Name: Sebastian James Sampao Section: 3DSA

*Formative Lab Exercise #5:*

**Workflow Orchestration with Kestra**

# **Kestra Overview**

**A diagram of a computer

AI-generated content may be incorrect.**

Sources:

<https://kestra.io/docs/architecture>

1. Kestra Key Features

Kestra is an open-source orchestration platform that:

* Lets you define workflows declaratively in YAML
* Allows non-developers to automate tasks with a no-code interface
* Keeps everything versioned and governed, so it stays secure and auditable
* Extends easily for custom use cases through plugins and custom scripts.

Kestra allows you to combine the flexibility of code-based orchestration with a no-code interface that anyone can learn in minutes.

Architecture

1. JDBC Backend

* This is similar to the Apache Airflow Metadata Database, this contains the metadata for the orchestration based on either mySQL or PostgreSQL

1. Server

* The central part of the system contains:
  1. Scheduler
  + The component that is responsible for scheduling jobs, sending tasks to the executor, and managing the flow of events in the workflow.
  1. Executor
  + Responsible for the orchestration logic including flow triggers.
  1. Worker
  + These are the processes that carry out the computation of runnable tasks or jobs
  1. Webserver for the
  + Serves both the API and the User Interface

Sources:

<https://kestra.io/docs/why-kestra>

<https://kestra.io/docs/architecture>

1. Differences with Airflow

* YAML based instead of the code based
* The UI allows you to edit directly the workflow
* Less verbose than Airflow
* Does not require Python knowledge to create complex pipelines
* Execution is event driven unlike Airflow that rely on third-party executioners like Celery

# **Lab Part 0: Getting Started with Kestra**

### **Task 0.1: Setting up Kestra using Docker Compose**

*Bash*

curl -o docker-compose.yml \

https://raw.githubusercontent.com/kestra-io/kestra/develop/docker-compose.yml

### **Task 0.2: Accessing the Kestra UI**

*Bash*

docker-compose build

docker-compose up

*Web Browser*

Access: localhost:8080  
Username: [admin@kestra.io](mailto:admin@kestra.io)  
Password: Admin1234

📸 Screenshot Submission 0.3:

Show that Kestra is running on Docker and the UI is accessible

A black screen with colorful text

AI-generated content may be incorrect.A screenshot of a computer

AI-generated content may be incorrect.

🧠 Checkpoint Question Submission 0.4:

Checkpoint: What are the main services that Kestra launches when run with Docker Compose?

🧠 Checkpoint Question Submission 0.5:

Checkpoint: How does the Kestra UI help you understand what’s happening in the background?

# **Lab Part 1: Writing Your First DAG on Kestra**

### **Task 1.1: Create a simple sequential DAG in YAML**

* hello-world.yaml – Appendix 2

📸 Screenshot Submission 1.2:

YAML file for your first DAG on Kestra

A screenshot of a computer

AI-generated content may be incorrect.

📸 Screenshot Submission 1.3:

Show your simple Kestra flow loaded and a successful run.

A screenshot of a computer

AI-generated content may be incorrect.

🧠 Checkpoint Question Submission 1.4:

In your YAML, what structure is used to define a sequence of tasks?

1. Tasks are defined under the tasks section and tasks are defined as a YAML list item starting with –, a unique ID, and a type that defines the plugin that executes the work.

🧠 Checkpoint Question Submission 1.5:

Compared to Airflow’s BashOperator, how does Kestra’s equivalent task behave?

* Airflow’s BashOperator and Kestra’s io.kestra.plugins.scripts.shell.Commands behave fundamentally the same: both execute shell commands as part of a workflow. The difference is that Kestra provides native alternatives like io.kestra.plugins.core.http.Download for curl, which are more robust and avoid issues with container mounts and permissions.

