Cairo University
Faculty of Computers and Artificial Intelligence
Computer Science Department

Microservices Autonomous Event Surveillance, Troubleshooting & Recovery Operations (M.A.E.S.T.R.O)

Supervised By:

Dr. Desoky Abdelqawy

Implemented By:

20210377	**Marwan Ahmed Abdelfattah (Al)**
20210274	**Amr Khaled El-Hennawi**
20210363	**Mohamed Waleed Mohamed**
20210251	**Ali Aldeen Mohamed Hanafy**
20210507	**Zeyad Mohamed Maher Karsoun**

Graduation Project

Academic Year 2024-2025

Final Documentation

[Page is intentionally left blank]

Table of Contents

List Of Figures

Chapter 1: Introduction

1.1 Motivation

Today, most software is built using cloud-based technology. A popular way of building this software is called microservices architecture. Instead of creating one large program (monolith), this method breaks it down into many smaller parts called microservices. Each microservice does one job and can be built, changed, and used without affecting the others. This makes the software faster to develop, easier to manage, and better at handling lots of users. Companies in areas like banking, online shopping, and phone networks use microservices to improve their services.

One of the biggest challenges with microservices is what happens when something goes wrong. In traditional systems, a tool like **Kubernetes** can restart a failed microservice after it stops working or after a set threshold. But this process takes time. In apps that need to work all the time like online trading, hospitals, or live video this short delay can cause big problems. Users can get frustrated, systems can slow down, and sometimes one failure can cause many others, crashing the whole system. Since these systems react only after something breaks, they're always one step behind.

Our project aims to provide a solution to this challenge. Instead of reacting to failures, it predicts them before they happen. It uses a reinforcement learning model and system logs to spot early warning signs that something might fail. It looks at CPU, memory, latency, and traffic the service is using. These logs come from monitoring tools like Prometheus, which track the metrics of the system.

1.2 Problem Definition

Based on the motivation outlined, the problem addressed by this project can be defined as:

How to design and implement a smart and autonomous autoscaling system for containerized applications in Kubernetes that learns an optimal scaling policy to dynamically balance application performance (low latency) and resource cost (minimal pods), while remaining resilient to variable traffic loads and infrastructure instability.

This problem can be deconstructed into several key technical challenges that this work seeks to solve:

Inadequate State Representation: Standard autoscalers typically rely on a single, isolated metric such as average CPU utilization. This is an incomplete representation of an application's health. The problem is to define a richer state that includes not only resource metrics (cpu, memory) but also direct performance indicators (latency), traffic load (requests per second), and the current scale (number of replicas), providing a holistic view for decision-making.

Suboptimal Decision-Making Logic: The logic of threshold-based scaling is inherently limited and cannot adapt its strategy. The problem is to replace this rigid if-then logic with a sophisticated, learned policy. We formulate this as a Reinforcement Learning problem where an agent must learn to map observations of the system's state to one of three discrete actions: scale_up, scale_down, or no_op.

The Performance vs. Cost Trade-off: There is a direct and inherent conflict between maintaining a high-performance, low-latency service and minimizing operational costs by running fewer container instances. The problem is to explicitly model this trade-off. We address this by designing a reward function that provides positive feedback for staying within a target latency range and negative feedback (a penalty) for high latency and for each running pod. The agent's challenge is to maximize its cumulative reward, thereby learning to navigate this trade-off effectively.

System Latency in Scaling: The time it takes for a new pod to be scheduled, start, and begin serving traffic is non-zero. A purely reactive system that waits for metrics to cross a threshold is always behind the curve, as it must also wait for this "cold-start" period. The problem is to develop a system that can learn to act proactively, scaling out in anticipation of load to absorb traffic spikes without a preliminary period of performance degradation.

Environmental Volatility: Production environments are not stable. They are subject to unpredictable traffic patterns, "noisy neighbor" effects, and underlying infrastructure failures. An autoscaler trained in a perfect, simulated environment is likely to fail under real-world pressure. The problem is to create a system that is robust and resilient. This is addressed by training and testing the RL agent in an environment that simulates this volatility using traffic generators and chaos engineering tools.

By successfully addressing these sub-problems, this project aims to deliver a custom autoscaler that represents a significant advancement over existing solutions, providing a more efficient, intelligent, and resilient approach to managing modern cloud applications.

1.3 Project Objective

The primary objective of this project is to overcome the limitations of traditional, metric-based autoscalers by developing an intelligent, proactive, and resilient scaling system using artificial intelligence. During the initial planning phase, a solution combining classification and time-series forecasting was considered. However, an early attempt to gather the necessary training data revealed a significant obstacle: this supervised learning approach would require a vast and diverse dataset of system states and corresponding optimal actions, which would be exceedingly difficult to collect and label accurately.

This realization prompted a shift in strategy towards a more dynamic AI paradigm: Reinforcement Learning (RL). Unlike supervised learning, RL does not require a pre-existing dataset. Instead, the agent learns directly from real-time interaction with the environment. It receives a positive reward for actions that improve system performance and a negative reward for actions that degrade it. Through this continuous feedback loop, the agent autonomously learns the complex patterns and subtle signals that precede performance issues, allowing it to act preemptively without being explicitly programmed for every scenario.

This project, therefore, aims to engineer MAESTRO, a complete, end-to-end intelligent autoscaling system for Kubernetes that embodies this RL-first approach. The system leverages a Proximal Policy Optimization (PPO) agent to learn and execute a proactive scaling strategy. MAESTRO is not merely a theoretical model but a practical, deployment-ready system

composed of modular, containerized components designed to achieve the following specific objectives:

1. To Develop a Reinforcement Learning Agent for Proactive Autoscaling:

The core of MAESTRO is a PPO-based RL agent trained to understand the complex dynamics of an application environment. The agent maps a multi-dimensional state vector comprising [cpu_usage, mem_usage, n_replicas, latency, rps]to an optimal scaling action (scale_up, scale_down, or no_op). The success of this objective is measured by the agent's ability to maximize a cumulative reward signal that explicitly balances the competing goals of maintaining low latency and minimizing the replica count for cost efficiency.

2. To Design a Decoupled, Microservice-Based Architecture:

Rather than a monolithic controller, MAESTRO is built on a scalable and maintainable microservice architecture. This involves three distinct, containerized components managed by Kubernetes:

The RL Model Server: A dedicated inference server hosting the trained PPO model, exposing a simple API to provide scaling decisions on demand.

The Suggestion Server: An intermediary service that orchestrates the data flow between the controller and the model.

