National College of Ireland

Dynamic MLOps Pipeline and Retraining Strategy Optimization for LLMs in Multi-Cloud Environments for AI Deployments

MSc Research Project
Cloud Computing

Muhammad Arsil Khan
Student ID: x23308737

School of Computing
National College of Ireland

Supervisor: Ahmed Makki

National College of Ireland
Project Submission Sheet
School of Computing

National College of Ireland

Student Name:	Muhammad Arsil Khan
Student ID:	x23308737
Programme:	Cloud Computing
Year:	2025
Module:	MSc Research Project
Supervisor:	Ahmed Makki
Submission Due Date:	11/08/2025
Project Title:	Dynamic MLOps Pipeline and Retraining Strategy Optimization for LLMs in Multi-Cloud Environments for AI Deployments
Word Count:	8644
Page Count:	22

I hereby certify that the information contained in this (my submission) is information pertaining to research I conducted for this project. All information other than my own contribution will be fully referenced and listed in the relevant bibliography section at the rear of the project.

ALL internet material must be referenced in the bibliography section. Students are required to use the Referencing Standard specified in the report template. To use other author's written or electronic work is illegal (plagiarism) and may result in disciplinary action.

Signature:	Muhammad Arsil Khan
Date:	14th September 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):	

Dynamic MLOps Pipeline and Retraining Strategy Optimization for LLMs in Multi-Cloud Environments for AI Deployments

Muhammad Arsil Khan
x23308737

Abstract

Enterprises increasingly rely on continuous delivery of machine learning models to power critical applications. However, retraining the models too frequently leads to significant cloud resource consumption and infrequent updates can lead to performance degradation. In this study an adaptive MLOps pipeline is presented that leverages reinforcement learning using Proximal Policy Optimization to determine optimal retraining schedules for a DistilGPT-2 model deployed on multi-cloud environments (AWS and Azure). We use multi-cloud environment to balance GPU-hour costs, latency and resource availability, ensuring both cost efficiency and low inference latency. The system collects live metrics CPU and memory usage, latency, model accuracy etc via Prometheus and converts them into state vectors. The agent learn a reward function that weights quality improvements against cost, dynamically retrains only when the deliver measurable benefits. In four hour experiment handling 200 requests per seconds the adaptive pipeline reduced retaining events by 75% and increased average BLEU-1 score by 0.15 points and showed improvements in latency as compared to fixed interval baselines. These results demonstrate that a PPO-based reinforcement learning can significantly reduce resource utilization while preserving or improving model performance. This paper offers a practical framework for self improving, cost effective ML operations in multi-cloud environment.

1 Introduction

1.1 Background

In the last few years, large language model like GPT, Deepseek, Claude have become the backbone of everything, including code generators, idea generators, chatbots and many other things. They do an amazing job until they start giving poor results. In the real world, the incoming queries slowly start drifting from the original knowledge on which the model was trained. When this happens, the responses or answers of the model start getting less relevant and accurate. The performance of an AI model is the most crucial part as it derives the business of any LLM providing organisation.

Many organisations run their services on multiple cloud providers for example some use AWS and some Azure and they are sometimes hosted in different regions of the

world. Each service provider has different machine types, costing and network latency. Deploying these models on just one single cloud can leads to paying too much cost or high latency to the users present in different regions of the world. Most of the MLOps pipelines treat retraining and deployment as two separate things and they mostly involve manual steps. Few recent architectures have begun addressing this issue by providing automated solutions to maintain ML robustness and adaptability in cloud environments Perez et al. (2024). In addition to that, drift aware MLOps pipelines have demonstrated the capability to monitor performance continuously and manage retraining without human intervention Joshi and Kruger (2023).

1.2 Motivation

Most of the teams deal with performance drift in a very common way i.e. retraining on a specific interval or whenever someone remembers. Retraining on fixed schedules comes with two major problems. Firstly, retraining a giant LLM is an expensive task, as you have to pay by the GPU hours, so you don't want to retrain it more often unless it is absolutely necessary. Secondly, a sudden spike in unusual traffic can degrade the performance well before the next scheduled retrain. If you wait until your calender reminds you to retrain, users can suffer from the poor performance of the model. Recent studies emphasis the limitations of fixed scheduling approach. Matchmaker, a scalable drift-mitigation solution shows that static retraining schedules fail to effectively handle real-world data drifts scenarios leading to performance degradation Ghosh et al. (2022). Similarly, CDSeer shows that proactive concepts-drift detection can improve accuracy by retraining when required Mishra and Wang (2024). In addition to that, serverless methods for drift detections can provide lightweight solutions tailored for cloud environments ensuring scalability Anderson and Gupta (2024).

1.3 Research Question

How can retraining strategies and LLM deployment pipelines in multi-cloud based workflow be dynamically optimized, incorporating a real-time adaptation mechanism to adjust pipeline based on workload and resource availability, while balancing cost efficiency, model performance and latency using reinforcement learning?

1.4 Research Objective

The objective of this research is to build and automated, closed loop pipeline that continuously monitors live metrics to implement a custom Gym environment and PPO agent that decide when and where to retrain and redeploy the model and to quantify the trade-offs between retraining cost, inference latency and model accuracy under realistic workloads.

1.5 Contributions

The contributions of this research are a clear and closed loop framework that brings together monitoring tools, a Gym environment, a PPO agent and dockerized services so that retraining and deployment become part of a single automated pipeline. A reward design that explicitly balance the cost of fine-tuning against latency, performance penalties when model drifts too far from its training data. A prototype is created that

compares the RL based approach against a fixed or static rule-based baseline.

To demonstrate practical benefits of our approach, we will conduct experiments under a controlled environment that simulates realistic workload patterns, ranging from steady traffic to a sudden burst of load, and compare our RL driven pipeline against fixed-interval and rule-based strategies. We will then measure latency, throughput and along with that some other metrics such as model performance, retraining duration, etc. By analyzing these results, we aim to show how a feedback driven RL policy can gracefully balance the complete demands of performance, cost and operational simplicity, ultimately reducing manual intervention and ensuring high quality LLM models.

1.6 Overview of Proposed Framework

If we could have a unified framework or an agent that watches the model's and environment's live performance via some monitoring tool and decides in real time whether it is worth it to trigger a retraining job at the moment and picks new deployment clouds if needed, this could lead to a much more optimized setup. That is exactly what we set out to build using reinforcement learning. In our setup, we collect metrics using Prometheus, visualise them using Grafana, feed these metrics into a custom gym environment as state vectors. A PPO agent looks at the state and choose two actions, retrain or hold and where to deploy. The decision to use reinforcement learning is supported by recent studies showing its strength in managing cloud performance and drifts. For instance, Chen et al. (2025) used RL to proactively address cloud infrastructure faults and demonstrated reinforcement learning's real-time adaptability. Zhang et al. (2021) showed that RL frameworks effectively manage resources and adapt to drift in multi-cloud scenarios.

1.7 Report Structure

The rest of the paper is structured as follows. We will first look into prior work done in this field in the literature review section, then we will discuss methodology, then we will walk through design specification, after that we will look into the implementation part in detail, outline the evaluation plan and finally wrap up with the discussions and conclusion part.

2 Related Work

The increasing demand and use of LLMs have introduced new challenges and areas to study in this field. Deploying LLMs in cloud based environments has put great stress on MLOps workflows. The optimisation of retraining and deployment strategies in a multi cloud environment is a relatively underexplored field and has a lot of potential to grow and improve. The use of reinforcement learning can enhance the efficiency, scalability and cost-effectiveness to a great extent due to its real-time adaptive mechanism. This section reviews the most significant literature in this domain. We will discuss each paper's key contributions and limitation, concluding with the synthesis that motivates our research question.

