Configuration Manual

MSc Research Project
MSc in Cloud Computing

Priya Shanmugam
Student ID: 23273518

School of Computing
National College of Ireland

Supervisor: Shreyas Setlur Arun

National College of Ireland

MSc Project Submission Sheet

School of Computing

Student Name:	Priya Shanmugam		
Student ID:	23273518		
Programme:	MSc in Cloud Computing	**Year:**	2024-25
Module:	MSc Research Project		
Supervisor:	Shreyas Setlur Arun		
Submission Due Date:	11.08.2025		
Project Title:	Real-Time, Cost-Aware Resource Scheduling in Multi-Cloud Systems Using PPO-Based Reinforcement Learning		
Word Count:	2565	**Page Count:**	15

Signature: Priya Shanmugam ……………………………………………………………………………………

Date: 07.08.2025 ……………………………………………………………………………………

PLEASE READ THE FOLLOWING INSTRUCTIONS AND CHECKLIST

Attach a completed copy of this sheet to each project (including multiple copies)	□
Attach a Moodle submission receipt of the online project submission, to each project (including multiple copies).	□
You must ensure that you retain a HARD COPY of the project, both for your own reference and in case a project is lost or mislaid. It is not sufficient to keep a copy on computer.	□

Assignments that are submitted to the Programme Coordinator Office must be placed into the assignment box located outside the office.

Office Use Only	
Signature:	
Date:	
Penalty Applied (if applicable):	

Configuration Manual

Priya Shanmugam
Student ID: 23273518

1 Introduction

This setup guide follows the steps involved to simulate and test a reinforcement learning based cloud task scheduler on iFogSim2 and Proximal Policy Optimization (PPO). The overall goal of the project is to apply Deep Reinforcement Learning to execute an intelligent, SLA-aware scheduling mechanism in a multi-cloud environment as well as to compare the application with traditional scheduling methods.

There is interaction among the following components as the system architecture:

- **iFogSim2 (Java):** It is the simulation engine; it is used to create cloud tasks and perform a simulation execution of them using heterogeneous cloud providers (e.g. AWS, Azure and GCP).
- **Python PPO Agent**: The agent is a Deep RL agent, trained to make cloud selection in real-time by Python PPO Agent using Stable-Baselines3 library.
- **Socket Interface:** Allows both the Java-based simulator and the python agent to communicate in real-time to exchange the state-action.
- **Streamlit Dashboard:** is a custom-built interface through which the simulation outputs include SLA compliance rate, task distribution, and cost metrics could be visualized.

This manual will aim to instruct the reader through a process that they will follow to install, configure and execute the simulation model so as to be able to replicate the simulation setting and test various scheduling algorithms, such as PPO, A2C, DQN, FCFS, and Round Robin.

2 Hardware and Software Requirements

This section outlines the essential hardware and software environment used to implement, train, simulate, and evaluate the proposed PPO-based cloud task scheduler.

2.1 Hardware Specifications

Component	**Specification**
Operating System	Windows 10 Pro (64-bit) / Ubuntu 20.04 LTS
Processor	Intel Core i7, 11th Gen / Ryzen 7
RAM	16 GB DDR4
Storage	512 GB SSD

2.2 Software Environment

Tool/Software	**Version**	**Purpose**
Java SDK	OpenJDK 1.8+	Required for running iFogSim2 simulator
Python	3.10+	Backend for PPO agent, logging, dashboard
Eclipse	2023.1+	Java IDE for editing and running iFogSim2
VS Code	1.82+	Python development and code editing
Google Colab	Latest (Online)	Used for training PPO model on uploaded datasets
Streamlit	1.34+	Dashboard for visualizing results
Stable-Baselines3 [1]	2.2.1	PPO training using Gym-style environments [1]
Pandas [7] / **NumPy**[8]	Latest	Data preprocessing and logging
Matplotlib[9] /**Seaborn** [10]	Latest	Plotting evaluation results

2.3 Additional Libraries and Dependencies

All Python libraries were installed using pip:

```
pip install stable-baselines3 pandas numpy matplotlib seaborn streamlit
```

The dependencies of java were handled through the iFogSim2 project configuration and included cloudsim.jar, commons-math3 and gson dependencies.

3 Environment Setup

The project infrastructure will consist of two components (Java-based cloud simulation (iFogSim2) [2] and Python-based PPO model training and decision-making). The environs set up is such to allow real-time socket communication between these two elements. The work was spaced out in Eclipse IDE (Java), Visual Studio Code (Python) and Google Colab [3] (to train a model and conduct experiments on demand).

