled from their business logic

Here is an example workflow defined by different DSLs.

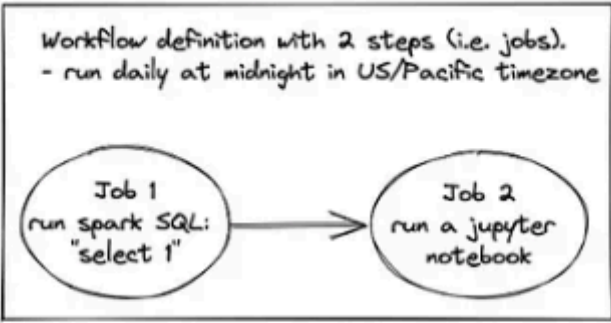
Workflow Definition	YAML DSL
<p>Workflow definition with 2 steps (i.e. jobs).</p> <ul style="list-style-type: none"> - run daily at midnight in US/Pacific timezone 	<pre> Trigger: cron: '@daily' tz: US/Pacific Workflow: id: demo.pipeline jobs: - job: id: job1 type: Spark spark: script: SELECT 1; - job: id: job2 type: Notebook notebook: input_path: s3://path/to/notebook.ipynb </pre>
Python DSL	Java DSL
<pre> 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')) </pre>	<pre> 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(); </pre>

Figure 4. An example workflow defined by YAML, Python, and Java DSLs

Additionally, users can also generate certain types of workflows on UI or use other libraries, e.g.

- In Notebook UI, users can directly schedule to run the chosen notebook periodically.
- In Maestro UI, users can directly schedule to move data from one source (e.g. a data table or a spreadsheet) to another periodically.
- Users can use [Metaflow](#) library to create workflows in Maestro to execute DAGs consisting of arbitrary Python code.

Parameterized Workflows

- Lots of times, users want to define a dynamic workflow to adapt to different scenarios but complete dynamism is hard to maintain and troubleshooting
- Instead, Maestro provides three features to assist users to define a parameterized workflow
 - Conditional branching
 - Sub-workflow
 - Output parameters

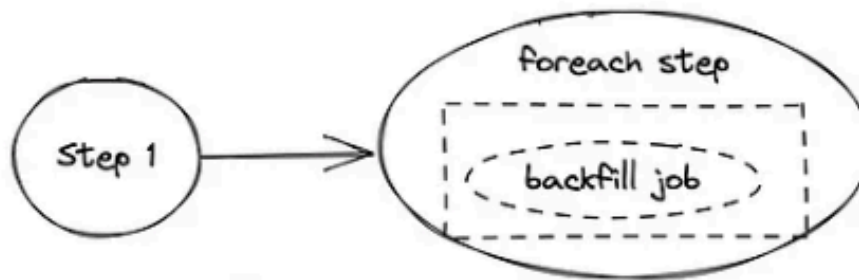
- so, users can define those changes as sub workflows and then invoke the appropriate sub workflow at runtime

An example (using YAML DSL) of backfill workflow with 2 steps.

- In step1, the step computes the backfill ranges and returns the dates (from 20210101 to 20220101) back.
- Next, foreach step uses the dates from step1 to create foreach iterations.
- Finally, each of the backfill jobs gets the date from the foreach and backfills the data based on the date.

```
Workflow:
  id: demo.pipeline
  FROM_DATE: 20210101 #inclusive
  TO_DATE: 20220101   #exclusive
  jobs:
    - job:
        id: step1
        type: NoOp
        '!dates': dateIntsBetween(FROM_DATE, TO_DATE, 1); #SEL expr
    - foreach:
        id: step2
        params:
          date: ${dates@step1} #reference upstream step parameter
        jobs:
          - job:
              id: backfill
              type: Notebook
              notebook:
                input_path: s3://path/to/notebook.ipynb
                arg1: $date #pass the foreach parameter into notebook
```

Workflow definition with 2 steps including a foreach step in the DAG. Within foreach step, there is a sub-DAG consisting of 1 backfill job.



Runtime workflow instance based on the above workflow definition.

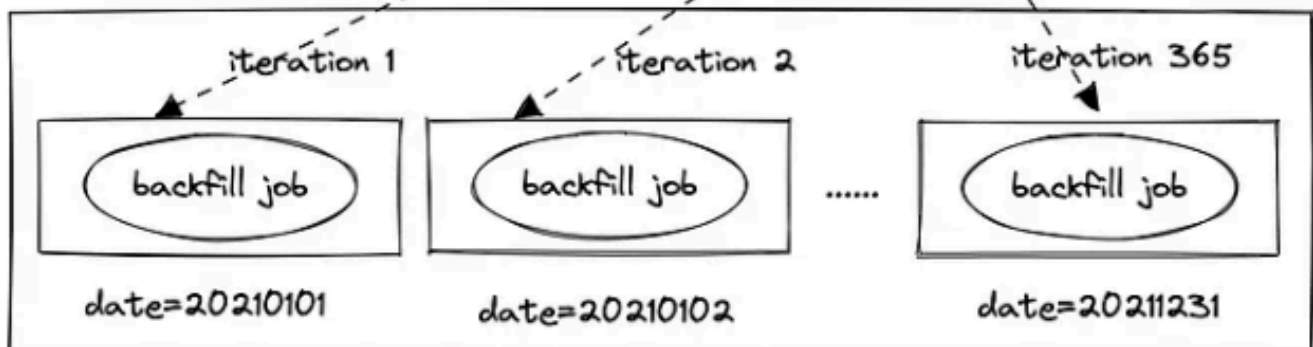
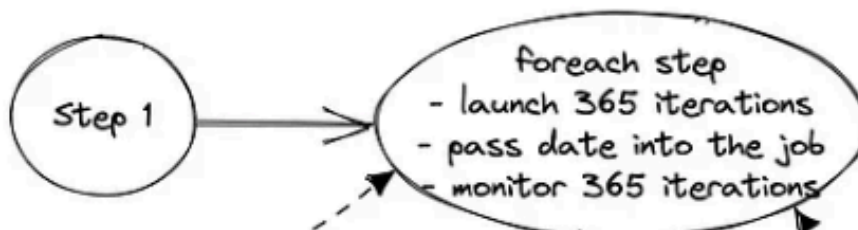


Figure 5. An example of using parameterized workflow to backfill data