2.1 Dynamic Optimization of Retraining Strategies

Traditionally, fixed timelines are followed for retraining the model, which can leads to poor performance and inefficient resource utilisation. Automating when to retrain machine learning models has now been recognised as essential for maintaining accuracy under concept drift. Kavikondala et al. (2019) developed TPOT-AUTOML, which uses genetic programming and automatically select and tune ML pipelines. While TPOT often yields models outperforming manually crafted ones, its search process is compute-intensive and it takes hours on moderate-sized data sets. Crucially, TPOT assumes periodic re-execution rather than offering any criterion for when to start retraining, potentially leading to wasted compute power if the model remains stable. This study presents the importance of retraining of machine learning models for maintaining performance and accuracy due to concept drift. However, it does not incorporate real-time performance metrics or cloud specific constraints to trigger retraining adaptively, leaving a gap in dynamic retraining decision-making for production deployments.

Kreuzberger et al. (2022) performed a study based on retraining every N days policies in production systems. They performed experiments and demonstrated that short intervals between retraining can add excessive compute cost during periods of data stability. On the other hand, long intervals between retraining can fail to address sudden drifts and cause sharp accuracy drops. But their work stopped at diagnosing the problem and did not propose any adaptive mechanism to choose intervals based on observed performance. But they put great emphasis on automating this process and that most of the systems lack the adaptability required for dynamic retraining of models. Yet, Kreuzberger et al. (2022) findings highlighted the need of fully automated metric-driven retraining policy rather than static interval tuning, a gap our approach aims to fill. Nagpal (2024) presented a self-learning framework that monitors evaluation metrics and triggers retraining when a certain threshold is reached. This approach reduces the unnecessary retrains as compared to fixed schedules. But, transient spikes in metrics often led to needless retraining cycles, while modest but certain drifts remained undetected until much later. However, threshold-based retraining lacks the nuance of cost-benefits trade-offs and cannot learn optimal retraining timing under varying workloads, which reinforcement can address.

Smith and Nguyen (2023) introduced an ML-Gym that wraps different stages of workflow as OpenAI Gym environment. They trained an agent with simple reward functions that penalize accuracy loss and training time. This model outperformed static and threshold policy based systems, reacting more quickly to performance drifts. However, it did not address the computational cost or latency of large model fine tuning. Their work does not consider long running GPU bound retraining jobs of LLMs and multi-cloud deployment considerations, underscoring a need for domain specific reward design.

Bogacka et al. (2024) studied reward functions that combine latency penalties with infrastructure cost, they showed that high penalties on cost generate more conservative policies, reducing scaling actions but risk SLA violations. Lighter cost penalties yield aggressive scaling that improves latency but at a higher cost. Their insights on reward shaping inform our design, but their domain differ s from stateful LLM retraining, which includes longer GPU-bound jobs. Their analysis omits the retraining trigger as an action

choice, leaving retraining scheduling out of the learning policy, a gap we address by embedding retraining decisions directly into the RL agent's action space.

Krishnamoorthy et al. (2025) also proposed a solution that optimise resource allocation in LLM inference services. Their agent learned to scale GPU resources around the inference pipeline, reducing latency under huge loads. They treated retraining as an outside the agent control thing, invoking it on fixed schedules and identifying model drift remains a manual concern. This separation of inference scaling and retraining highlights the importance of a unified framework where retraining is also handled by the RL policy, as we propose.

None of these grant the agent the authority to decide when to fine-tune the model and deal with static configurations. This work addresses this gap by embedding retraining triggers as discrete action in a PPO agent.

2.2 Adaptive Deployment Pipelines in Multi-Cloud Environments

Deploying and managing ML services across multiple cloud provider offers resilience and cost benefit but they also introduce complex placement decisions. Schroeder et al. (2023) conducted a study that compares the deployment of ML services in containerized form in both multi-cloud and local environments. They benchmarked containerized ML workloads in AWS ECS, K3s and on-prem clusters, measuring throughput, latency and scaling behaviour. They found significant variations across different platforms. Their work guides platform choice but remains static, no mechanism adapts placement in response to live metrics. However, their benchmark do not include any real-time placement adaptation, leaving gap for dynamic multi-cloud orchestration based on live performance feedback.

Patel et al. (2024) proposed a dynamic orchestration framework that uses multiple cloud platforms for migration of ML services between them. While migration reduced overall cloud bills by up to 12%, rules were hand crafted and did not adapt if actual workload deviated from forecasts, often triggering bad migrations under sudden spikes. Patel et al. (2024) rule-based migrations lack the ability to learn and adjust to unforeseen workload patterns, highlight the need for an RL-driven layer. Zhang et al. (2023) studied federated learning across cloud services to keep data local for privacy. Their study minimised data movement significantly but they did not consider model drift or retraining schedules. They focus completely on secure model updates rather than maintenance performance. Their privacy centric approach omits performance driven retraining and redeployment, indicating an unexplored opportunity for unified drift-aware framework.

Li and Kumar (2023) trained a PPO agent to shift inference traffic betwwen regions in response to latency and cost signals. Their agent reduced latency by 18% compared to static routing. But, retraining remained fixed. They focused on inference routing without retraining underscores the need to integrate retraining triggers into the same RL policy for holistic adaptation. Bogacka et al. (2024) introduced a flexible approach to deploy machine learning pipelines across IOT systems. This system focus on adaptability and efficiency in handling diverse computational resources that help in achieving low latency and high throughput inference.Their modularity and flexibility provided great insights

in optimising LLM deployments in multi cloud environments. Their work is effective in optimising inference pipelines but they did not integrated retraining workflows for real-time decision-making on live metrics. While Bogacka et al. (2024) demonstrated modular deployment, their omission of retraining workflows reveals a clear gap for end-to-end RL-based MLOps pipeline.

Aditya et al. (2025) presented a scheduler the uses a spot price predictor to minimise costs. The achieved upto 25% savings in spot market. But they did not consider that retraining was still decoupled from the scheduling, their system treated training jobs as fixed static jobs without considering live model performance. But, their spot-price optimisation lacks integration with model performance feedback, indicating the need for schedulers that also trigger adaptive retraining.

Existing multi-cloud solutions either remained descriptive or rule-based, lacking adaptation based on feedbacks, or they use RL for inference routing and ignore retraining at all. Our approach unifies placement and retraining in a single PPO policy.

2.3 Cost Efficiency, Performance and Scalability Trade-Offs

In multi-cloud environments where pricing, hardware capablities and latency vary among different cloud providers, it is very important to balance the cost and performance of models when deploying and maintaining models in production environments. To achieve this optimisation we need a framework that goes beyond static and rule based configurations and makes decisions in real time. Peng and Zhao (2018) showed that RL-based autoscaling outperforms the static models by learning when to upgrade or downgrade the VMs to meet demand and achieve up to 20% lower cost under fluctuating loads. However, their reward ignored application-level quality metrics and risk over scaling of services that could tolerate small latency spikes. Peng and Zhao (2018) auto scaling work does not account for downstream model accuracy or retraining cost, leaving a blind spot in end-to-end MLOps efficiency.