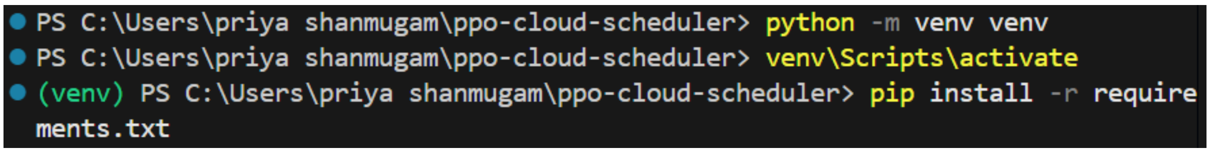
3.1 Python Environment Setup

The isolation of the dependencies was accomplished by creating a specific Python virtual environment via venv. The needed libraries were installed based on a requirements.txt file that had the following:

```
streamlit
pandas
plotly
matplotlib
seaborn
numpy
```

Command used:

```
python -m venv venv
source venv/bin/activate # On Unix or MacOS
venv\Scripts\activate     # On Windows
pip install -r requirements.txt
```

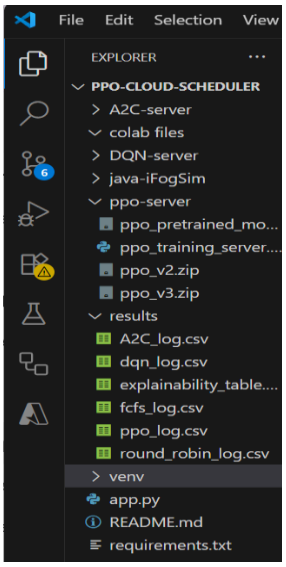


```
● PS C:\Users\priya shanmugam\ppo-cloud-scheduler> python -m venv venv
● PS C:\Users\priya shanmugam\ppo-cloud-scheduler> venv\Scripts\activate
● (venv) PS C:\Users\priya shanmugam\ppo-cloud-scheduler> pip install -r require
  ments.txt
```

3.2 Project Structure

The project was organized in the following directory layout:

- ppo-server/ → Python PPO training + socket server
- DQN-server/ and A2C-server/ → Additional DRL model servers
- java-iFogSim/ → Java simulator integrated with socket client
- results/ → Log files (CSV) of task metrics, used for dashboard evaluation
- app.py → Streamlit-based visualization dashboard
- requirements.txt → Dependency list for Python



```
PPO-CLOUD-SCHEDULER
  > A2C-server
  v colab files
  > DQN-server
  > java-iFogSim
  v ppo-server
      ppo_pretrained_mo...
      ppo_training_server....
      ppo_v2.zip
      ppo_v3.zip
  v results
      A2C_log.csv
      dqn_log.csv
      explainability_table....
      fcfs_log.csv
      ppo_log.csv
      round_robin_log.csv
  > venv
    app.py
    README.md
    requirements.txt
```

3.3 Java Environment Setup (iFogSim2)

- Java simulation was set up via Eclipse IDE. iFogSim2 was imported as a Java project and external libraries required (CloudSim, JSON parsers) added to build path.

- Core simulation: `MultiCloudSchedulingSim.java`
- PPO integration: `PPOClient.java` (handles socket-based communication)

3.4 Google Colab

For training PPO models on larger datasets or testing alternative DRL algorithms (e.g., A2C, DQN), **Google Colab** was used. It allowed faster experimentation using GPU/TPU runtime environments without local installation hassles.

- Models were exported as .zip files (e.g., `ppo_v2.zip`)
- Trained models were then loaded in `server.py` for inference

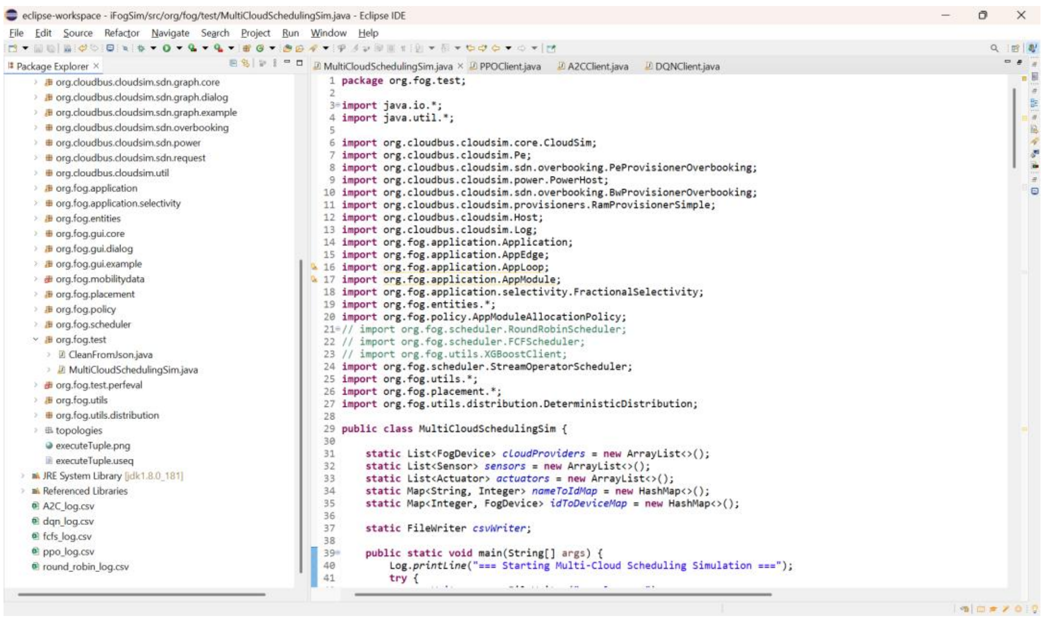
4 Step-by-Step Configuration

In this section, the whole step by step process to execute this hybrid PPO-based cloud scheduling system will be explained where iFogSim2 (Java) and Python (PPO agent) are connected to each other through sockets. Where this becomes necessary, code snippets and explanation are provided.