The Custom Autoscaler Controller: A Kubernetes controller that acts as the "actuator." It polls the Suggestion Server for the next action and interacts directly with the Kubernetes API to adjust the replica count of the target application.

3. To Implement a Realistic Training and Evaluation Environment:

To ensure the agent learns a robust policy, a key objective was to create a high-fidelity simulation environment. This includes:

Utilizing K6 to generate dynamic and realistic traffic patterns, mimicking everything from steady-state load to sudden, sharp spikes.

Integrating Chaos Mesh to inject controlled failures and stress, such as pod kills and resource contention, forces the agent to learn a policy that is not just efficient but also resilient.

By accomplishing these objectives, MAESTRO delivers a fully functional, intelligent autoscaling system. It is designed to benefit any organization relying on microservices, from large enterprises like banks and e-commerce platforms to smaller IT teams. By transforming application management from a reactive chore to a smart, preventative process, MAESTRO promises fewer outages, lower operational costs, and a more reliable user experience in the fast-moving world of cloud computing.

1.4 Gantt chart of project time plan

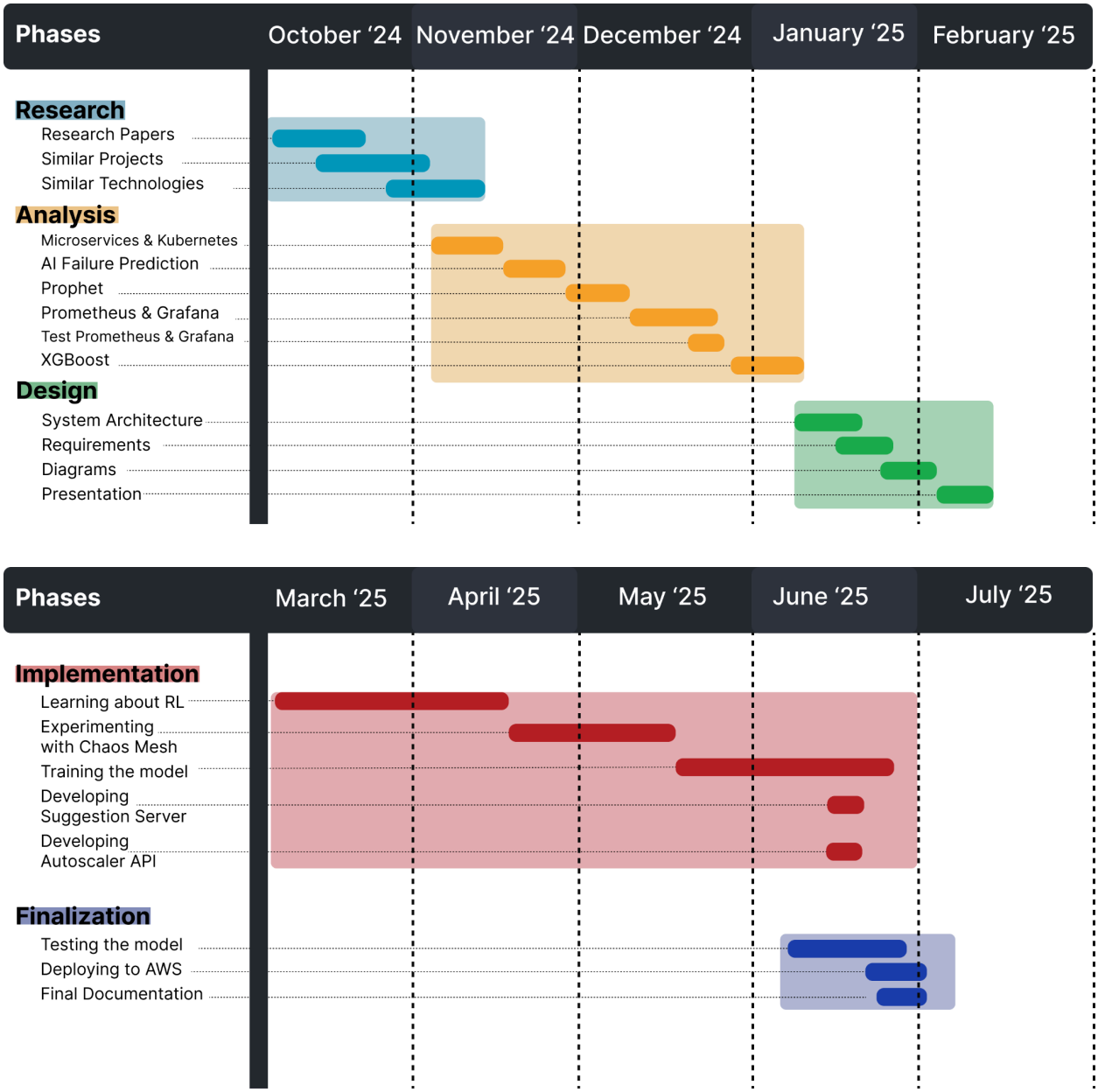

Figure 1. Gantt Chart

1.5 Project Development Methodology

To manage the complexity of developing MAESTRO, a system that integrates Kubernetes infrastructure, real-time control, and experimental machine learning, a hybrid development methodology was adopted. This approach combines the high-level, structured planning of a phased model with the iterative, flexible execution of Agile (Scrum) for the core development and training cycles. This hybrid model provided both a clear, long-term roadmap and the adaptability needed to address the research challenges inherent in building a novel RL-based system.

1. Phased Approach for High-Level Structure:

The overall project was segmented into four distinct, sequential phases, as outlined in the Gantt chart (Section 1.4). This provided a clear, macro-level structure:

Phase 1: Research & Design: A foundational phase focused on literature review and architectural design, ensuring a solid and well-understood plan before development.

Phase 2: Core Component Development: A dedicated implementation phase where all major software components were built according to the design specifications.

Phase 3: Training & Evaluation: This phase commenced once a functional system was in place and was focused exclusively on training the RL agent and conducting performance benchmarks.

Phase 4: Finalization & Documentation: The concluding phase, focused on refining the codebase and preparing the final project report.

This phased approach ensured that each major stage of the project had clear goals and well-defined entry and exit criteria.

2. Agile Scrum for Iterative Development and Experimentation:

Within the Development and Training phases, we employed an Agile methodology using the Scrum framework to handle uncertainty and rapid experimentation. This was crucial for both building the software and training the AI model.

Iterative Development: Work was divided into short sprints (typically 2 weeks), each delivering a usable piece of the system. For instance, an early sprint focused on a basic Kubernetes controller, while a later one focused on integrating the chaos engineering tools. This allowed for continuous integration and early detection of issues.

