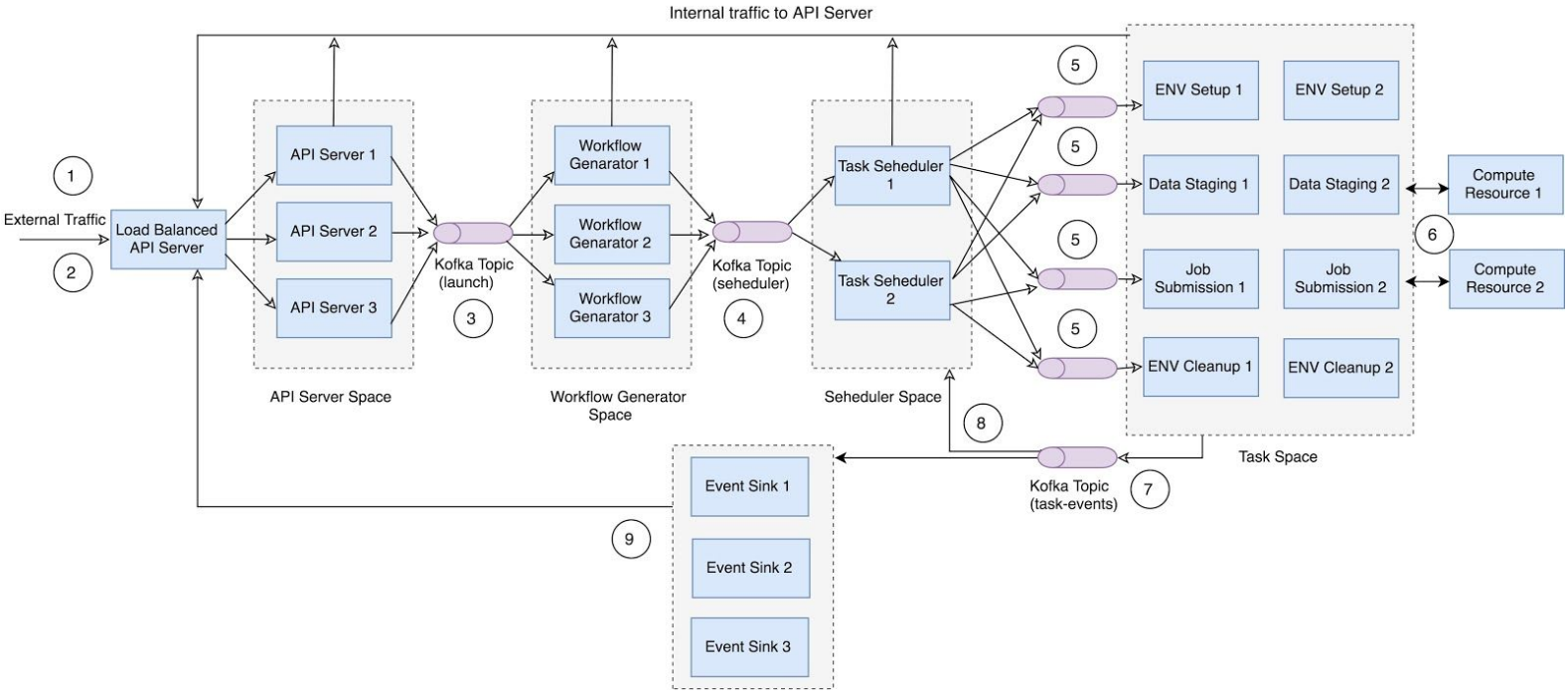
Container Based Task Execution Workflow for Airavata

Design Document

Dimuthu Upeksha

Architecture



Component Overview

API Server

API Server is the entrypoint to accept and launch Experiments in the system. Functionality wise this is analogous to current API server of Airavata. API Server is used both in external invocations such as Experiment creations by users and internal invocations such as catering Experiment and Task details to internal microservices. API Server is a stateless microservice and critical to the operation of other the components of the platform. So it is essential to keep multiple copies of API Server with a load balanced endpoint

Workflow Generator

Once an Experiment launch is triggered by the user, Workflow Generator parses the Experiment and creates a Process and a DAG of Tasks assigned to that particular Process. These Tasks can be identified as distinct steps that should be followed in order to execute the Experiment on

a compute host. Examples for Task types are Environment Setup Tasks, Job Scheduling Tasks and Data Staging tasks.

Task Scheduler

Once the Workflow Generator generated a Task DAG for a particular Process, Task Scheduler schedules each Task of the Process according the DAG order and the events received from the previously executed Tasks.