4.1 Clone & Set Up iFogSim2

Clone the forked iFogSim2 project of your own and/or pull it down yourself. JAVA simulation is executed using Eclipse.

`MultiCloudSchedulingSim.java`


eclipse-workspace - iFogSim/src/org/fog/test/MultiCloudSchedulingSim.java - Eclipse IDE
File Edit Source Refactor Navigate Search Project Run Window Help
Package Explorer
MultiCloudSchedulingSim.java
PPOClient.java
A2CClient.java
DQNClient.java

```
org.cloudbus.cloudsim.sdn.graph.core
org.cloudbus.cloudsim.sdn.graph.dialog
org.cloudbus.cloudsim.sdn.graph.example
org.cloudbus.cloudsim.sdn.overbooking
org.cloudbus.cloudsim.sdn.power
org.cloudbus.cloudsim.sdn.request
org.cloudbus.cloudsim.util
org.fog.application
org.fog.application.selectivity
org.fog.entities
org.fog.gui.core
org.fog.gui.dialog
org.fog.gui.example
org.fog.mobilitydata
org.fog.placement
org.fog.policy
org.fog.scheduler
org.fog.test
    CleanFromJson.java
    MultiCloudSchedulingSim.java
org.fog.test.perfeval
org.fog.utils
org.fog.utils.distribution
topologies
executeTuple.png
executeTuple.useq
JRE System Library [jdk1.8.0_181]
Referenced Libraries
A2C_log.csv
dqn_log.csv
fcfs_log.csv
ppo_log.csv
round_robin_log.csv
```

```java
package org.fog.test;

import java.io.*;
import java.util.*;

import org.cloudbus.cloudsim.core.CloudSim;
import org.cloudbus.cloudsim.Pe;
import org.cloudbus.cloudsim.sdn.overbooking.PeProvisionerOverbooking;
import org.cloudbus.cloudsim.power.PowerHost;
import org.cloudbus.cloudsim.sdn.overbooking.BwProvisionerOverbooking;
import org.cloudbus.cloudsim.provisioners.RamProvisionerSimple;
import org.cloudbus.cloudsim.Host;
import org.cloudbus.cloudsim.Log;
import org.fog.application.Application;
import org.fog.application.AppEdge;
import org.fog.application.AppLoop;
import org.fog.application.AppModule;
import org.fog.application.selectivity.FractionalSelectivity;
import org.fog.entities.*;
import org.fog.policy.AppModuleAllocationPolicy;
// import org.fog.scheduler.RoundRobinScheduler;
// import org.fog.scheduler.FCFScheduler;
// import org.fog.utils.XGBoostClient;
import org.fog.scheduler.StreamOperatorScheduler;
import org.fog.utils.*;
import org.fog.placement.*;
import org.fog.utils.distribution.DeterministicDistribution;

public class MultiCloudSchedulingSim {

    static List<FogDevice> cloudProviders = new ArrayList<>();
    static List<Sensor> sensors = new ArrayList<>();
    static List<Actuator> actuators = new ArrayList<>();
    static Map<String, Integer> nameToIdMap = new HashMap<>();
    static Map<Integer, FogDevice> idToDeviceMap = new HashMap<>();

    static FileWriter csvWriter;

    public static void main(String[] args) {
        Log.printLine("=== Starting Multi-Cloud Scheduling Simulation ===");
        try {
```

This file has the main simulation logic, which includes task generation and PPO client call.

Code Snippet – PPO Integration Logic

```java
double[] stateVec = generateStateVector(task);  // Normalized state vector
int selectedCloud = PPOClient.getPPOAction(taskId, stateVec, reward, done, nextState, cost, slaMet, slaDeadline, execTime);
assignTaskToCloud(task, selectedCloud);
```

On every new task, the simulator initially computes a normalized state vector (CPU, RAM, SLA etc.). This is sent to `PPOClient.getPPOAction()` which talks to the Python-based PPO server returning an action that matches a cloud provider (e.g. 0: AWS, 1:

Azure, 2: GCP). Then the work is sent to a chosen cloud. This dynamic scheduler substitutes the static policies such as the Round-Robin one.

4.2 Set Up Python Socket Server

You will require a Python [5] server that will wait and listen to instances of tasks vectors by the Java code and respond with the best choices of cloud in a trained PPO model.