Experimentation-Driven Training: The training phase was, by nature, a series of experiments. Each training run was treated as a mini-sprint with a specific hypothesis, such as, "Will increasing the penalty for high latency lead to a more aggressive scale-up policy?" This "hypothesis-driven" loop is the essence of Agile applied to machine learning research, allowing for rapid tuning of the model's behavior based on empirical results.

The following Scrum practices were key to managing this process:

Backlogs: A Product Backlog was maintained with a prioritized list of all features and tasks. At the start of each cycle, a Sprint Backlog was created with the tasks selected for the current sprint.

Regular Feedback: Ongoing communication with the project supervisor served as the "customer" feedback loop, ensuring the project stayed aligned with its objectives. Sprint Reviews were held to demonstrate completed work.

Continuous Improvement: After each sprint, Sprint Retrospectives helped the team reflect on the process and identify areas for improvement in the next cycle.

This hybrid methodology provided the structure necessary to complete a complex project on schedule while offering the flexibility required to innovate and solve the non-deterministic challenges of training an intelligent agent.

1.6 Project Tools and Technologies

The successful implementation and evaluation of this project relied on a diverse stack of modern software tools and a capable hardware environment. The following sections detail the specific software (SW) and hardware (HW) components that were utilized.

Software (SW) Stack

1. Containerization & Orchestration:

- Docker: The foundational technology for containerizing all application components, including the RL Model Server, Suggestion Server, and Custom Autoscaler. Docker ensured environment consistency and portability.
- Kubernetes: The core container orchestration platform. It was used to manage the deployment, networking, and lifecycle of all system components and the target application.
- Minikube: Minikube is an open-source tool that enables the quick setup of a local Kubernetes cluster on macOS, Linux, and Windows systems. It creates a single-node cluster within a virtual machine or container on your local machine, providing a convenient environment for development, testing, and learning purposes.
- kubectl: The standard command-line interface for interacting with the Kubernetes cluster, used for deploying applications, inspecting logs, and managing resources.

2. Reinforcement Learning & Machine Learning:

- Python 3.8+: The primary programming language for developing the reinforcement learning agent and the associated server-side components.
- Stable Baselines3: A high-quality, open-source library built on PyTorch, providing a reliable and pre-built implementation of the

Proximal Policy Optimization (PPO) algorithm. This accelerated the RL development process significantly.

- PyTorch: The underlying deep learning framework used by Stable Baselines3 for building and training the neural network policy.
- Gymnasium (formerly OpenAI Gym): The standard toolkit for developing and comparing reinforcement learning environments. It was used to create the custom environment that simulated the autoscaling problem for the RL agent.
- Flask / FastAPI: A lightweight Python web framework used to build the RL Model Server, exposing the trained model's inference capabilities via a REST API.

3. Monitoring, Load Testing, & Chaos Engineering:

- Prometheus: A leading open-source monitoring and alerting system. It was deployed to scrape metrics (CPU, memory) from the Kubernetes nodes and pods, providing essential data for the agent's observation space.
- K6: A powerful, open-source load testing tool. K6 was used to write scripts that generated dynamic and configurable HTTP traffic against the target application, allowing us to simulate various load scenarios from steady-state to sudden spikes.
- Chaos Mesh: A cloud-native chaos engineering platform for Kubernetes. It was used to inject faults and system stress (e.g., high CPU load, memory consumption, pod failures) into the cluster during training and evaluation to test the autoscaler's resilience.

4. General Development:

- AWS: Cloud computing platform that provides a wide range of services and tools for hosting applications and managing resources. In the context of this scenario, AWS can be used to host a dummy client application and our failure prediction tool on a remote cluster.
- Visual Studio Code: The primary code editor used for development, with extensions for Docker, Kubernetes, Python, and Go.
- Git & GitHub: The version control system used to track all project code, configuration files, and documentation.

Hardware (HW) Environment

The project was developed and tested on a local machine with the following specifications to ensure sufficient resources for running a multi-node Kubernetes cluster and computationally intensive training tasks:

- CPU: Intel Core i7 / AMD Ryzen 7 (or equivalent) with at least 8 cores to handle the Kubernetes control plane, multiple worker nodes, and the ML training load.
- Memory (RAM): 32 GB of RAM. This was critical for allocating sufficient memory to the Docker Desktop VM, the Kind cluster nodes, and the memory-intensive training processes.
- Storage: 512 GB NVMe Solid State Drive (SSD). A high-speed SSD was essential for fast container image loading, quick pod startup times, and efficient read/write operations during model training.
- Operating System: Windows 11 with WSL 2 (Windows Subsystem for Linux) or a native Linux distribution (e.g., Ubuntu 20.04+). The use of a Linux environment (native or via WSL2) is standard for cloud-native development.

1.7 Report Organization

The remainder of this report is structured to provide a comprehensive overview of the project, from the theoretical background to the empirical results. The document is organized into the following chapters:

Chapter 2: Background and Related Work

This chapter provides the necessary theoretical foundation for understanding the project. It begins with an overview of core concepts in Kubernetes, including the architecture of controllers and the functionality of the standard Horizontal Pod Autoscaler (HPA). It then delves into the principles of Reinforcement Learning, with a specific focus on the Proximal Policy Optimization (PPO) algorithm. Finally, it presents a review of existing academic and industry research in the field of intelligent, learning-based autoscaling, contextualizing our work within the current state of the art.

Chapter 3: System Design and Architecture

This chapter details the "how" of the project, presenting the complete technical design of our proposed solution. It breaks down the architecture into its three main components: the Custom Autoscaler Controller, the Suggestion Server, and the RL Model Server. It describes the responsibilities of each component, the API contracts that connect them, and the flow of data and decisions through the system. This section also formally defines the RL environment, including the observation space, action space, and the logic of the reward function.

Chapter 4: Implementation and Training

This chapter transitions from design to execution, covering the

practical implementation details of the system. It discusses key code segments, the tools used for development (e.g., client-go, Stable Baselines3), and the process of containerizing each service with Docker. A significant portion of this chapter is dedicated to the model training methodology, explaining how the K6 traffic generator and Chaos Mesh were integrated into the training loop to produce a robust and resilient policy.

Chapter 5: Results and Evaluation

This chapter presents the empirical findings of the project. It showcases the performance of our trained RL autoscaler under various load scenarios. Through a series of graphs and quantitative metrics, we conduct a comparative analysis, benchmarking our solution against the default Kubernetes HPA. The evaluation focuses on key performance indicators such as application latency, the number of active pods (cost), and the stability of the system, providing clear evidence of the effectiveness of our approach.