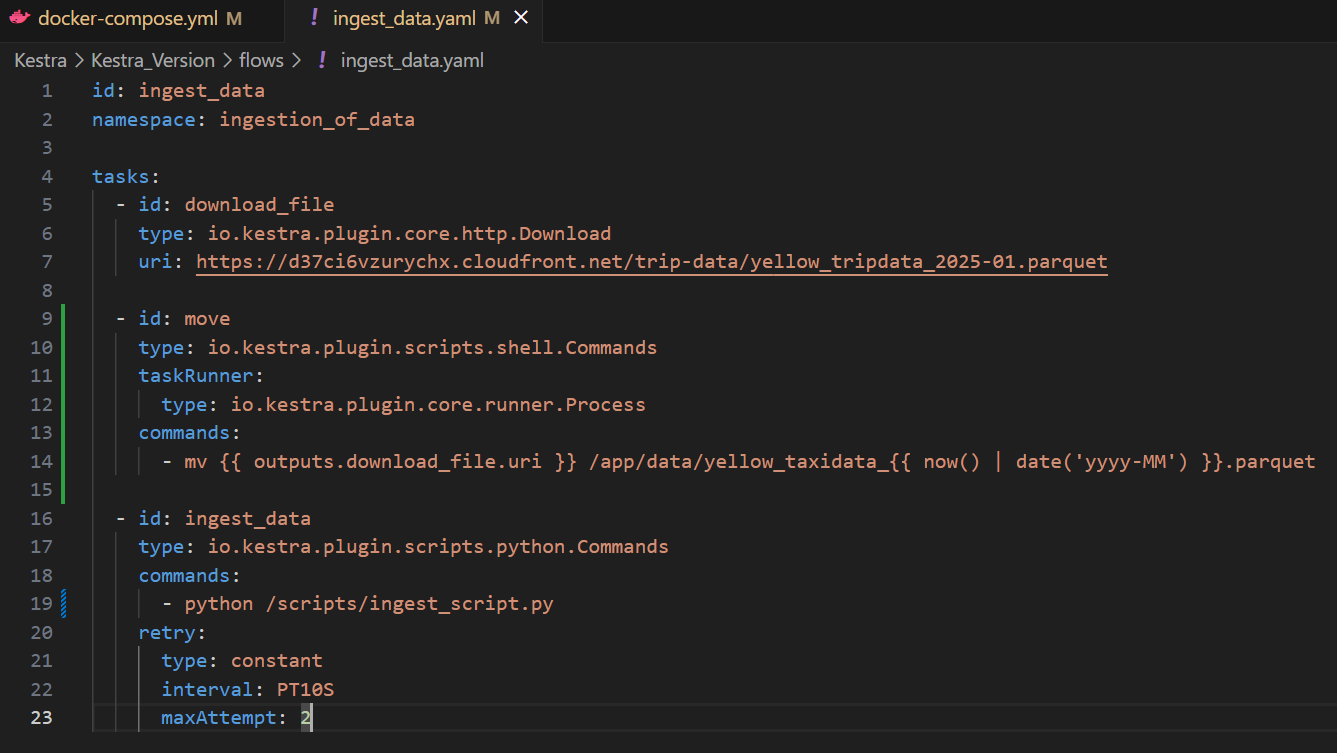
# **Lab Part 2: Writing Your First DAG on Kestra**

### **Task 2.1: Use a download task to fetch NYC Yellow Taxi data**

* ingest\_data.yaml – Appendix 3

📸 Screenshot Submission 2.2:

YAML file for your Ingestion DAG



📸 Screenshot Submission 2.2:

Show your ingestion DAG and log output of data download.



🧠 Checkpoint Question Submission 1.4:

Which Kestra task did you use to download the taxi dataset?