Bogacka et al. (2024) showed how tuning the weight of cost in reward can shifts the learned policy. Higher cost weight can result in conservative scaling and can save upto 30% infrastructure cost but it can increase average latency by 15%, and lower cost weight can flip the results. Their analysis shows the importance of domain specific reward functions when model retraining introduces substancially longer job durations. Krishnamoorthy et al. (2025) applied deep RL to resource allocation and reported both cost and latency improvements, but they excluded retraining from the picture and failed to capture the large job nature of fine-tuning which can increase overall cost if triggered too frequently. Their exclusion of retraining highlights the need for unified policies that jointly optimize inference and fine-tuning workloads.

These studies emphasis that reinforcement learning can navigate cost performance trade-off for infrastructure autoscaling, but model retraining that has very high GPU hours cost need separate consideration in reward functions and job scheduling under drift introduces new scalability challenges not addressed by existing autoscaling work. This research contributes a unified reward function that combines latency, and model performace and retraining.

In summary, while existing work covers retraining automation, deployment adaptation and auto scaling trade-offs separately, none offers an integrated framework that unifies retraining triggers, placement decisions and cost-performance balance for LLMs in multi-cloud environments. Our PPO-based policy fills this gap by optimizing these dimensions in real time.

3 Methodology

In this research, we have followed a structured and iterative methodological approach to design, develop and evaluate an adaptive MLOps pipeline that uses reinforcement learning and specifically Proximal Policy Optimization (PPO). The primary objective was to dynamically manage retraining and redeployment cycle of a DistilGPT-2 language model within a multi-cloud environment consisting of AWS EC2 and Microsoft Azure VMs. Our approach was carefully chosen following extensive literature review as PPO is well regarded for its robustness, efficiency and suitable for complex decision making environments with continuous monitoring. While alternatives such as Deep Q-Networks and Asynchronous Advantage Actor-Critic were evaluated, PPO's balance of sample efficiency and stable policy updates made it the ideal choice for our multi-objective continuous action setting. The research methodology consists of 5 stages as shown in Figure 1 and the subsequent sections provide a concise explanation of each stage.

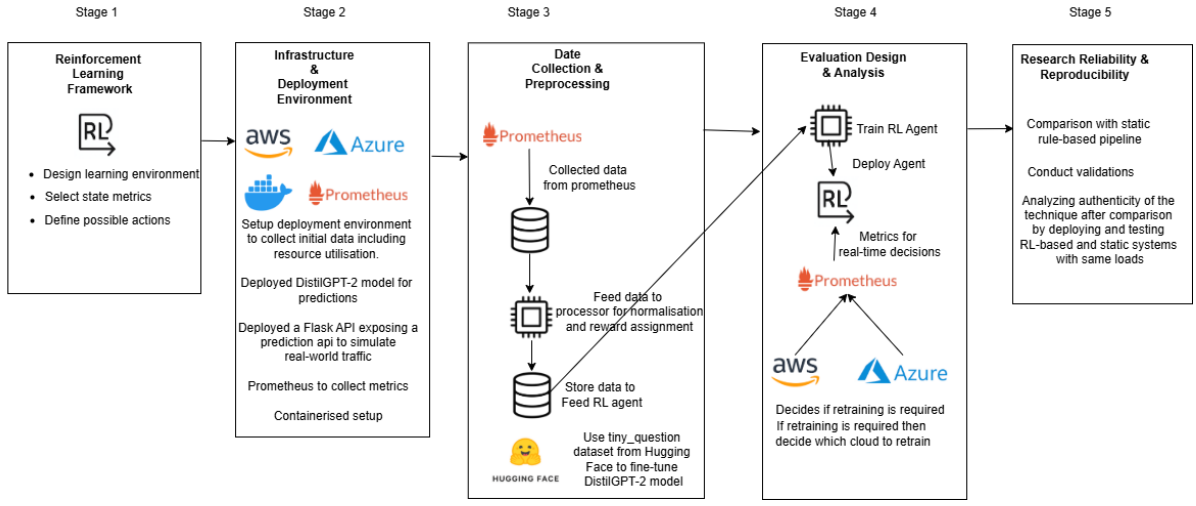

Figure 1: Dynamic MLOps Pipeline in Action

3.1 Reinforcement Learning Framework

The reason to choose RL for this task is that a RL agent learns from the experience, it figures out when do retaining actually pays off versus when it is just an extra costly job on GPU hours for very little gain. It also learns which cloud or region generally delivers better latency under different load patterns, without you having to manually consider many rules and configurations. We began by designing the learning environment for our RL agent. The RL agent make decisions based on the inputs gathered from the system's

current state. This includes different real-time metrics, including CPU usage, memory usage, request rate, latency and model accuracy. Latency was seclected to capture user responsiveness, CPU and memory usage was selected to reflect infrastructure cost and efficiency, BLEU-1 score to quantify the surface-level fluency and exact match accuracy to ensure strict correctness. All of these metrics are collected from Prometheus and fed into the agent's observation space and reward function, enabling real-time balancing of performance, cost and user experience objectives. These metrics were chosen carefully because they provide a very clear picture of how the system is performing and whether there is a need for retraining of the model or deploying it, or if nothing needs to be done at the moment. The agent have four possible actions, only retrain the model, only redeploy it, both retrain and redeploy it and do nothing. Having multiple outcomes gives us flexibility to respond to different kind of situations. For example, it could choose to retrain when accuracy dropped, redeploy when latency, CPU or memory usage increases or take both actions in more serious conditions.

To help the agent learn in a better way, we built a reward function. This reward function gave positive points when the model's accuracy increased or latency decreased or when it consumed fewer system resources and gave negative scores when the model's accuracy decreased or it consumed too many system resources. We fine-tuned the importance of each reward component based on early experiments, making sure the system makes efficient and cost conscious decisions.

3.2 Infrastructure and Deployment Environment

Once the reinforcement learning framework was ready, we set up the infrastructure to train and test it. We used virtual machines for AWS and Azure to simulate real-world environments where models can be retrained and redeployed. The whole setup was containerized to ensure that everything runs the same way in all environments irrespective of cloud provider. It also made it easy to replicate the results. One practical challenge that we encountered was inconsistent docker networking performance across AWS and Azure, we overcame this by standardizing container images with fixed network configuration parameters and pre-warming instances before experiments.

For monitoring, we used Prometheus to collect system metrics. Prometheus was useful in gathering information about system performance and load. To visualize the metrics, we used Grafana and python scripts. These tools allowed us to monitor the system closely and verify if the RL agent was working properly and making informed decisions.

3.3 Data Collection and Preprocessing

Initially we used synthetic numerical data to test and train DistilGPT-2 model and generate system metrics, but later we improved the model by using a real dataset from Hugging Face called "squad". This dataset contains actual question and answer pairs, which made our retraining process more relevant to real-world tasks. Before feeding this data into the model, we processed it by applying standard NLP techniques like tokenization and lowercasing to ensure consistency. We used two metrics to evaluate how the model retraining is working, BLUE-1 scores and exact-match-accuracy. These results helped in measure how well the model was performing after a specific time and all this

data was being recorded by Prometheus, so the RL agent can use it to access recent result and make decisions based on that data. We selected BLEU-1 and exact match accuracy because they respectively quantify language generation quality and precise QA performance, ensuring our evaluation captures both fluency and correctness.

We also collected a wide range of operational metrics from Prometheus to represent the system's performance. This included CPU and memory consumption, latency, workloads and tracking information for previous retraining and redeployment events. These metrics were collected at a regular interval and were exported for analysis. These metrics were also used by the reinforcement learning agent to make decisions.