Chapter 6: Conclusion and Future Work

The final chapter summarizes the project's achievements and contributions. It revisits the problem definition and discusses how our objectives were met. It concludes with a reflection on the key takeaways and limitations of the current work, and proposes several exciting directions for future research. These potential extensions include exploring more complex reward functions, applying the model to multi-service applications, and integrating different types of predictive forecasting.

Appendix: Installation and Usage Guide

The appendix provides a practical, step-by-step guide for setting up the project environment, deploying the system components, and running a demonstration of the intelligent autoscaler. This includes instructions for configuring the Kubernetes cluster, deploying the necessary YAML manifests, and initiating the load testing scripts.

Chapter 2: Related work

The challenge of intelligently and automatically managing resources for cloud applications is an active area of research in both academia and industry. Our work builds upon a foundation of existing knowledge in Kubernetes autoscaling, reinforcement learning for systems, and predictive resource management. This chapter reviews significant contributions in these areas, highlighting their approaches and positioning our project in context.

2.1 Kubernetes Native Autoscaling

The primary baseline for any custom autoscaling solution is the native tooling provided by Kubernetes. The Horizontal Pod Autoscaler (HPA) [1] is the standard mechanism for scaling workloads. It operates by periodically checking a specified metric (e.g., average CPU utilization across all pods) against a target value. If the current metric deviates from the target, the HPA's control loop calculates a new replica count and updates the deployment. While robust and widely used, its limitations are well-documented: it is purely reactive, acting only after a threshold has been breached, and it typically relies on a single, indirect metric (like CPU) which may not accurately reflect application performance.

Our work differs from these native solutions by replacing their reactive, threshold-based logic with a proactive, learning-based policy. Instead of a single metric, we use a multi-dimensional state vector to make more informed decisions, aiming to anticipate load rather than just react to it.

2.2 Reinforcement Learning for Kubernetes Orchestration

The application of Reinforcement Learning to the broader Kubernetes control plane is an active and promising area of research. RL agents have been proposed to optimize various facets of cluster management, from resource allocation to network routing.

A prime example of this is the thesis "Scheduling Kubernetes Tasks with Reinforcement Learning" by Sonia Matranga [3]. This work introduced a custom scheduler plugin for Kubernetes that utilized a Deep Q-Network (DQN) agent. The agent's objective was to learn an optimal pod placement policy by observing the state of cluster nodes. By training different models with distinct reward functions, the system could learn to optimize for various goals, such as:

- Energy Efficiency: Consolidating pods onto a minimal number of nodes.
- Load Balancing: Distributing pods evenly across all available nodes.
- Latency-Awareness: Placing pods on nodes with the lowest network latency to the user.

While this work provides valuable insight into applying RL within the Kubernetes control plane, its focus on pod placement (scheduling) is fundamentally different from our project's focus on replica-count management (autoscaling). The scheduler plugin addresses the "where should a new pod run?" question, whereas our project addresses the "how many pods should be running?" question. Nonetheless, it demonstrates the viability of integrating RL agents with core Kubernetes components to create more intelligent systems.

2.3 Reinforcement Learning for Cloud Resource Management

The application of Reinforcement Learning (RL) to systems problems, and specifically to cloud resource management, has gained significant traction.

A notable example is presented by Google in their work on a generic RL-based resource management framework [4]. They demonstrate the ability of RL agents to optimize resource allocation in complex, large-scale systems. Similarly, the work by Mao et al. on a deep learning-based framework for cluster scheduling, "Decima" [5], showcases how RL can learn sophisticated scheduling policies that outperform traditional heuristics by considering job characteristics and system state.

More directly related to autoscaling, the paper "Reinforcement Learning for Web Application Autoscaling" by D. G. R. de Jesus et al. [6] explores using Q-learning to manage a three-tier web application. They show that an RL agent can reduce costs and SLO violations compared to threshold-based policies. However, their approach uses a more traditional Q-learning algorithm, which can be less stable and sample-efficient than modern policy gradient methods like PPO.

Another relevant project, "FIRMA" [7], proposes a predictive autoscaling system that uses machine learning to forecast workload and proactively adjust resources. While effective, FIRMA focuses primarily on time-series forecasting to inform a heuristic-based scaling logic, rather than employing an end-to-end RL policy that learns the control logic itself.

Our project builds upon these ideas but makes several key distinctions. We employ Proximal Policy Optimization (PPO), a state-of-the-art algorithm known for its stability and performance in continuous control problems. Furthermore, our approach is distinguished by its holistic and resilient training methodology. We explicitly incorporate user-facing latency and requests per second into the agent's observation space and, crucially, integrate chaos engineering into the training loop. This novel step of training the agent under simulated failure conditions (e.g., pod kills, resource stress via Chaos Mesh) is designed to produce a policy that is not only efficient but also significantly more robust and production-ready than models trained in idealized environments.

2.4 Predictive and Proactive Autoscaling Systems

Beyond pure RL, there is a class of systems that focus on predictive autoscaling. For instance, "Compass" [8] is a system that uses time-series forecasting models (like ARIMA or LSTM) to predict future resource demand and scale the system ahead of time. These systems have proven effective at handling predictable, cyclical traffic patterns.

The primary difference in our approach is that we do not decouple the prediction from the control logic. In a forecasting-based system, one model predicts the load, and a separate, often heuristic-based, system decides how to act on that prediction. In our end-to-end RL system, the PPO agent learns both the predictive patterns and the optimal control policy simultaneously. It implicitly learns that a rising rps trend will likely lead to higher latency and learns to scale out as a single, unified policy, which can be more powerful in handling complex, non-linear system dynamics.

In summary, while our project stands on the shoulders of significant prior work, it contributes a novel approach by combining a state-of-the-art PPO agent with a comprehensive state representation and a unique, resilience-focused training methodology that incorporates chaos engineering, aiming to bridge the gap between academic research and a robust, practical autoscaling solution.

Chapter 3: System Analysis

3.1 Project Specification

The project's primary goal is to design, implement, and deploy an intelligent autoscaling controller for Kubernetes deployments. This controller leverages a trained Proximal Policy Optimization (PPO) model to dynamically adjust the number of pod replicas, optimizing for both application performance and resource cost.

3.1.1 Functional Requirements

FR1: State Observation: The system must be capable of collecting a set of real-time metrics that constitute the observation state for the RL model. This includes:

- Average CPU utilization across all pods of the target deployment.
- Average Memory utilization across all pods of the target deployment.
- The current number of replicas for the target deployment.
- End-to-end application latency (e.g., p95 latency).
- The number of requests per second (RPS) hitting the service.