Key code: ppo_training_server.py

```
# === Start Socket Server ===
server = socket.socket(socket.AF_INET,
socket.SOCK_STREAM)
server.bind((HOST, PORT))
server.listen(1)
print(f"✅ PPO Inference Server
running on {HOST}:{PORT}...")
```

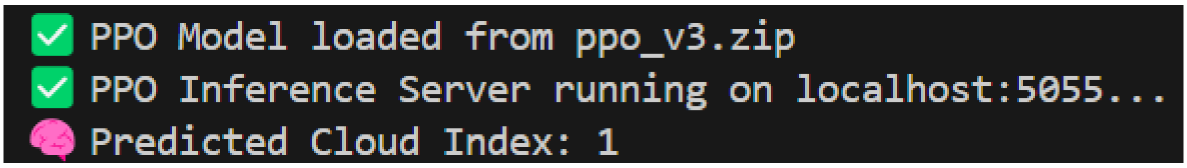
This causes a TCP server to start on port 5055 which is always listening for task state data inputs.

```
payload = json.loads(data)
state = np.array(json.loads(payload['state']),
dtype=np.float32)
```

Parses incoming Json and reads a state vector of the task.

```
action, _ = model.predict(state)
response = str(int(action))
client.send(response.encode())
```

The trained PPO model makes a guess on the optimal cloud index (0, 1, 2) and transmits it to the Java simulator.



```
✅ PPO Model loaded from ppo_v3.zip
✅ PPO Inference Server running on localhost:5055...
🧠 Predicted Cloud Index: 1
```

4.3 Java → PPO Communication via `PPOClient.java`

This Java class sends state vectors to the Python server and receives the cloud decision.

```
Socket socket = new Socket(SERVER_HOST, SERVER_PORT);
PrintWriter out = new
PrintWriter(socket.getOutputStream(), true);
BufferedReader in = new BufferedReader(new
InputStreamReader(socket.getInputStream()));
```

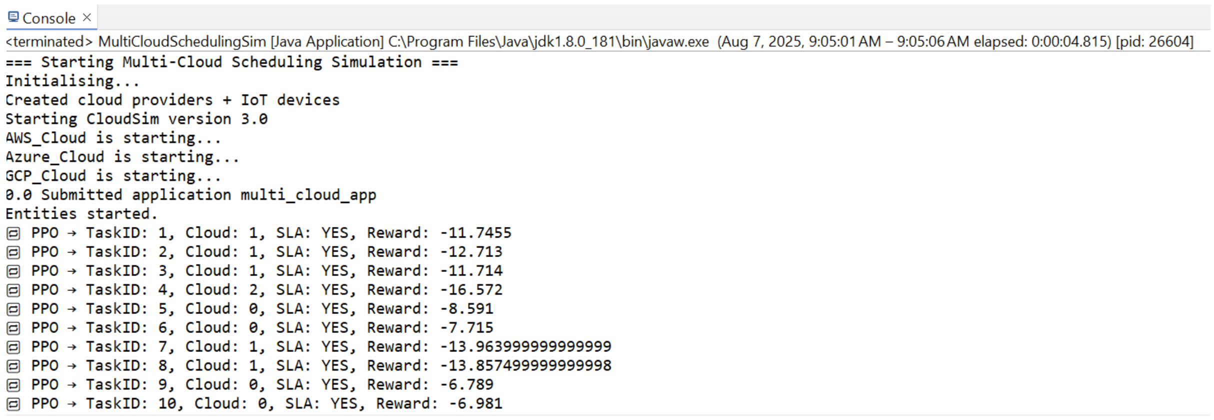
Connects to the Python server over TCP.

```
JSONObject payload = new JSONObject();
payload.put("state", stateToString(state));
...
out.println(payload.toJSONString());
```

When it is task-related data (state, reward, etc.), it is task-packet by encoding into a JSON and sending.

```
String response = in.readLine();
return Integer.parseInt(response);
```

Receives the selected cloud index (e.g., 0 for AWS) and returns it for task execution.