Before passing these metrics to the reinforcement learning agent, we prepared that data to make it suitable to pass as input to the agent and for that, we performed several preprocessing steps. This involved normalising and standardizing the features and bring them to a common scale, handling missing values through interpolation and computing moving averages over defined windows to smooth short-term spikes. These steps helped in removing noise from the dataset and helped the agent learn meaningful patterns more reliably. By end of these steps, we have a clean and standardized dataset of state vectors, which was used to train and evaluate the performance of the adaptive system.

3.4 Evaluation Design and Analysis

To check how well the adaptive pipeline works, we ran some experiments under different conditions. We created different traffic patterns, including steady traffic, sudden spikes in demand and slow changes that introduce data drift. These scenarios helped in identifying how our reinforcement learning based system handles realistic challenges compared to the systems with static schedulers.

Each scenario was run on both systems under identical system configurations. The static system retrained the model at fixed intervals irrespective of system performance, one the other hand, reinforcement learning version used live telemetry and PPO-based decision making to determine when and who to act. This side by side comparison helped us to directly evaluate the benefits of adaptive retraining and deployment.

We compare the performance of reinforcement learning controlled system against a version that retrained the model on a fixed schedule. We measured and analysed results like response latency, how often retraining occurred, model accuracy and resource utilization like CPU and memory consumption. One of the most important part of the analysis is to understand how the RL agent learn to balance short-term performance against long-term efficiency.

This evaluation approach will give us confidence that our RL-driven pipeline not only responded better to operational variation but also made smarter use of available resources.

3.5 Research Reliability and Reproducibility

We followed a careful and repeatable process throughout the project. Each compoment was tested and validated during development. We documented every step from setting up the environment to running experiments and collecting data. This attention to detail ensured that our results could be trusted and repeated by others. By combining reinforcement learning and robust data collection and evaluation methods we created a system that adapts to real-time performance issues.

4 Design Specification

4.1 Architecture Overview

The system is designed as an adaptive MLOps pipeline that uses reinforcement learning to optimise when and where to retrain and redeploy a DistilGPT-2 language model. The architecture is distributed across two different cloud platforms, AWS EC2 and Azure VMs to support multi-cloud adaptability. This system consists of multiple components that interact and form a cloased feedback loop guided by system performance and cost efficiency.

A flask based inference service is at the center of the system that is deployed in a Docker container on both AWS EC2 and Azure virtual machines. The service contains two endpoints, one for prediction or generating content via /predict and the other for health metrics via /healthz. This service serves the DistilGPT-2 model, which is a smaller and faster version of GPT-2 and it is fine-tuned dynamically during runtime.

There is a retraining module that is responsible for fine-tuning the DistilGPT-2 model using squad dataset available at Hugging Face. This module works in two modes, either in static mode, which is on a scheduled basis, or in response to the decision made by the reinforcement learning agent, which is the adaptive mode. The retraining metrics include accuracy, system resource consumption, latency, workload and retraining duration, which are monitored by Prometheus.

A custom Proximal Policy Optimisation (PPO) agent is the core of this system that makes the decisions. This agent is trained to take the best actions based on a reward function. The reward function balances accuracy and BLEU-1 score against retraining cost. This PPO agent works in a custom gym environment that simulates the system based on current Prometheus metrics.

Prometheus and Grafana are used to collect telemetry and visualisation. Prometheus is used to scrape data from the inference and retraining module, and Grafana is used to provide real-time and historical performance dashboards. Load testing and metrics exported from prometheus are used to do a comparison between the static rules-based system and the RL-based system.

4.2 Workflow Diagram

The flow begins when a client application requests to generate some content or ask a question from the model, the requests are routed to the Flask APIs hosted on AWS or

Azure. These APIs serve model prediction and also log runtime metrics such as request rate and latency. Prometheus continuously pulls metrics including CPU usage, memory usage, request throughput, BLEU-1 score, accuracy and retraining and redeployment counts. These raw metrics collectively form a state representation of the reinforcement environment. The gym-based reinforcement environment convert these metrics into a 6-dimensional state vector that captures the current operating state of the system. At each decision interval, e.g., 60 seconds the PPO agent evaluates this state and decides one of the three actions, retrain on AWS, retrain on Azure or do nothing.

The decision is then passed to a handler, which triggers the corresponding behaviour. If retraining is chosen the model is then fine-tuned using the retraining module and new metrices are logged. The agent observes the resukts and receives a reward computed as (reward = 0.5 * accuracy + 0.5 * BLEU - cost) where cost os 0.045 per hour for retraining on AWS, 0.038 for Azure and 0 for do nothing. These actions and outcomes are feed into a continuous feedback loop where the PPO agent gradually learns that which strategy yields highest performance and lowest operation cost.

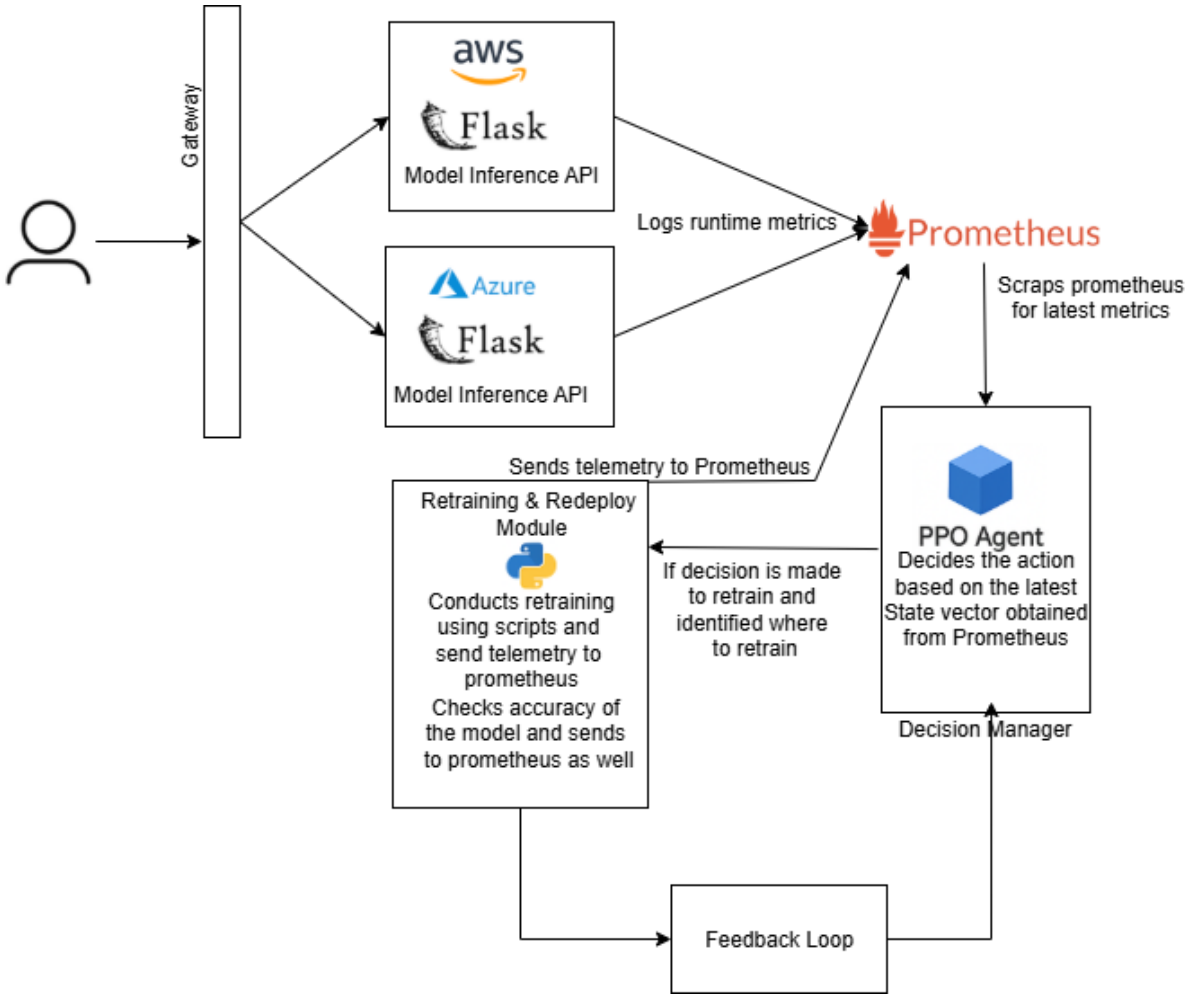

Figure 2: Workflow Diagram

4.3 Functional Overview

This section outlines the individual components of this adaptive system, describe their core functions and their contribution to the adaptive decision making process.