FR2: Action Inference: The system must query the trained PPO model with the current observation state and receive a discrete action recommendation. The possible actions are:

- Scale Up: Increase the replica count.
- Scale Down: Decrease the replica count.
- No-Op: Maintain the current replica count.

FR3: Autonomous Scaling: Based on the inferred action, the system must automatically execute the scaling operation by interacting with the Kubernetes API server.

If the action is Scale Up, the controller must increment the spec.replicas field of the target deployment.

If the action is Scale Down, the controller must decrement the spec.replicas field of the target deployment.

FR4: Configurable Polling: The system must operate on a configurable control loop, periodically repeating the observation, inference, and action cycle. The interval of this loop (e.g., every 15 seconds) must be user-configurable.

FR5: System Deployment: All components of the system (Auto-scaler, Suggestion Server, RL Model Server) must be containerized and deployable on a Kubernetes cluster via standard YAML manifests.

3.1.2 Non-Functional Requirements

NFR1: Performance: The decision-making latency of the entire scaling loop (from metric collection to action execution) must be low, ideally completing within a few seconds, to enable timely responses to workload changes.

NFR2: Reliability & Availability: The autoscaling system must be highly available. The failure of a single component (e.g., the RL Model Server) should have a predictable fallback behavior (e.g., pausing scaling actions) and should not crash the entire control loop. The system must be designed to automatically recover from transient failures.

NFR3: Scalability: The autoscaling solution itself must be lightweight and scalable, capable of managing multiple target deployments within a cluster without introducing significant resource overhead.

NFR4: Resilience: The system must demonstrate resilience to adverse conditions. It should make sensible scaling decisions even during periods of network instability, pod failures, or resource stress, as simulated by tools like Chaos Mesh.

NFR5: Modularity & Maintainability: The architecture must be decoupled. The three main components (Auto-scaler, Suggestion Server, RL Model Server) must be independently deployable and updatable, facilitating easier maintenance and upgrades.

3.2 Use Case Diagram

Actors:

- System Administrator: The user who deploys, configures, and monitors the autoscaler.
- Kubernetes API Server: An external system that the controller interacts with.
- Monitoring System (Prometheus): An external system providing CPU/Memory/RPS metrics.
- Latency App: An external system providing the latency metric.

Use Cases:

- Configure Autoscaler: The administrator sets the target deployment, min/max replicas, and polling interval.
- Autonomously Scale Deployment: The core, automated use case performed by the system components.
- Monitor Scaling Events: The administrator observes the actions taken by the autoscaler.