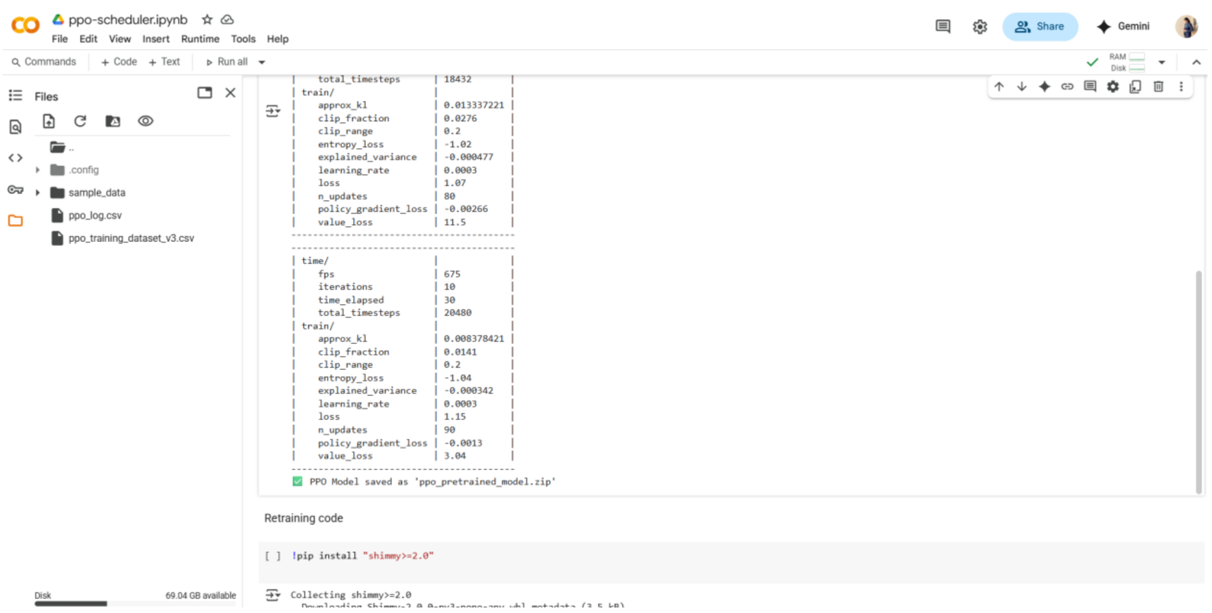
```
Console ×
<terminated> MultiCloudSchedulingSim [Java Application] C:\Program Files\Java\jdk1.8.0_181\bin\javaw.exe  (Aug 7, 2025, 9:05:01 AM – 9:05:06 AM elapsed: 0:00:04.815) [pid: 26604]
=== Starting Multi-Cloud Scheduling Simulation ===
Initialising...
Created cloud providers + IoT devices
Starting CloudSim version 3.0
AWS_Cloud is starting...
Azure_Cloud is starting...
GCP_Cloud is starting...
0.0 Submitted application multi_cloud_app
Entities started.
PPO → TaskID: 1, Cloud: 1, SLA: YES, Reward: -11.7455
PPO → TaskID: 2, Cloud: 1, SLA: YES, Reward: -12.713
PPO → TaskID: 3, Cloud: 1, SLA: YES, Reward: -11.714
PPO → TaskID: 4, Cloud: 2, SLA: YES, Reward: -16.572
PPO → TaskID: 5, Cloud: 0, SLA: YES, Reward: -8.591
PPO → TaskID: 6, Cloud: 0, SLA: YES, Reward: -7.715
PPO → TaskID: 7, Cloud: 1, SLA: YES, Reward: -13.963999999999999
PPO → TaskID: 8, Cloud: 1, SLA: YES, Reward: -13.857499999999998
PPO → TaskID: 9, Cloud: 0, SLA: YES, Reward: -6.789
PPO → TaskID: 10, Cloud: 0, SLA: YES, Reward: -6.981
```

4.4 Train PPO Model in Google Colab

You used an offline training on your PPO model using a Colab notebook and exported it to your computer so that it can be used to make inferences.[6]

```
model = PPO("MlpPolicy", env, verbose=1)
model.learn(total_timesteps=20000)
model.save("ppo_v2.zip")
```

The tuples of state-action, reward used in training the PPO agent are taken out of historical logs. This yields a model which is subsequently used in `ppo_training_server.py`.



```
|    total_timesteps      | 18432       |
| train/                  |             |
|    approx_kl            | 0.013337221 |
|    clip_fraction        | 0.0276      |
|    clip_range           | 0.2         |
|    entropy_loss         | -1.02       |
|    explained_variance   | -0.000477   |
|    learning_rate        | 0.0003      |
|    loss                 | 1.07        |
|    n_updates            | 80          |
|    policy_gradient_loss | -0.00266    |
|    value_loss           | 11.5        |
-----------------------------------------
-----------------------------------------
| time/                   |             |
|    fps                  | 675         |
|    iterations           | 10          |
|    time_elapsed         | 30          |
|    total_timesteps      | 20480       |
| train/                  |             |
|    approx_kl            | 0.008378421 |
|    clip_fraction        | 0.0141      |
|    clip_range           | 0.2         |
|    entropy_loss         | -1.04       |
|    explained_variance   | -0.000342   |
|    learning_rate        | 0.0003      |
|    loss                 | 1.15        |
|    n_updates            | 90          |
|    policy_gradient_loss | -0.0013     |
|    value_loss           | 3.04        |
-----------------------------------------
✅ PPO Model saved as 'ppo_pretrained_model.zip'
```

```
[ ] !pip install "shimmy>=2.0"
```

```
Collecting shimmy>=2.0
```

4.5 Running the End-to-End Simulation

Step 1: Start Python PPO Server

Open VS Code or terminal in the project folder and run:

```
python ppo_training_server.py
```

This activates inference socket server on `localhost:5055`.

Note: Ensure the model file `ppo_v2.zip` is placed in the correct directory and matches the path in the Python server.