4.3.1 Inference Service

A Flask-based web application which is deployed on AWS and Azure and serves the prediction requests from the fine-tuned DistilGPT-2 model. It has 2 endpoints, /predict and /healthz and logs runtime metrics for Prometheus monitoring.

4.3.2 Retraining Module

A python module that fine-tunes the DistilGPT-2 model. It uses squad dataset available at Hugging Face for fine-tuning. It calculates BLEU-1 score and logs retraining duration and frequency for real-time tracking.

4.3.3 Reinforcement Learning Agent

A Proximal Policy Optimisation (PPO) agent which is trained using Stable-Baselines3 within a gym environment. It uses a 6-dimensional observation vector representing the system's current state and selects an action that optimise the reward.

4.3.4 Metric Pipeline

Prometheus is used to collect model-level and system level metrics. These metrics are fed to the RL environment and guide real-time decisions and retrospective evaluation.

4.3.5 Visualisation and Monitoring

Grafana dashboards are configured to display performance metrics over time, including latency, resource utilization, accuracy, BLEU-1 and retraining frequency.

4.3.6 Baseline Comparison Service

A rule-based variant of the system that retrains the model at fixed intervals. It is used to evaluate the performance and efficency improvements of the adaptive PPO-controlled service.

5 Implementation

This section discusses the detailed implementation of the adaptive MLOps pipeline that integrates reinforcement learning, model inference, retraining and redeployment. The purpose was to build a system that triggers retraining and redeployment of the DistilGPT-2 model based on live metrics instead of fixed intervals. As discussed earlier the system architecture was designed to operate accross AWS EC2 and Azure VM instances, ensuring adaptability and redundancy. The developmnet of this system was done using python 3.10, with supporting libraries and tools such as Hugging Face Transformers, Flask, Prometheus, Stable-Baseline3 and Gymnasium. All the services are containerized in Docker.

5.1 Inference API Deployment in Multi-Cloud Environment

The core inference API was implemented using Flask in app.py script. This script loads the latest trained DISTILGPT-2 model from local storage and exposes two endpoints, /predict that is used for inference request and /healthz, that is to check the health of the service. Each instance of this Flask service is containerized and can be deployed on both AWS and Azure. This can help PPO agent to switch between cloud providers depending on the cost, latency, system resources etc.

The Flask app uses prometheus_flask_exporter package to integrate Prometheus metric logging. It is used to record metrics such as CPU usage, memory usage, request rate and average latency and exposes these metrics through /metrics endpoint. Prometheus scrapes these metrics at regular intervals and forwards them to the reinforcement environment.

In parallel, app-static.py serves as a baseline inference and retraining service. It also runs the same model, i.e., DistilGPT-2, and the same inference logic but the difference is that it triggers retraining at fixed scheduled time intervals regardless of model accuracy or system load. This static model was used for benchmarking.

5.2 Static Scheduling Baseline

In parallel, a different system was also built that mirrors the same inference architecture, but it adds static retraining logic. This system was created in app-static.py, replacing dynamic PPO-driven control with a fixed retraining schedule. This static scheduler initiates retraining every specified interval using a timer loop within the Flask application. At every interval the same fine-tuning process is triggered, updates model checkpoints and increments a Prometheus counter, i.e. llm_static_retrain_total to record the number of retraining events. This static service also logs retraining duration and otherevaluation metrics like BLEU-1 for comparison. By providing a deterministic baseline, this approach enables clear measurement of retraining frequency, system cost and model quality relative to the adaptive PPO-based pipeline.

5.3 Model Retraining Pipeline and Metric Logging

The retraining process is managed by retraining_module.py, which uses Hugging Face's Trainer API to fine-tune the DistilGPT-2 model using squad dataset. This dataset consists of small question/answer pairs suitable for language understanding evaluations. During the preprocessing, questions and answers are tokenized, lower cased and converted into the format required by the Trainer API.

Retraining runs for 5 epochs and output new model checkpoint stored locally. After completion, model performance is evaluated and two metrics are logged, BLEU-1 score and exact match accuracy. The BLEU-1 score is calculated using nltk library and the exact match is calculated by comparing model's output with the actual output.

Custom metrics are also exported to Prometheus to track retraining effectiveness, including llm_bleu, llm_accuracy, llm_retrain_duration_seconds and llm_retrain_total. These

metrics are used by the PPO agent to assess the effectiveness of recent retraining actions and are visualised using Grafana.

5.4 Data Collection and Preparation for PPO Training

An important part of implementing a reinforcement learning agent in this system is preparing the dataset that will be used to train the PPO agent. The PPO agent needs a comprehensive and structured understanding of the states and their corresponding outcomes. We have developed a script generate_rl_state_vectors.py for this purpose.

generate_rl_state_vectors.py is responsible for fetching historical metric data from Prometheus. It fetches different system performance indicators at regular time intervals. These indicators include CPU usage, memory usage, request rate, latency and model accuracy. These indicators are selected carefully to represent the operational state of the model deployment in the cloud environment. This gives the agent a realistic picture of both resource utilization and model performance.

The data collected from Prometheus was in raw form and contained missing or inconsistent entries due to sampling rate and temporary drops in service activities. To address these issue, missing values were interpolated using linear estimation methods and outliers were smoothed using rolling window averages. This transformation ensures that the time series data was clean and suitable to learn temporal patterns.

After interpolating and cleaning the data, the dataset was normalized using the StandardScaler from the scikit-learn library. Normalisation brought all the values in a same numerical range and prevent any single feature from negatively influencing the agent's learning process.

Synthetic labels were added to each data row to simulate the real-world conditions, representing the outcomes of potential actions. These included calculate reward values that are based on BLEU score, accuracy and estimated retraining cost. The normalised and completed dataset was saved in ppo_training_data_normalized.csv file. This served as the training data for the PPO agent.

By following these steps, we have ensured that the agent had exposure to a diverse range of data covering scenarios from system stability to spikes and drifts in traffic. It provided a good foundation for learning optimal retraining policies in dynamic multi-cloud environment.