Chapter 4: System Design

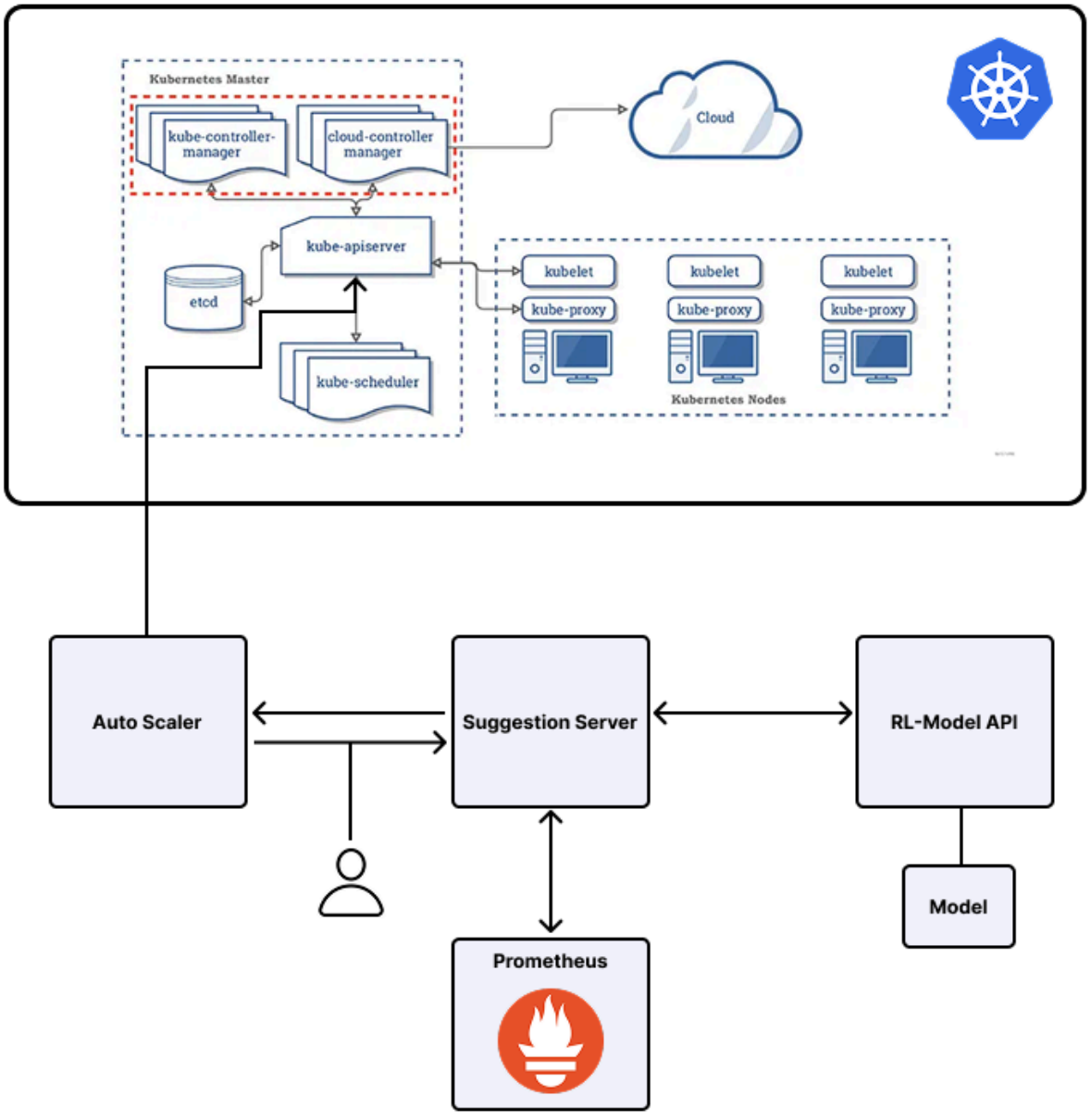

Figure 2. System Diagram

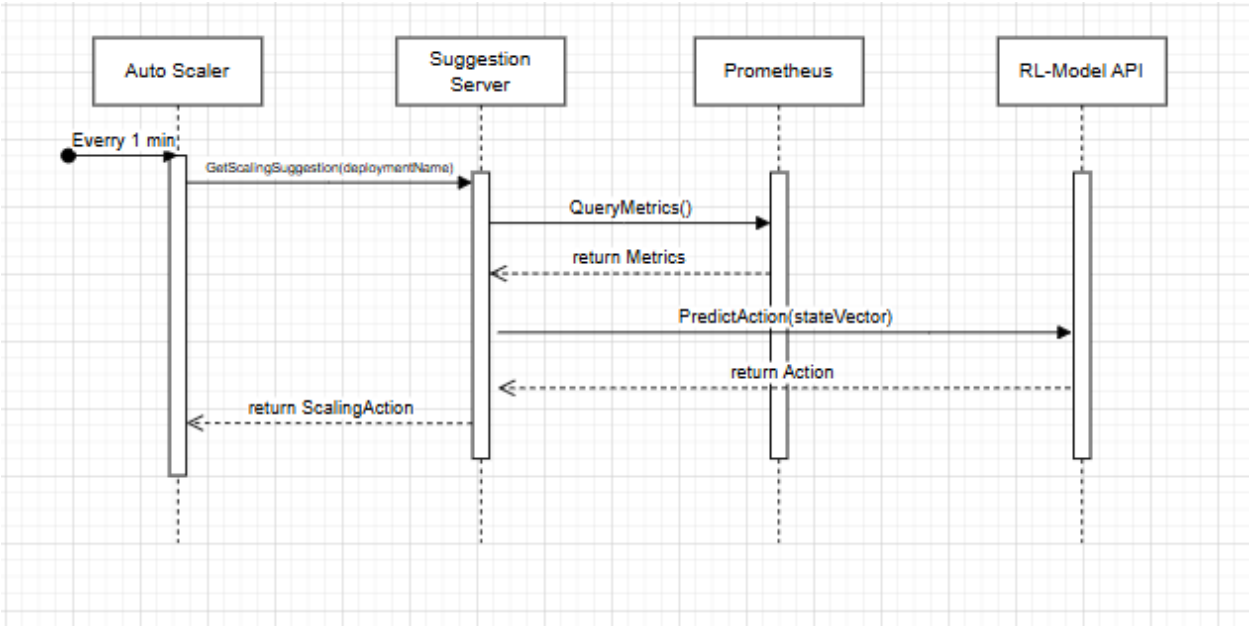

Figure 3. Sequence Diagram

4.1 Architecture

The architecture of the proposed RL-HPA system is composed of four key, decoupled components designed for modularity and scalability.

Custom Autoscaler: A Python-based Kubernetes controller responsible for acting upon scaling decisions.

Suggestion Server: A Python-based API server that serves as the inference endpoint for the RL model.

RL Agent & Environment: The core intelligence, consisting of the trained PPO model and the MicroserviceEnv where it learns.

Prometheus: The monitoring backbone, providing the real-time metrics necessary for the system's state representation.

4.2 Custom Autoscaler

The Custom Autoscaler is a dedicated controller running within the Kubernetes cluster. It periodically queries the Suggestion Server to retrieve the optimal replica count for each managed deployment. It then uses the Kubernetes API to apply this count, triggering the platform's native scaling mechanisms. This design separates the decision logic from the actuation, allowing the core RL model to be updated and retrained without altering the in-cluster controller.

4.3 RL Model and Environment

The core of the intelligent autoscaling system is a Reinforcement Learning agent trained using the Proximal Policy Optimization (PPO) algorithm, implemented with the Stable-Baselines3 library.

The agent's learning process takes place within a custom environment, MicroserviceEnv, which conforms to the gymnasium.Env interface. This environment directly interfaces with a live Kubernetes cluster, providing a high-fidelity training ground.

Action Space: The agent can choose from a discrete set of three actions:

- 0: Scale Down (decrease replica count by 1).
- 1: Do Nothing (maintain current replica count).
- 2: Scale Up (increase replica count by 1).

Observation Space: The state provided to the agent is a 5-dimensional vector composed of critical, real-time metrics, constructed by the StateBuilder class:

- CPU Usage (%): Sum of CPU usage across all pods.
- Normalized Memory: Memory usage normalized against the total memory capacity of all current replicas.
- Current Replicas: The current number of active pods.
- P95 Latency (ms): 95th percentile request latency.
- Requests Per Second (RPS): The current traffic load on the service.

Reward Function: The RewardCalculator class provides feedback to the agent.

```
r1 = (max_replicas - new_state[2]) / max_replicas
r2 = 0
terminated = False
latency = new_state[3]
latencySoftConstraint = float(annotations.get('latencySoftConstraint', -1))
latencyHardConstraint = float(annotations.get('latencyHardConstraint', -1))
if latencySoftConstraint != -1 and latencyHardConstraint != -1:
    if latency > latencyHardConstraint:
        r2 = -1
        terminated = True
    elif latency > latencySoftConstraint:
        r2 = 1.0 - (latency - latencySoftConstraint) / (latencyHardConstraint - latencySoftConstraint)
    else:
        r2 = 1
reward = 0.3 * r1 + 0.7 * r2
```

Figure 4. Reward Function

Chaos Engineering Integration:

To build resilience, the ChaosExperimentManager is invoked at each step to potentially inject a PodKill fault, forcing the agent to learn recovery strategies.

4.4 RL Model Training

The PPO agent is trained using the provided train.py script. The model architecture is a standard Multi-Layer Perceptron (MlpPolicy). The training process utilizes several callbacks for robust experimentation:

EvalCallback: Periodically evaluates the agent's performance on a separate environment and saves the best-performing model.

WandbCallback: Integrates with Weights & Biases to log all training metrics, hyperparameters, and model checkpoints for full observability and reproducibility.

PodTrackingCallback: A custom callback that records the agent's scaling decisions during training, allowing for post-hoc visualization of the learned policy's behavior.

4.5 Suggestion Server

The Suggestion Server is a lightweight Python (Flask) application. It exposes a single primary endpoint that the Custom Autoscaler calls:

- GET /suggest?deployment=<name>:

Receives a request for a scaling suggestion for a specific deployment.

Queries Prometheus for the latest CPU, memory, latency, and RPS metrics.

Constructs the 5-dimensional state vector using the StateBuilder.

Sends a request to RL-Model API with the 5-dimensional state vector.