Step 2: Run Java Simulation

In Eclipse, run:

```
MultiCloudSchedulingSim.java
```

This acts as the trigger to the simulation, and where each task is passed as the following:

Step 3: PPOClient Sends Data

```
int selectedCloud = PPOClient.getPPOAction(taskId, stateVec,
reward, done, nextState, cost, slaMet, slaDeadline, execTime);
```

This line sends the state to the Python server using socket communication and receives the optimal cloud index (0: AWS, 1: Azure, 2: GCP).

Step 4: Logging and Metrics

- After the action is accepted, task is performed on the chosen cloud.

➢ The result of the task (SLA Met, Cost, Execution Time) is logged in a CSV-file within the project folder.

5 Model Comparison Setup

In order to compare the PPO based-scheduler to other practices, we created four other models**: DQN, A2C, Round Robin, and FCFS** to compare. This provided a wholesome test between strategies of classical and deep learning structures.

5.1 DQN and A2C Integration

The models of DQN, and A2C were trained with the Stable-Baselines3 library in Google Colab. Learning was done with a historical task state vector dataset. These were exported as `.zip` files of the models (`dqn_v1.zip, a2c_v2.zip`) and moved to local project directories (`DQN-server/, A2C-server/`).

Each model has its own socket server similar to PPO's:

```
# Load trained DQN model
model = DQN.load("dqn_v1", custom_objects={"lr_schedule":
lambda _: 0.0003})

#  Predict cloud action from received state
state = np.array(request['state']).reshape(1, -1)
action, _ = model.predict(state, deterministic=True)

#  Send back cloud index (e.g., 0 for AWS)
response = {'action': int(action[0])}
conn.sendall(json.dumps(response).encode())
```

This logic takes care of the input in socket, extracts the task state, makes the model predicts the action of a best cloud, and submits the index of action to the Java simulation. Such servers listen on a certain port (`e.g., 9999 for DQN, 5055 for A2C`) and return the predicted cloud index.

In `MultiCloudSchedulingSim.java`, switching to DQN or A2C simply requires calling the respective client (e.g., `DQNClient.java`), which sends the state and receives the action just like PPOClient.

5.2 Round Robin and FCFS Integration

Unlike DRL models, **Round Robin** and **FCFS** strategies were directly coded inside MultiCloudSchedulingSim.java without any Python dependencies.

- **FCFS** logic simply executes tasks in the order they arrive and sends them to the first available cloud.
- **Round Robin** uses a rotating counter (e.g., `cloudIndex = (cloudIndex + 1) % numClouds`) to distribute tasks cyclically.

Java Code Snippet (Round Robin Logic)

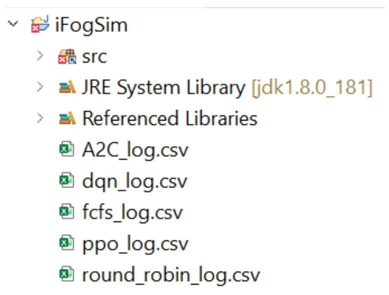
```java
public class RoundRobinScheduler {
    private static int rrIndex = 0;

    public static int
getNextCloudIndex(List<FogDevice> cloudProviders) {
        int index = rrIndex;
        rrIndex = (rrIndex + 1) %
cloudProviders.size();
        return index;
    }
}
```

This simple Java class makes use of modulo indexing to fairly assign tasks on a round robin basis to available cloud provider using available cloud providers.

Note: The FCFS is similar the implementation but is based on cloud availability time and queueing own policy instead of details round-robin cycling.

5.3 Output Logging

Each model logs results to a separate file inside project folder:

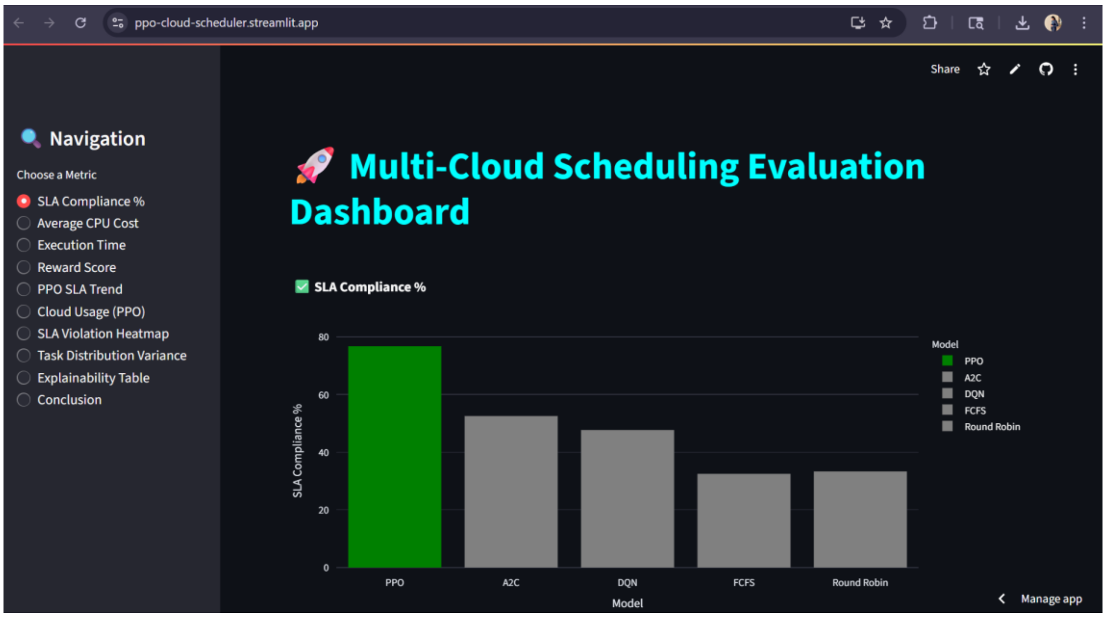


```
iFogSim
  src
  JRE System Library [jdk1.8.0_181]
  Referenced Libraries
  A2C_log.csv
  dqn_log.csv
  fcfs_log.csv
  ppo_log.csv
  round_robin_log.csv
```