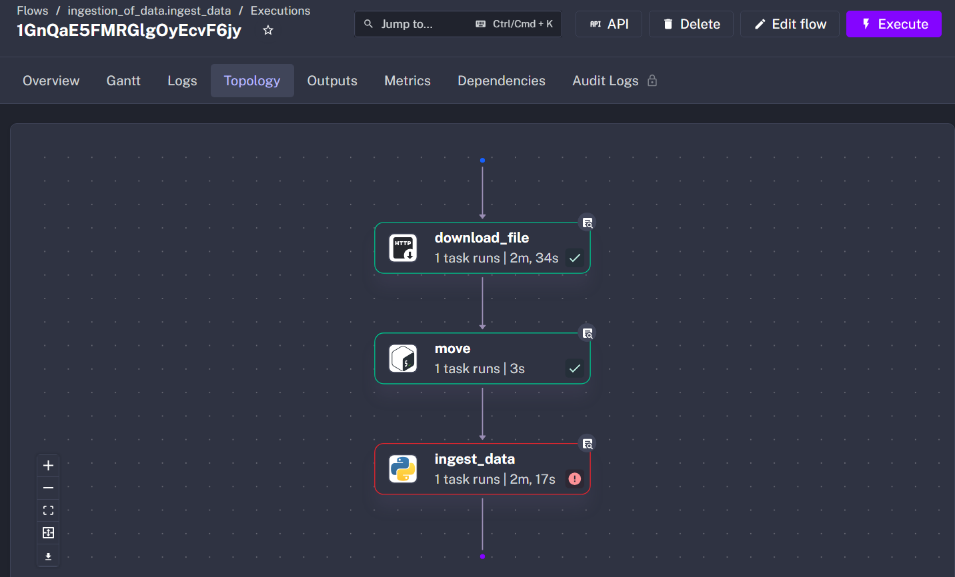
1. I used io.kestra.plugin.core.http.Download, a native kestra task that allows you to download files and store it in the temporary internal folder of kestra

🧠 Checkpoint Question Submission 1.4:

How did you make sure the file was saved to the correct location?

1. I used io.kestra.plugin.scripts.shell.Commands, a kestra plugin along with a io.kestra.plugin.core.runner.Process, a kestra task runner in order to move the downloaded file from the temporary folder into a bind-mounted folder.