Tasks

A Task is a microservice that has a clearly defined set of commands to run on a Compute Host. There could be different kind of Tasks specialized in different functionalities. Data Staging Tasks are capable of fetching data from a cloud store and place in a given path of the compute host. Job scheduling Tasks are capable of running the executables of the application on the compute host.

Event Sink

Event Sink is responsible for capturing events that are emitted by each task and persist them in the database by invoking the API Server.

Message Flow

1. User creates Application Modules, Compute Resources, Application Deployments, Application Interfaces and Experiments by invoking the REST API of API-Server.

2. Using the same API, user triggers a launch of an Experiments

3. API Server accepts the launch request and stores states in the database. Then forward the launch request to launch topic to be picked by a Workflow Generator

4. A Workflow Generator picks the launch request from launch topic and starts to generate a Task DAG for the Experiment. This DAG is wrapped by a Process object and forwards the Process object to scheduler topic to be picked by a Task Scheduler

5. Task Scheduler picks the Process object and fetches the Task DAG of it. Once the DAG is loaded, Scheduler starts to schedule one by one. According to the Task type,

Scheduler forwards the execution request to a suitable topic to be picked by the correct Task Executor.

6. A Task Executor picks the task execution request and executes the task on target compute host.

7. State of the Task is published to the task-events topic which will be eventually consumed by the Scheduler.

8. Scheduler reads the status of the Task and if it is a Success state, next Task of the DAG is executed. Same process is followed until the DAG is completed and if all Tasks are successfully executed, Scheduler updates Process of the DAG as Completed

9. Event sinks are running in the background to monitor task-events topic and to persist them in the database for future debugging purposes.

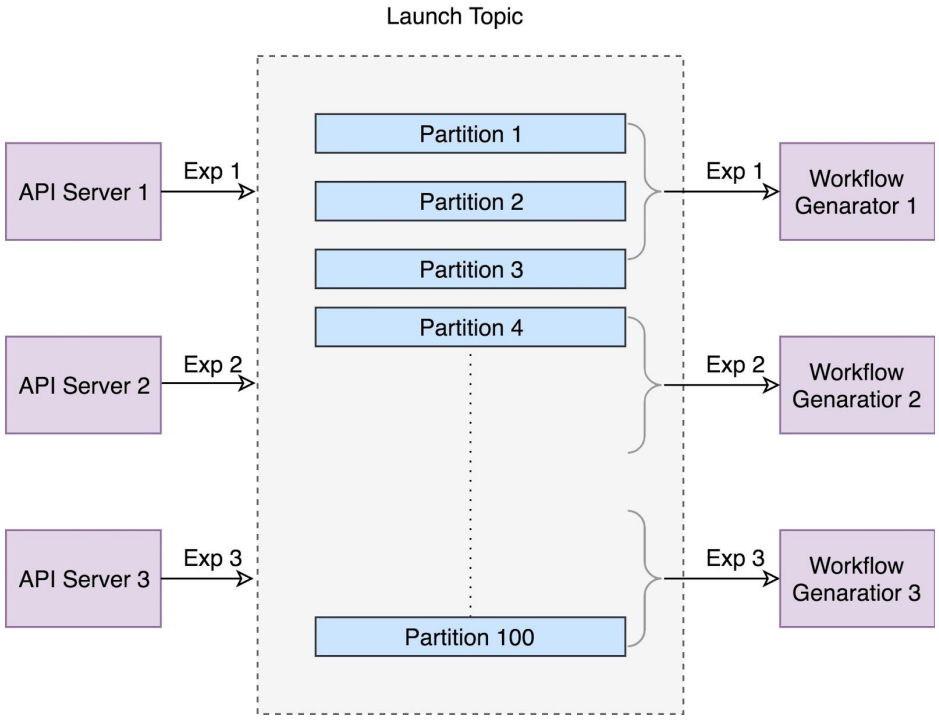
Communication and Coordination Between Components

Two communication mediums are mainly used.

1. HTTP request response model to query API server by external entities and internal microservices
2. Communication between other components is performed through a Kafka message broker.

Passing Experiment launch command from API server to Workflow Generator

Messaging medium : Kafka
Topic name : airavata-launch
Partitions : 100



When a user triggers an Experiment launch through API Server, API Server drops the Experiment id to “launch” Kafka topic. There are one or many Workflow Generators watching at the “launch” topic for an experiment id. As all Workflow Generators are in a single kafka consumer group, it is guaranteed that a particular experiment id is read by only one Workflow Generator. This ensures that the same experiment is not launched twice. So we do not want explicit cluster wide coordination (cluster locks or leader election) to access “airavata-launch” topic from Workflow Generators.