Reserves the deployment action.

Returns a JSON object with the action to the Custom Autoscaler.

4.6 RL-Model API

The RL-Model API is a dedicated and streamlined Python microservice, built with Flask, that serves as the inference engine for the system. It encapsulates the trained PPO model, exposing its decision-making capability via a simple and robust API.

- POST /predict:

Receives a request containing the 5-dimensional state vector in a JSON payload.

Validates the incoming data to ensure all required metrics (cpu_usage, memory_usage, etc.) are present.

Passes the state vector to the pre-loaded PPO model's .predict() method.

Receives the deterministic action (0, 1, or 2) from the model.

Returns a JSON object containing the predicted action to the Suggestion Server.

Chapter 5: Implementation and Testing

Training the RL model

The training stage began with identifying the best model. Initially, we trained it using a simple reward function to evaluate its scalability and detect any potential conflicts. Once this baseline was established, we introduced stress using Chaos Mesh. However, this led to a conflict where stress injection interfered with pod scaling. After resolving the issue, the model was able to train successfully without further problems.

Next, we refined the reward function to account for both latency and the number of pods. To improve separation of concerns and enhance communication between APIs, we developed the autoscaler API and the suggestion server.

After 2,000 training iterations, the model exhibited a consistent upward trend in the mean reward graphs. Based on this progress, we extended training to 20,000 iterations and trained two versions of the model: one with stress and one without. Both models were exposed to generated traffic via random requests. The model trained without stress achieved a higher mean reward and demonstrated better overall performance particularly after the 16,000th iteration, where a notable increase in mean reward was observed.

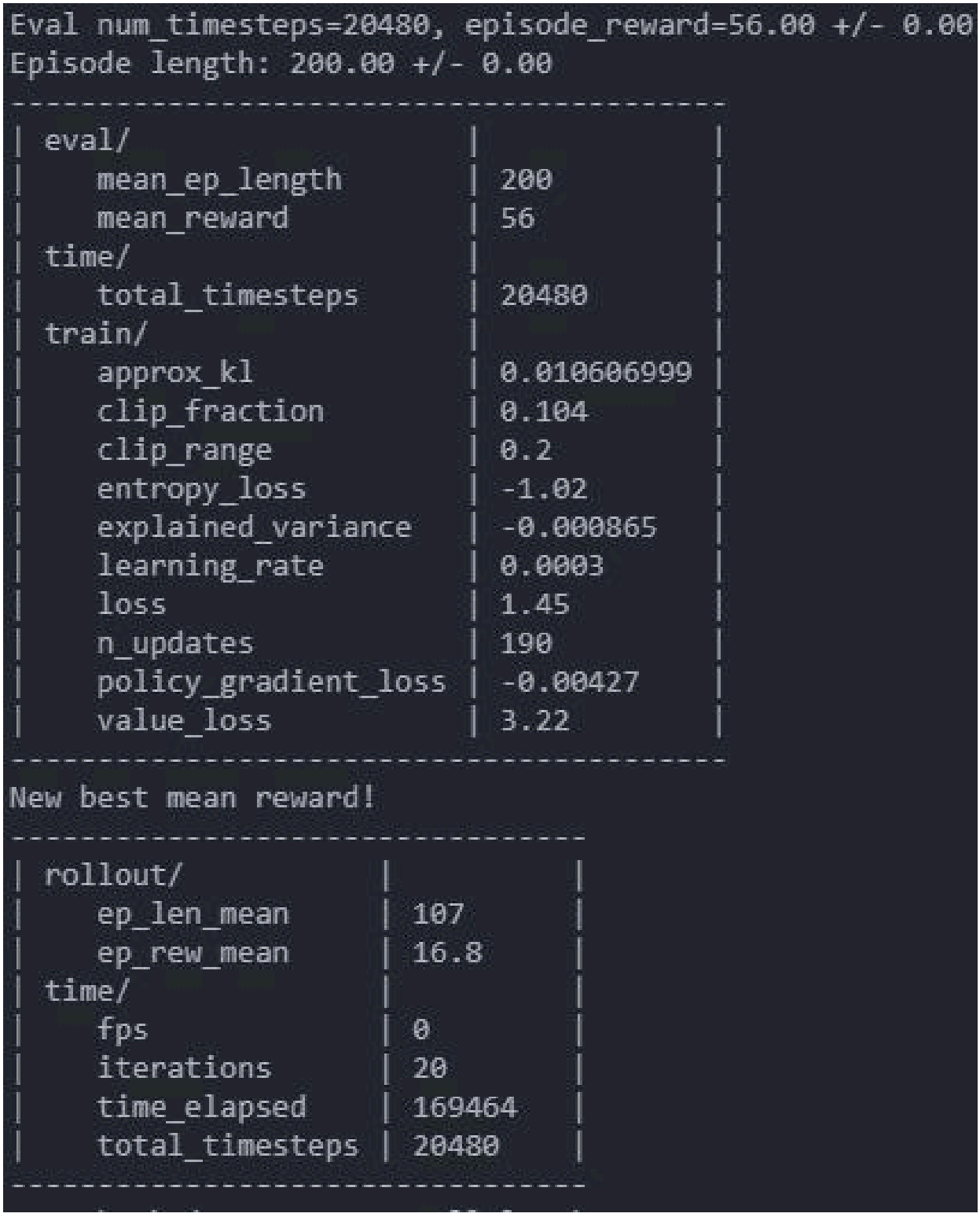

Figure 5. Model results after training

On the other hand, the model that was exposed to stress, suffered from a major decrease in performance and a decline in the mean reward graph, this was due to an issue in the training process, where we tried to simulate a situation where a pod is stressed highly enough that it would fail, this was done using a pod kill feature provided by Chaos Mesh, but it came at the cost of putting a great load on the training process, where the model would take a lot more time to train due to the very high stress needed to complete the pod kill process.

Furthermore, we gave a negative reward to the model when a pod would be killed, which we realized later on was a mistake, since the model had no way to prevent the pod kill process, and at the same time it would be rewarded a negative reward without any fault from its side, which resulted in a decrease in its overall performance and episode length, since the episode would end whenever a negative reward was given to the model.

Based on the results of this experiment, we decided to discard the entire pod kill experiment after reaching 7000 iterations without a noticeable improvement, and instead agreed to just train the model under normal stress scenarios, without very high stress conditions like this.