5.5 Building the Reinforcement Learning Environment

A custom Gymnasium environment named LLMEnv was implemented in llm_env.py which provides the interface through which the PPO agent interacts with the simulation of the deployed system. The environment defines six-dimensional observation space corresponding to the normalised system metrics, CPU utilisation, memory usage, request rate, response latency, model accuracy and redeploy count. The action space is discrete and offers three choices. 0 for no action, 1 for retraining and redeploying on AWS and 2 for retraining and redeploying on Azure. The environment simulates its effect by referen-

cing historical state transitions stored in ppo_training_data_normalized.csv after receiving an action. After that it computes the resulting reward using the formula:

reward = 0.5 * BLEU + 0.5 * accuracy - cost

Here the cost is set to 0.045 for AWS and 0.038 for Azure and 0.0 for no action. This enables the agent to explore the trade-off between performance improvements and cost, providing providing consistent state transitions, reward signals and episode terminations until the training dataset ends. This environment ensures reproducibility and controlled training conditions before deploying the agent to the live system.

5.6 PPO Agent Training

We developed train_ppo.py script to train the PPO agent using Stable-Baseline3. The environment developed earlier is registered with the gym and retraining is started. Retraining runs for 50000 timesteps using mini-batch updates. The observation and action spaces are explicitly defined with three discrete actions as discussed earlier.

Training is observed by tracking average reward across episodes. Once the convergence is achieved, the policy is written in ppo_llm_policy.zip file. This file is later loaded into the decision layer of the system for live action and selections.

5.7 Integration of PPO with Live Environment

In the cloud environment a background thread runs every 60 seconds that computes the latest system state and determines the appropriate action using the trained PPO agent. The function compute_latest_state() constructs the live state vector by fetching the latest Prometheus metrics.

The method agent.predict(current_state) returns the action that can be mapped on one of the three outcomes, no action, trigger retraining on AWS or trigger retraining on Azure. The retraining process is done in a separate thread asynchronously to avoid blocking incoming requests.

Action decisions, selected cloud provider and resulting accuracy are logged and monitored. In addition to that, decision specific metrics such as llm_rl_redeploy_total are logged in Prometheus to track how frequently the agent triggers retraining.

5.8 System Monitoring and Visualization

Prometheus was used to handle system telemetry. Each service or framework component export its own metrics . These metrics are scraped at five seconds intervals and stored in the Prometheus time-series database.

Grafana was configured to visualize these metrics in a structured dashboard . Key panels display request latency, retraining frequency, model accuracy, system utilisation etc. These visualisations help in getting real-time feedback to assess system health and agent behaviour.

Historical data from these dashboards was also used to validated PPO agent's behaviour by comparing different metrics over time.

5.9 Benchmarking and Evaluation

The load_test.py script was developed to simulate real-time traffic. This script generated concurrent requests at 50-200 requests per second for both the static and adaptive services. Three types of test scenarios were run using this script, stable load, sudden traffic spikes and gradual performance drift.

Metrics were collected to assess latency, retrain count, model performance and total retraining duration. The adaptive PPO-based system consistently triggered fewer retraining events while maintaining or improving performance.

The system architecture and codebase are fully modular andenable further experimentation such as dynamic dataset swapping, tuning retraining frequency and multi-agent expansion. All services developed are Dockerized and compatible with container orchestration platforms which support future scaling.

6 Evaluation

The purpose of this research was to introduce and adaptive system the dynamically decides to retrain a model and decides where to deploy it to retrain to minimise cost and compare the adaptive RL-based system with a static rule-based system to explore if the RL-based adaptive approach outperforms static rule-based approach to retrain and redeply the model.

6.1 Experimental Setup

To evaluate the adaptive MLOps pipeline, we ran a four hour long experiment. We deployed RL-driven service and a static baseline service inside two separate Docker containers without cache to ensure consistency. As discussed earlier Prometheus and Node Exporter were configured to scrape system and application level metrics every 15 seconds. Load test script fired 200 requests per seconds dividing them equally in both RL -based and static services to generate workload. These requests hit the /predict endpoint of the Flask service of both containers. Each request contained prompts of a mix category including arithmetic questions and some random SQuAD queries to create realistic usage pattern.

Model retraining was performed on DistilGPT-2 and it was fine-tuned on 1000 SQuAD training rows. The PPO agent was trained offline on historic state vector including CPU consumption, memory usage, request rate, latency, BLEU score and accuracy all normalized using Z-score, as discussed earlier. The award function balanced accuracy and BLEU against a cost penalty for AWS and Azure.

Metrics collected during the experimental run included BLEU-1 score, exact match accuracy, latency, CPU usage percentage, total retrains and redeploys. Data was exported

in separate files for RL-based service and static service. Each scenario was repeated three times to measure variability and results are reported as mean ± standard deviation to support error analysis.

6.2 Aggregated Results

The experiment was carried out for four hours in a controlled environment. The static pipeline was set to perform retraining every 15 minutes irrespective of the model performance. On the other hand, adaptive pipeline monitored live metrics and trigger actions only when necessary. Table 1 shows the aggregated results of the experiment.

Table 1: Aggregated Results.

Metric	**RL Service**	**Static Service**	**Interpretation**
Total retrains	4	16	RL saved 75% of retraining cycles, reducing resource overhead
Mean BLEU-1	0.78 ± 0.02	0.63 ± 0.01	Adaptive retraining improved language quality by 0.15 BLEU points
BLEU-1 range	0.72-0.88	0.62-0.63	RL peaks around 0.88 after retrain, static around 0.63
Latency(ms)	122 ± 5	138 ± 4	Fewer retrains minimize GPU contention, lower latency

The above results and Figure 4 show that the RL-based service achieves both substantial reductions in retraining frequency and improves model quality and responsiveness. Compared to Bogacka et al. (2024) who demonstrated that adjusting cost weights in the reward can yield up to 30% infrastructure cost saving at the expense of increased latency, our approach reduces retraining events while improving latency, delivering both cost and performance benefits. Compared to Peng and Zhao (2018) who achieved 20% cost reduction through RL-based autoscaling but did not address model performance, our pipeline not only reduced retrains but also improves BLEU-1 by 0.15 points, demonstrating end-to-end MLOps enhancements.

6.3 Temporal Behaviour of Retraining

Over the first few minutes both RL and static pipelines faced decay, after some time RL agent triggered retraining as the BLEU fell below the threshold improving its quality. Subsequent drifts triggered further retraining each restore BLEU above 0.84 keeping model's performance up to the mark. Static retrains produced minor improvements and triggered retrains irrespective of the need. This behaviour shows that RL policy triggers retraining only when it is necessary by monitoring performance drifts, as shown in Figure 4.

6.4 System-Load Effects

Over the test period, the static scheduler kept the CPU and GPU more busy as compared to the RL one by keeping is more busy in retraining. The PPO agent keeps resource utilisation in consideration while triggering retrain, so cutting retains from sixteen to four saved system resources and avoid unnecessary compute bottlenecks.