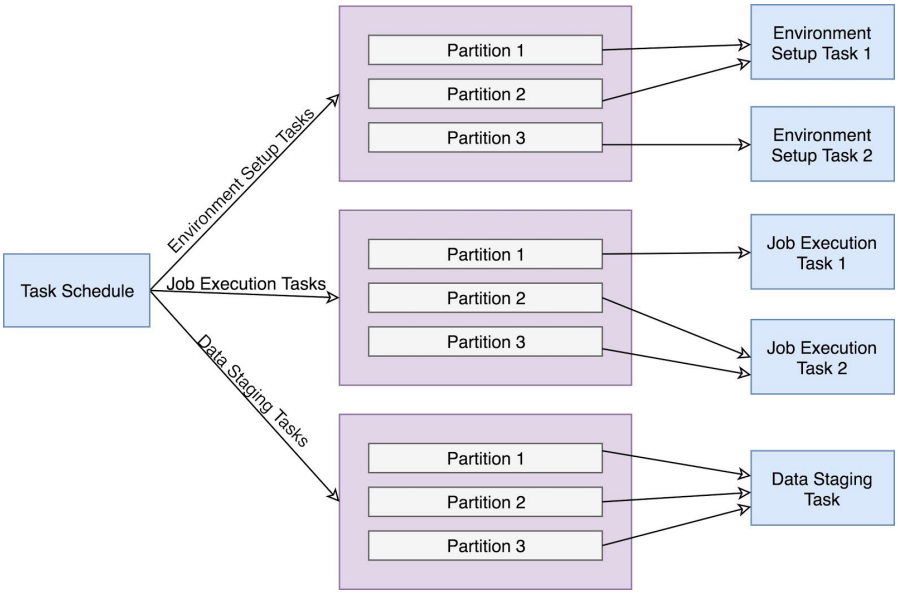
Passing Process execution command from Workflow Generator to Task Scheduler

Messaging medium : Kafka
Topic name : airavata-scheduler
Partitions : 100

Workflow Generator creates a Process for the launching Experiment and attach a Task DAG to the Process. Then the Process id is dropped to “scheduler” topic. Task Scheduler pool is watching for the Process ids to be appeared from the topic. Because the topic is partitioned and Task Schedulers are grouped into a single kafka consumer group, it is guaranteed that a particular process id is consumed by exactly one Task Scheduler. So we do not want explicit cluster wide coordination (cluster locks or leader election) to access “airavata-scheduler” topic from Workflow Generators.

Passing tasks to execute from Task Scheduler to Task Executors

Messaging medium : Kafka
Topic name : specific to task type
Partitions : 100



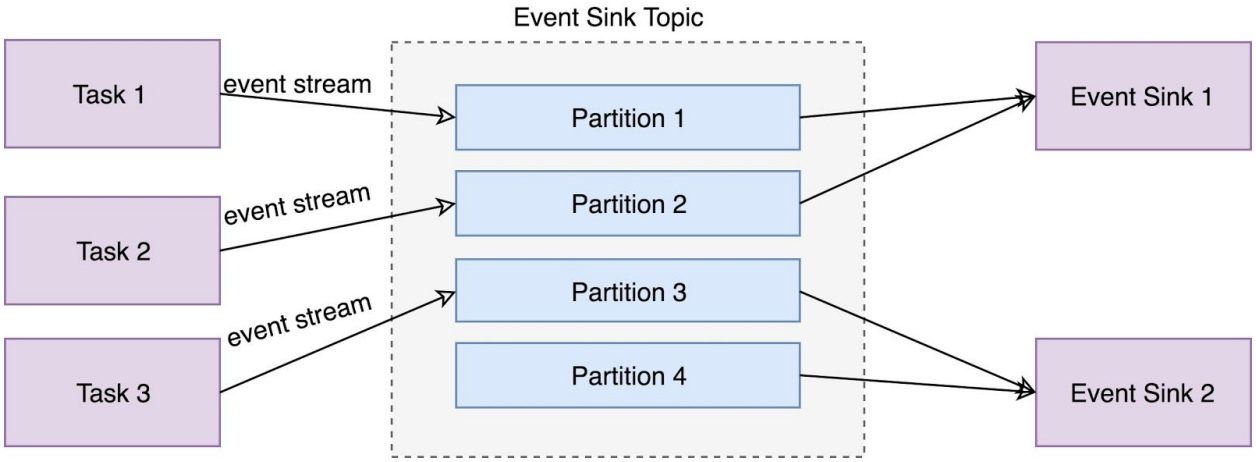
Task scheduler fetches the Task DAG for the fetched Process and select next running task from the DAG. Depending the type of the Task, Scheduler determines the topic that it should be

published. There are separate topics for each Task type. For example, if the Task is a data staging type, Scheduler drops the task id to “ingress-staging-task” topic. For each task type, there are one or many Task Executor microservices watching at the particular topic. Task Executors of a particular type are grouped by a single kafka consumer group which makes sure a task is executed by exactly one Task Executor.