# **Lab Part 3: Executing and Monitoring DAG on Kestra**

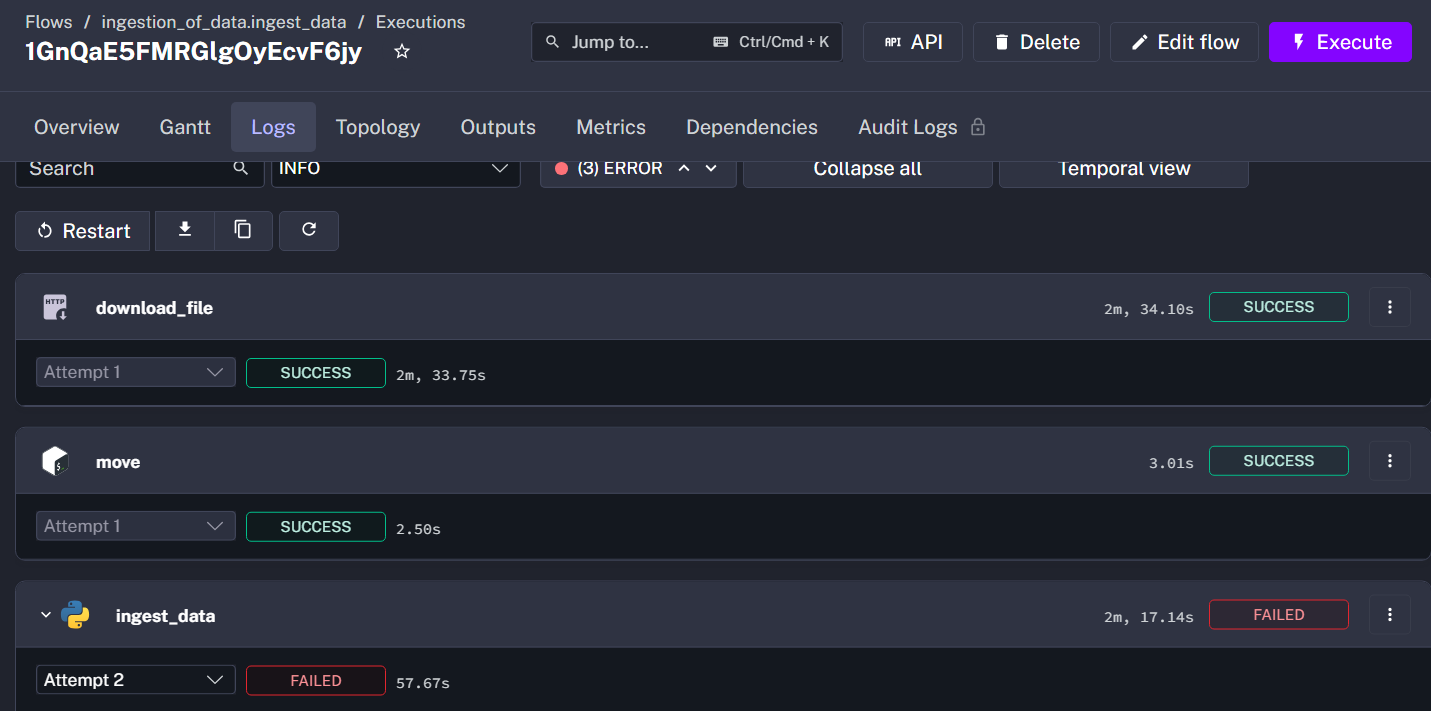


📸 Screenshot Submission 3.3:

Show the execution status and the run view (graph view).

📸 Screenshot Submission 3.4:

Show the execution status and the run view (timeline view).



🧠 Checkpoint Question Submission 3.5:

What’s the difference between a failed flow run and a failed task?

1. Failed flows are when the entire workflow execution breaks, this is mostly due to a task that has not fully recovered, or has a persistent failure in execution. On the other hand, failed tasks are step failure, it doesn’t necessarily immediately break a workflow execution if error handling methods and/or retries are put in place.

🧠 Checkpoint Question Submission 3.5:

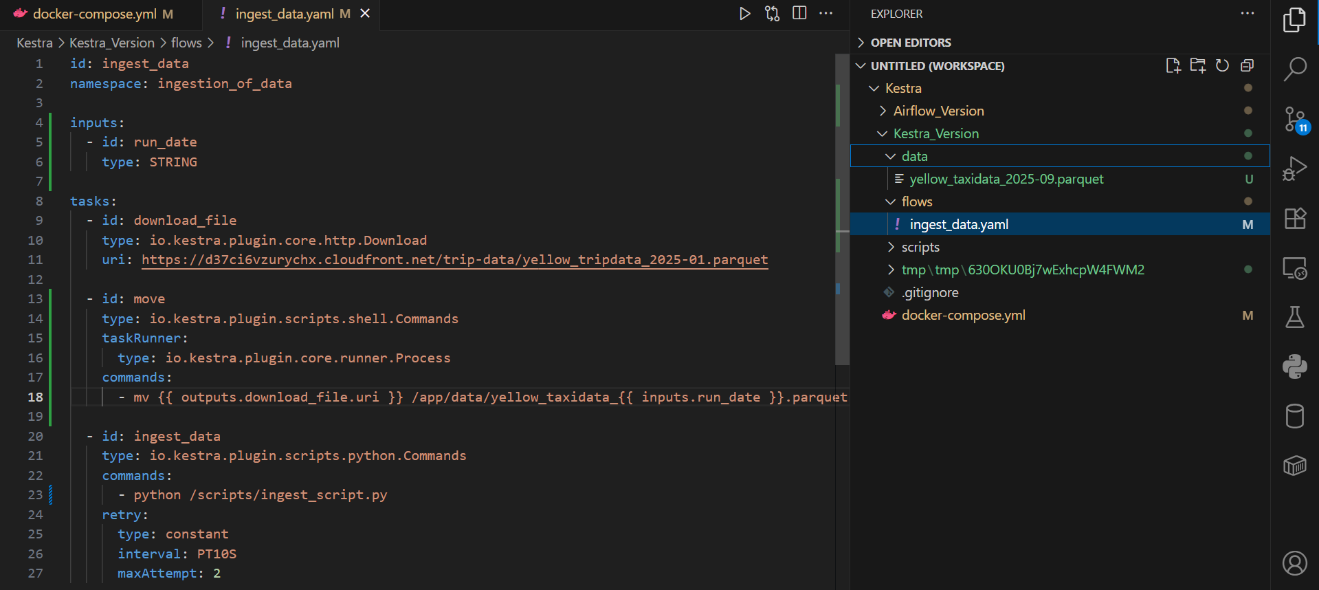
What features does Kestra provide to help you debug failed flows?