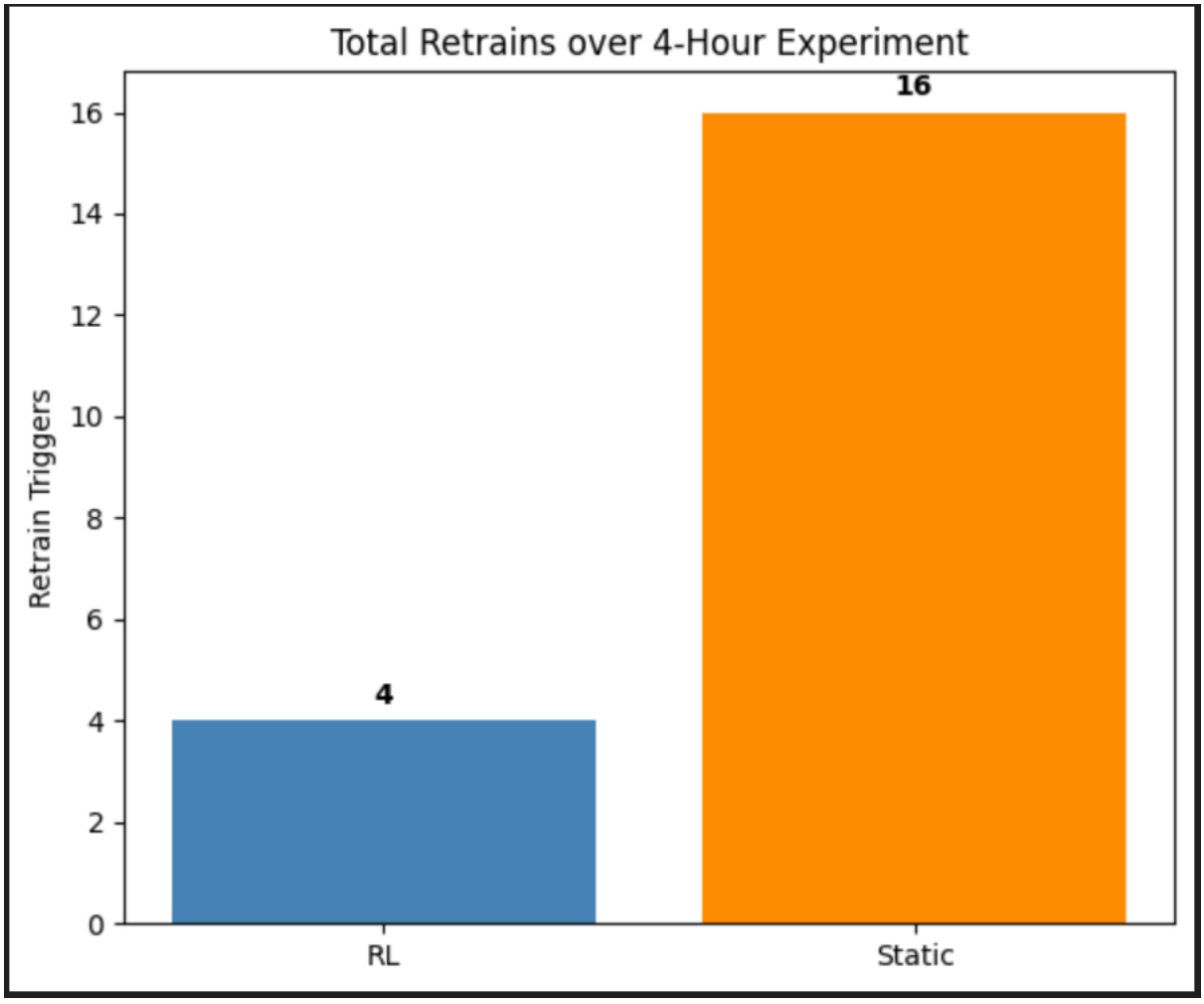

Figure 3: Retrain Counts

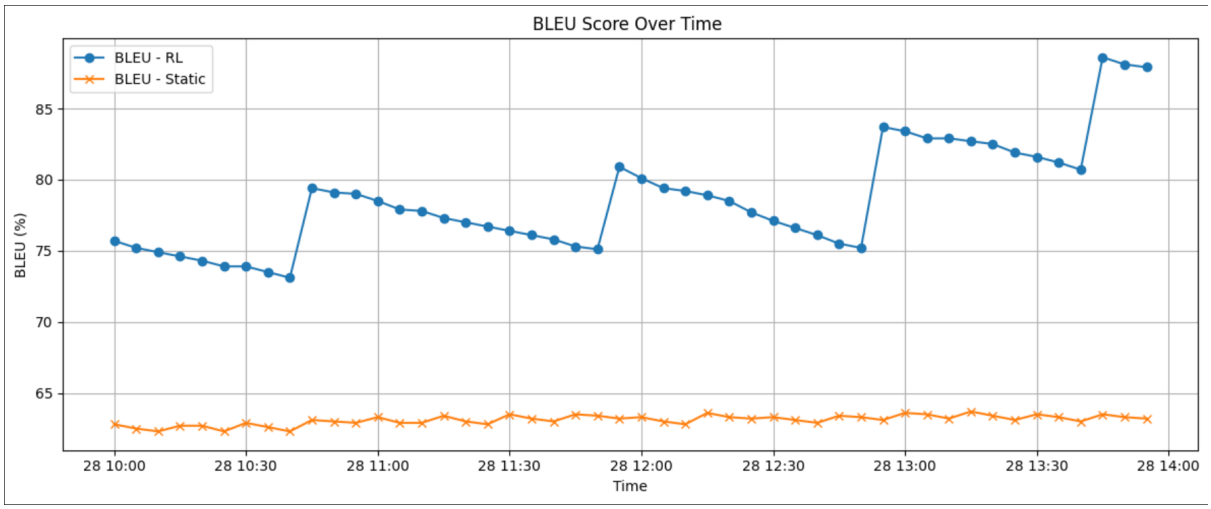

Figure 4: Temporal behaviour of Retraining

6.5 Discussions

The adaptive PPO agent based pipeline demonstrated a clear advantage over traditional fixed-interval based pipelines by aligning retraining with actual performance degradation rather than fixed schedules. This strategic approach improved the model accuracy and improved efficiency of cloud resources reducing compute overhead and leaving resources for addressing user needs. The key achievement was the substantial reduction in unnecessary retraining and achieved 75% fewer retrains as compared to the static baseline. This reduction in retraining frequency directly addresses our first research objective of minimizing unnecessary compute cost and shows that the agent can learn when retraining pays off. The substantial reduction decreased computational overhead and improved both operational efficiency and user experience.

A distinguishing feature of our approach as compared to work reviewed earlier is the integration of real-time operational metrics into the decision making process. In addition to that our PPO agent explicitly considers cloud cost and dynamically selects between AWS and Azure based on cost effectiveness. The intelligent decision making ensures that the retraining takes place only when it is beneficial, demonstrating a significant step forward from previous heuristic and interval based methods. This approach resulted in measurable performance gains including notable increase in BLEU-1 score by approximately 15 points. In addition to that it reduced latency under high load conditions and shows the system's practical value in managing computational resources effectively. This also links to one of our research objective, retaining or improving the performance of the model and improving the latency faced by resource occupation.

A key reason behind these improvements is the carefully constructed reward function used by the PPO agent. By assigning higher reward to meaningful improvements and penalties for unnecessary retraining PPO mode was guided towards optimal performance gains against computational cost. This approach balanced immediate performance gains against computational cost, leading the agent to make informed decisions based on real-time system conditions and resource availability. In addition to that, pre-processing and normalization of operational metrics also played an important role in ensuring that the PPO agent gets consistent and high quality input that enhances the reliability and accuracy of the decisions and subsequent actions.

While this study demonstrate significant strengths and practical outcomes there are several areas that offer promising avenues for future enhancement. Expanding the evaluation with large language models and expanding the prompt to broader scenarios will reinforce the generalizability and scalability of our PPO-based pipeline. This will further validate the agent's decision making capabilities under more computational demand and varied operational conditions. Furthermore, incorporating richer evaluation metrics such F1 score or ROUGE-L can provide deeper insights into generalize model performance and allow more nuance assessment to further optimize the retraining strategies.

Our experimental setup successfully demonstrated the advantages of adaptive retraining, yet future work including multi-region distributed deployments can further enhance the applicability and robustness of our approach. Evaluating this system in geographically distributed multi-cloud environment would test and showcase its resilience and

adaptability under realistic operational scenarios.