Task event publishing

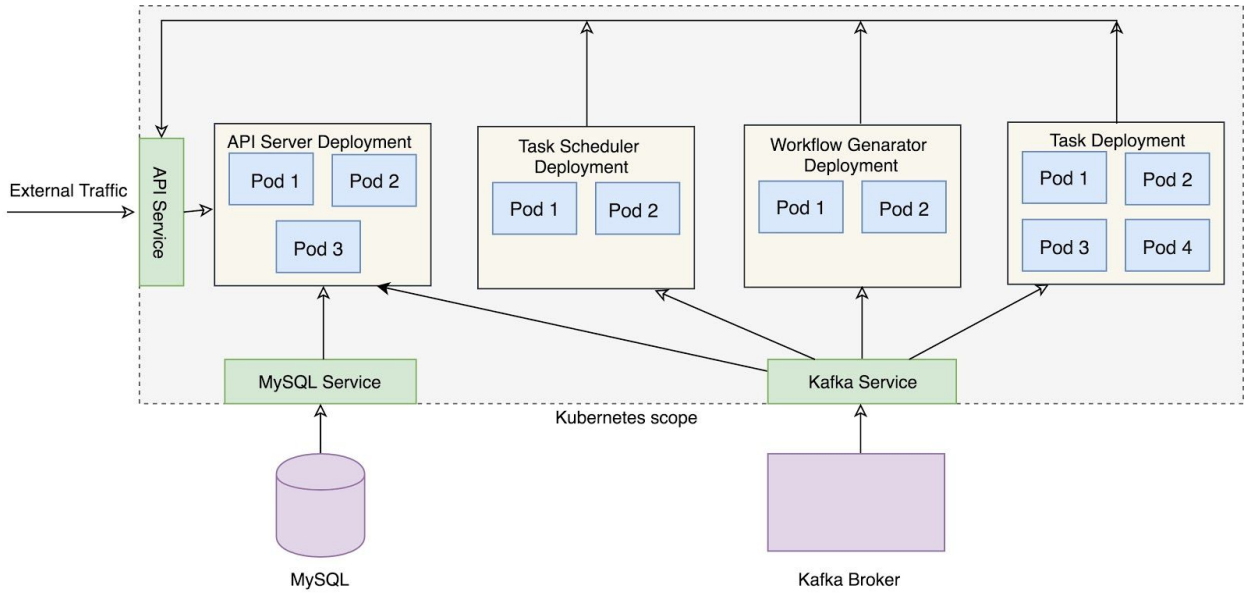
Messaging medium : Kafka
Topic name : event-sink
Partitions : 100

For each task there could be several runtime states to be published. Depending on these events, Task Scheduler makes decisions to execute next tasks in the DAG. So the correct order and the guaranteed delivery of events are very critical. So we should make sure that the events of a particular Task Executor are routed to a single partition of the “event-sink” topic. To achieve this, Task Executors are publishing messages to the topic with a same message key (<process-id>-<task-id>).



Deployment

Each microservice is bundled as a Docker image and pushed to an docker registry (in my case, to DockerHub). Microservices are deployed inside a Kubernetes cluster as Kubernetes Deployments, to enable rolling updates and dynamic scaling. API Server is exposed to outside and internal services using a Kubernetes Service which enables load balancing and service discovery out of the box. MySQL server and Kafka Broker are deployed outside the Kubernetes cluster.



Fault Tolerance

Each microservice that is deployed inside the platform can be horizontally scaled according to the demand while keeping the functionality of the platform consistent. For better performance and high availability, we need to have multiple instances of each Microservice deployed.

Future Work

1. Have to improve message flow transactionality. Current implementation is not 100% transactional in the process of message fetching from a Kafka topic and executing them. If the message can not be processed by a consumer, message should be put back to topic in order to be consumed by some other party.

2. Task Scheduler terminations should be handled gracefully. If a Task Scheduler got killed during a DAG execution process, new Task Scheduler should be able to continue from the last executing point. As Task Sinks are periodically persisting the Task events in the Database, this is technically possible