1. The Kestra UI itself provide a lot of useful information in not only diagnosing the point of failure but also fixing it. Kestra UI offers detailed logs and orchestration graphs, similar to airflow, but Kestra allows you to directly edit and change code on Kestra under the flows tab. Additionally, it allows to re-run, or retry failed executions with the same parameters/inputs.

# **Lab Part 4: Executing and Monitoring DAG on Kestra**

📸 Screenshot Submission 4.2:

Show your flow using a dynamic date-based filename and execution logs.



🧠 Checkpoint Question Submission 4.3:

How did you inject parameters like the date into your filename or commands?

1. First, the input section makes it so that Kestra asks the user for a input that is stored as a input with type datatype and identified with a unique ID. This input can be referenced as inputs.run\_date and similar to airflow, can be inserted using a Jinja based template.

🧠 Checkpoint Question Submission 4.4:

Why is it powerful to pass values into a DAG at runtime?

1. First, the input section makes it so that Kestra asks the user for a input that is stored as a input with type datatype and identified with a unique ID. This input can be referenced as inputs.run\_date and similar to airflow, can be inserted using a Jinja based template.

**Appendix 1: docker-compose.yaml**

***YAML***

volumes:

postgres-data:

driver: local

kestra-data:

driver: local

services:

postgres:

image: postgres:latest

volumes:

- postgres-data:/var/lib/postgresql/data

environment:

POSTGRES\_DB: kestra

POSTGRES\_USER: kestra

POSTGRES\_PASSWORD: k3str4

ports:

- "5432:5432"

healthcheck:

test: ["CMD-SHELL", "pg\_isready -d $${POSTGRES\_DB} -U $${POSTGRES\_USER}"]

interval: 30s

timeout: 10s

retries: 10

kestra:

image: kestra/kestra@sha256:ad3e4b61ef99bbcc31822bcb98d021124067acf3f5544eaaa3ebd6bfd013fddf

pull\_policy: always

user: "root"

command: server standalone

volumes:

- type: volume

source: kestra-data

target: /app/storage

- type: bind

source: ./data

target: /app/data

- type: bind

source: ./scripts

target: /app/scripts

- type: bind

source: ./tmp

- type: bind

source: /var/run/docker.sock

target: /var/run/docker.sock

environment:

KESTRA\_CONFIGURATION: |

datasources:

postgres:

url: jdbc:postgresql://postgres:5432/kestra

driverClassName: org.postgresql.Driver

username: kestra

password: k3str4

kestra:

server:

basicAuth:

username: admin@kestra.io

password: Admin1234

plugins:

configurations:

- type: io.kestra.plugin.scripts.runner.docker.Docker

values:

volume-enabled: true

repository:

type: postgres

storage:

type: local

local:

basePath: "/app/storage"

queue:

type: postgres

tasks:

tmpDir:

path: /tmp/kestra-wd/tmp

local-files:

allowed-paths:

- /app/data

- /app/scripts

- /tmp/kestra-wd

ports:

- "9090:8080"

- "8081:8081"

depends\_on:

postgres:

condition: service\_started

**Appendix 2: hello\_world.yaml**

***YAML***

id: hello-world

namespace: simple\_dag

tasks:

- id: step1

type: io.kestra.plugin.core.log.Log

message: "Hello! Running first step"

- id: step2

type: io.kestra.plugin.core.log.Log

message: "Goodbye! Running 2nd and last step"

**Appendix 3: ingest\_data.yaml**

***YAML***

id: ingest\_data

namespace: ingestion\_of\_data

inputs:

- id: run\_date

type: STRING

tasks:

- id: download\_file

type: io.kestra.plugin.core.http.Download

uri: https://d37ci6vzurychx.cloudfront.net/trip-data/yellow\_tripdata\_2025-01.parquet

- id: move

type: io.kestra.plugin.scripts.shell.Commands

taskRunner:

type: io.kestra.plugin.core.runner.Process

commands:

- mv {{ outputs.download\_file.uri }} /app/data/yellow\_taxidata\_{{ inputs.run\_date }}.parquet

- id: ingest\_data

type: io.kestra.plugin.scripts.python.Commands

commands:

- python /scripts/ingest\_script.py

retry:

type: constant

interval: PT10S

maxAttempt: 2