Each CSV contains: `TaskID`, `SelectedCloud`, `SLAMet`, `ExecutionTime`, and `Cost`.These files serve as the basis for dashboard visualizations and SLA comparisons.

6 Dashboard Visualization

Streamlit was used to create an interactive dashboard to show and analyze the performances of different scheduling models (`PPO`, `A2C`, `DQN`, `FCFS`, `Round Robin`). It reads log files in the `results/ folder` and provides them with the possibility to navigate through different metrics making use of a sidebar.

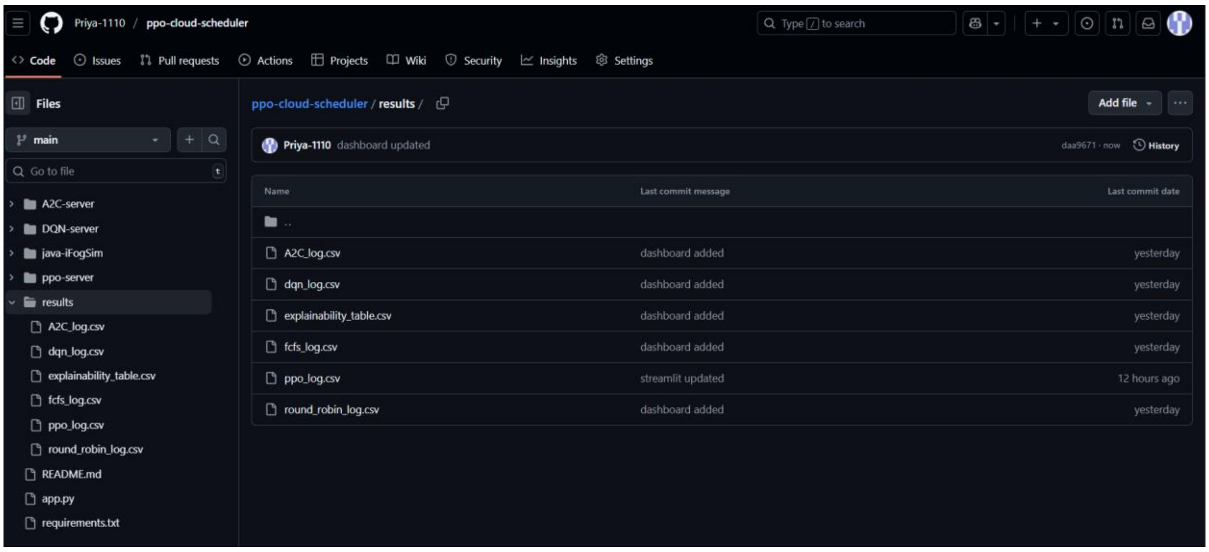


6.1 Deployment Steps

In this section, the implementation of the Streamlit-based cloud scheduling dashboard in the Streamlit Cloud service is also described, which allows one to check live results and visual comparisons.

Step 1: Prepare the GitHub Repository

Ensure your GitHub repo has the following structure:



```
A2C-server
DQN-server
java-iFogSim
ppo-server
results
    A2C_log.csv
    dqn_log.csv
    explainability_table.csv
    fcfs_log.csv
    ppo_log.csv
    round_robin_log.csv
README.md
app.py
requirements.txt
```

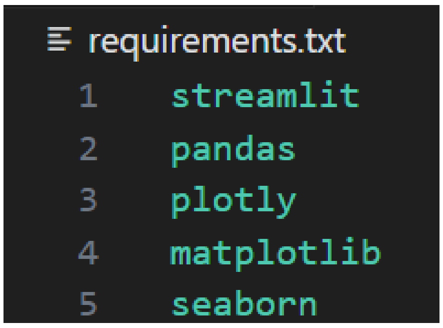
ppo-cloud-scheduler / results /

Priya-1110 dashboard updated daa9671 · now

Name	Last commit message	Last commit date
..		
A2C_log.csv	dashboard added	yesterday
dqn_log.csv	dashboard added	yesterday
explainability_table.csv	dashboard added	yesterday
fcfs_log.csv	dashboard added	yesterday
ppo_log.csv	streamlit updated	12 hours ago
round_robin_log.csv	dashboard added	yesterday

Step 2: Create requirements.txt

Include all necessary packages:



requirements.txt

```
streamlit
pandas
plotly
matplotlib
seaborn
```

This ensures Streamlit Cloud installs all required libraries when deploying.