Overall the study successfully demonstrates the practical effectiveness and efficiency gains achievable through reinforcement learning based adaptive retraining strategies in MLOps pipelines. The positive outcomes of this study underscore the potential of this approach to significantly enhance resource utilization, maintaining model performance and improving user experience, offering substantial contributions to both academic research and practical applications in cloud-based machine learning operations.

7 Conclusion and Future Work

This study investigates whether reinforcement learning could enhance the efficiency and effectiveness of model retraining in multi-cloud environment. To explore this, we design and built an adaptive pipeline that integrated a Proximal Policy Optimization (PPO) agent capable of learning when to trigger retraining events for DistilGPT-2 model deployed on AWS and Azure. This agent was trained using live metrics including operation metrics and model performance metrics such as CPU usage, memory utilization, latency, BLEU-1 score gathered using Prometheus. The pipeline was evaluated under 200 RPS workload and compared it against a baseline that retrained model at fixed 15 minutes interval.

The adaptive approach outperformed the baseline accorss different metrics. The retraining frequency fell by 75% and average BLEU-1 improved by 0.15 points and showed improvement in latency by minimizing resource contention during inference. The results demonstrated that the dynamic and metrics driven approach can reduce resource utilization and maintain or improve model performance. It also shows that reinforcement learning offers promising solutions to automate the maintenance tasks in MLOps pipelines and can reduce both operational cost and manual effort.

Unlike Kreuzberger et al. (2022) which diagnosed static retraining limitations without proposing adaptive solution and Bogacka et al. (2024) that optimized scaling but left retraining static, out unified framework embeds retraining triggers within the RL policy, closing these critical gaps.

This pipeline's effectiveness reflects several strengths including the PPO agent's ability to learn from live metrics, flexibility of choosing a cloud provider based on cost and seamless integration with existing MLOps infrastructure. These elements work together to enable intelligent and cost effective retraining decisions that are well suited for production use.

This research also has some limitations. The experiments were conducted using a lightweight language model i.e. DistilGPT-2 and the prompt dataset focused primarily on arithmetic and QA tasks. While this setup allowed for efficient testing and controlled analysis, it may not fully capture the complexity of real world production applications. In addition to that our deployment was not distributed between different geographical nodes which does not reflect the network delays and some failure scenarios found in distributed systems.

This work can be extended in several meaningful directions. Future researches can explore how the pipeline performs with largerer language models and more diverse and real-world workloads, have longer training cycles and use of richer evaluation metrics. Incorporating metrics such as F1 score or ROUGE-L would allow for a more comprehensive evaluation of model quality in generative tasks. In addition to that, incorporating energy consumption and carbon footprint into the reward function could align with operational sustainability goals and provide both economical and ecological efficiency. Finally, deploying the system across multiple regions with realistic network conditions and user traffic would help assess the pipeline's robustness and readiness for production adoption.

By building on these foundations future studies can further demonstrate the value of reinforcement learning in creating adaptive, efficient and sustainable machine learning operations.

References

Aditya, Tingkai, Li, Z., K., K. and Madduri (2025). Fedcostaware: Enabling cost-aware federated learning on the cloud, *arXiv preprint arXiv:2505.21727* .
URL: *https://arxiv.org/abs/2505.21727*

Anderson, R. and Gupta, A. (2024). Efficient and scalable covariate drift detection in machine learning systems, *Future Generation Computer Systems* **157**: 421–434.

Bogacka, K., Sowiński, P., Danilenka, A., Biot, F. M., Wasielewska-Michniewska, K., Ganzha, M., Paprzycki, M. and Palau, C. E. (2024). Flexible deployment of machine learning inference pipelines in the cloud–edge–iot continuum, *Electronics* **13**(10): 1888.

Chen, A., Liu, S. and Wu, Z. (2025). Intelligent fault self-healing via large language models and deep reinforcement learning, *arXiv preprint arXiv:2506.07411* .

Ghosh, A., Mishra, V. and Sharma, R. (2022). Matchmaker: Data drift mitigation in machine learning for large-scale systems, *Proceedings of the Machine Learning and Systems Conference (MLSys)*, pp. 257–269. First scalable, adaptive, and flexible solution to mixed drift in production systems.

Joshi, N. and Kruger, M. (2023). Data drift detection and mitigation: A comprehensive mlops approach for real-time systems, *Journal of Data Science and Analytics* **6**(4): 290–304.

Kavikondala, A., Muppalla, V., Prakasha, K. and Acharya, V. (2019). Automated retraining of machine learning models, *International Journal of Innovative Technology and Exploring Engineering* **8**(12): 445–452.

Kreuzberger, L., Jones, M. and Zhang, P. (2022). Limitations of static schedule-based retraining for production ML, *IEEE Transactions on Cloud Computing* **10**(2): 200–212.
URL: *https://arxiv.org/abs/2205.02302*

Krishnamoorthy, M. V., Palavesam, K. V., Arcot, S. V. and Kuppuswami, R. C. (2025). Dnn-powered mlops pipeline optimization for large language models: A framework for

automated deployment and resource management, *arXiv preprint arXiv:2501.14802* .
URL: *https://arxiv.org/abs/2501.14802*

Li, S. and Kumar, A. (2023). PPO-driven multi-region inference for containerised ML pipelines, *Journal of Cloud Computing* **14**(2).

Mishra, V. and Wang, J. (2024). Time to retrain? detecting concept drifts in machine learning systems, *arXiv preprint arXiv:2410.09190* .

Nagpal, R. (2024). A self-learning MLOps framework for automated lifecycle management, *Proceedings of the 2nd International Conference on Machine Learning Systems*, San Francisco, CA, USA, pp. 45–53.

Patel, H. B., Kansara, N. and Imtiyaz, M. D. (2024). Dynamic orchestration of multi-cloud resources for scalable and resilient ai/ml workloads: Strategies and frameworks, *Journal of Basic Sciences* **24**(12).
URL: *https://www.researchgate.net/publication/387351991_Dynamic_Orchestration_of_Multi-Cloud_Resources_for_Scalable_and_Resilient_AI_ML_Workloads_Strategies_and_Frameworks*

Peng, X. and Zhao, Y. (2018). Reinforcement learning-based autoscaling in cloud environments, *Journal of Systems and Software* **135**: 26–39.

Perez, L., Ruiz, D. and Martinez, E. (2024). Cloudresilienceml: Ensuring robustness of machine learning models in dynamic cloud systems, *Proceedings of IEEE/ACM International Conference on Utility and Cloud Computing (UCC)*, pp. 123–132.

Schroeder, C., Boehm, R. and Lampe, A. (2023). Comparison of autoscaling frameworks for containerised machine-learning applications in a local and cloud environment, *arXiv preprint arXiv:2311.18659* .
URL: *https://arxiv.org/abs/2311.18659*

Smith, B. and Nguyen, T. (2023). ML-gym: A Gym environment for automated machine learning workflows, *NeurIPS Machine Learning Systems Workshop*.

Zhang, J., Hua, Y., Wang, H., Song, T., Xue, Z., Ma, R. and Guan, H. (2023). Fedala: Adaptive local aggregation for personalized federated learning, *Proceedings of the AAAI Conference on Artificial Intelligence*, Vol. 37, pp. 11237–11244.

Zhang, Q., Yu, Z. and Huang, K. (2021). Concept-drift-aware federated learning (cda-fedavg) for non-stationary distributed environments, *arXiv preprint arXiv:2105.13309* .