Step 3: Deploy to Streamlit Cloud

1. Go to: https://streamlit.io/cloud [4]
2. Sign in with your GitHub account.
3. Click "New app".
4. Choose the repository and branch (e.g., main or master).
5. Set the main file path to app.py.
6. Click Deploy.

Step 4: Verify & Share

Once deployed, Streamlit Cloud will:

- Install the dependencies from requirements.txt
- Launch app.py
- Host your dashboard at a **public shareable URL** like:

https://ppo-cloud-scheduler.streamlit.app/

7 Troubleshooting and Fixes

This section highlights frequently encountered errors during setup and simulation, along with clear solutions to resolve them.

	Issue	Error Message / Symptom	Cause	Fix / Solution
1	Java socket connection	`java.net.ConnectException: Connection refused`	Python server not started before Java simulation	Run Python server (ppo_training_server.py) first, then run Java simulation
2	Python module import	`ModuleNotFoundError: No module named 'stable_baselines3'`	Dependencies missing	Run: pip install -r requirements.txt or install manually
3	Model not loading	`AttributeError: 'NoneType' object has no attribute 'predict'`	Model path incorrect or file missing	Verify model file (e.g., ppo_v2.zip) exists and PPO.load() path is correct

4	JSON decode / parse error	`json.decoder.JSONDecodeError`	Java client sent malformed JSON	Check JSON formatting in PPOClient.java; log payload before sending
5	Incorrect state dimensions	`ValueError: cannot reshape array of size X into shape (1,Y)`	State vector length mismatch	Ensure state sent from Java matches expected STATE_DIM in Python (usually 5)
6	Streamlit not launching	Nothing happens after streamlit run app.py	Streamlit not installed or blocked by firewall	Run: pip install streamlit and allow through firewall if prompted
7	Wrong port mismatch	Python listening on PORT=5055, Java using different port	Port mismatch between Java and Python	Ensure both Java PPOClient.java and Python server use the same host and port
8	Dashboard missing charts	No output / chart shows up in Streamlit	Metric function failed silently	Check log CSVs for missing columns or incorrect data types (e.g., NaN in SLAMet)

8 Conclusion

This configuration guide captured the entire process of setting up and testing the use of the Deep Reinforcement Learning (DRL) in the multi-cloud simulation environment to demonstrate real-time use of cloud scheduling. Setting up the iFogSim2 Java simulator, Python training and deployment of PPO-based socket servers, every process has been enumerated with relevant code snippets, and corresponding screen-shots, with solutions to troubleshooting issues.

The design is extremely modular:

- DRL models (PPO, DQN, A2C) perform in the form of autonomous Python servers and can be replaced effortlessly or trained in an alternative way.
- Java manages the simulation and state creation, and when changing the scheduling strategy, little changes need to be made.
- Logs are exported and stored as CSV, which is friendly to additional analysis or real-time dashboards.
- The Streamlit-powered dashboard is set to eventually load new logs to visualize as it does not associate with any future models.

This arrangement may be re-used, or may be extended to encompass:

- New algorithms of scheduling (e.g., Actor-Critic, SARSA)
- External APIs Prediction of the cost or detection of anomalies

- Ability to be deployed at scale through cloud system (ex. Streamlit Cloud or Docker)

With the logic decoupled in the form of simulation, decision-making and visualization modules, this framework has simple to extend framework to experiment in intelligent cloud orchestration in the future.

9 References

[1] DLR-RM (2023) *Stable-Baselines3: Reliable implementations of reinforcement learning algorithms in PyTorch*. GitHub.

[2] Mukherjee, M., Shu, L. and Wang, D. (2021) 'iFogSim2: An extended iFogSim simulator for mobility, clustering and microservices in edge computing', *IEEE Internet of Things Journal*, 8(3), pp. 2210–2223.

[3] Google Research (2025) *Google Colaboratory*.

[4] Streamlit Inc. (2025) *Streamlit – The fastest way to build and share data apps*.

[5] Python Software Foundation (2025) *Python Language Reference, version 3.10*.

[6] OpenAI (2023) *Gym: A toolkit for developing and comparing reinforcement learning algorithms*. GitHub.

[7] McKinney, W. (2010) 'Data structures for statistical computing in Python', in *Proceedings of the 9th Python in Science Conference*, pp. 51–56.

[8] NumPy Developers (2025) *NumPy*.

[9] Hunter, J.D. (2007) 'Matplotlib: A 2D graphics environment', *Computing in Science & Engineering*, 9(3), pp. 90–95.

[10] Waskom, M. (2025) *Seaborn: Statistical data visualization*.