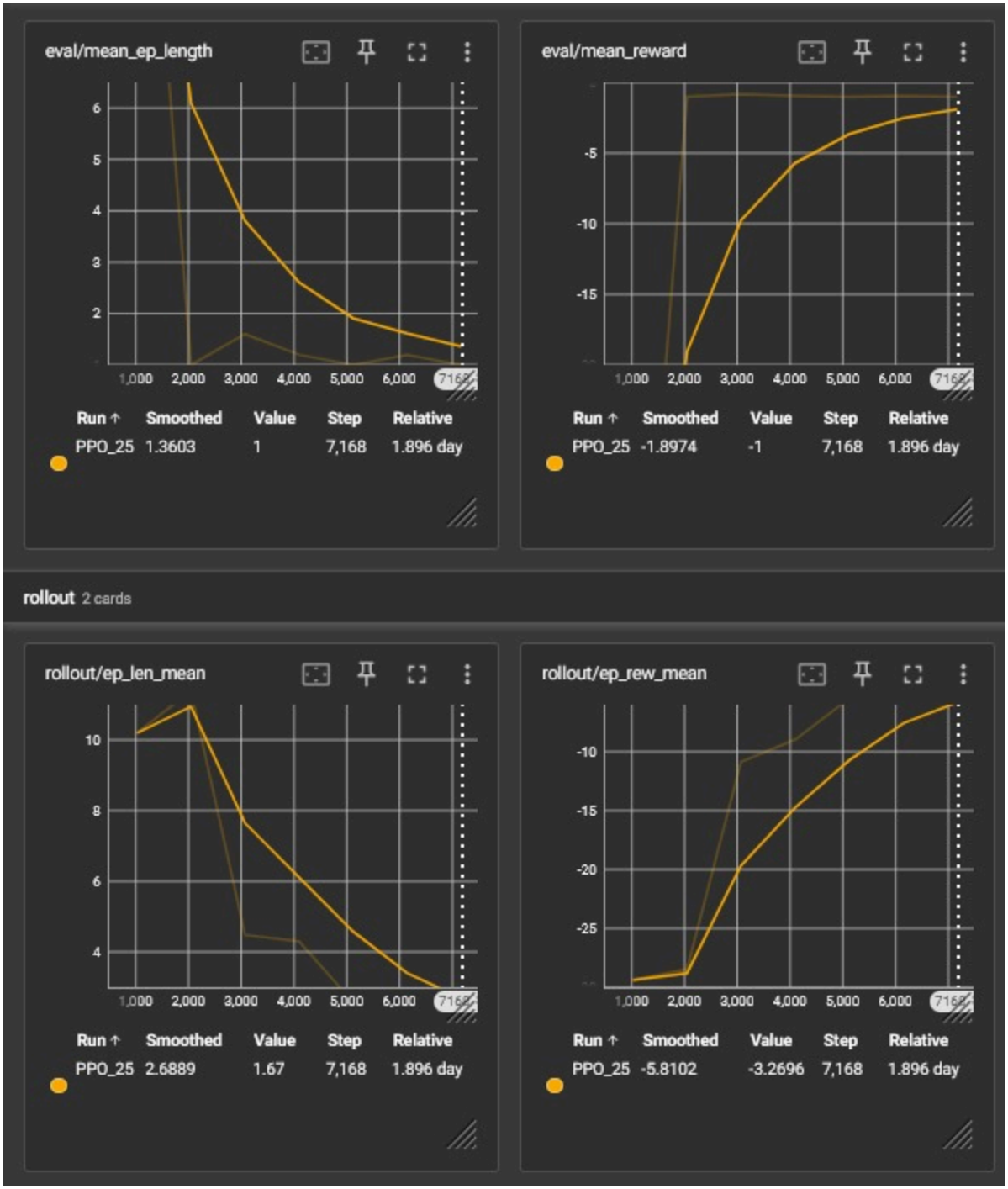

Figure 6. Graph of model with stress

Training/Evaluation Mean Reward

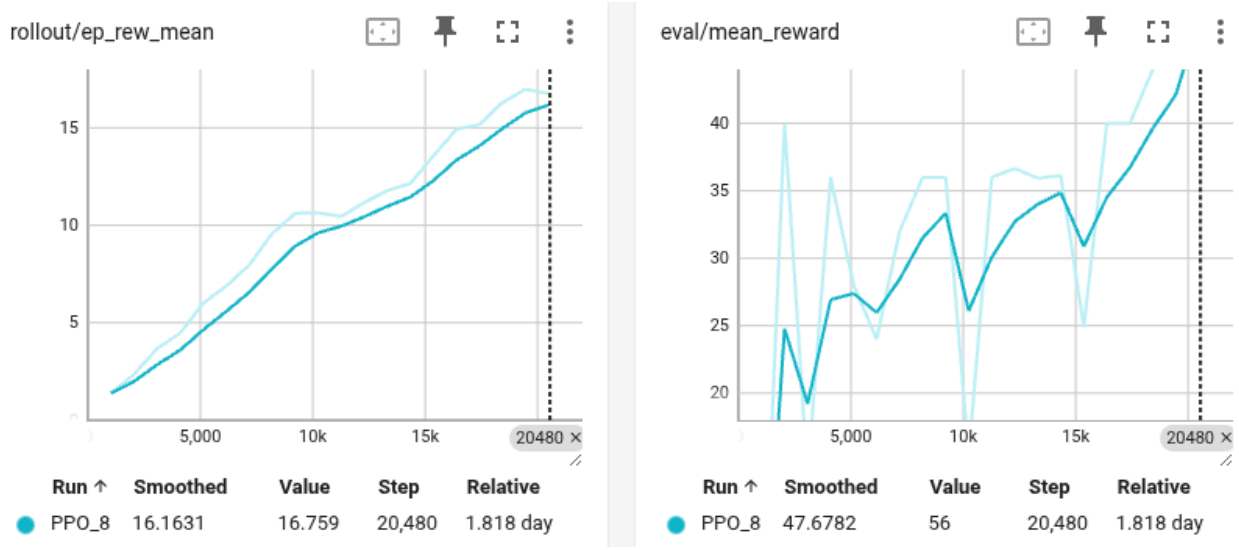

Figure 7. Mean Reward Graphs of model without stress

These graphs confirm the successful training of the PPO agent. The steadily increasing reward on the left (rollout/ep_rew_mean) shows that the agent consistently improved its strategy during the learning phase. The graph on the right (eval/mean_reward), while more volatile, confirms this upward trend during periodic evaluations, proving that the learned policy is effective and robust when exploration is turned off.

Overall, the graphs indicate that the agent has successfully learned to maximize its reward, achieving the desired balance between performance and resource efficiency, and is ready for deployment.

Training/Evaluation Mean Episode Length

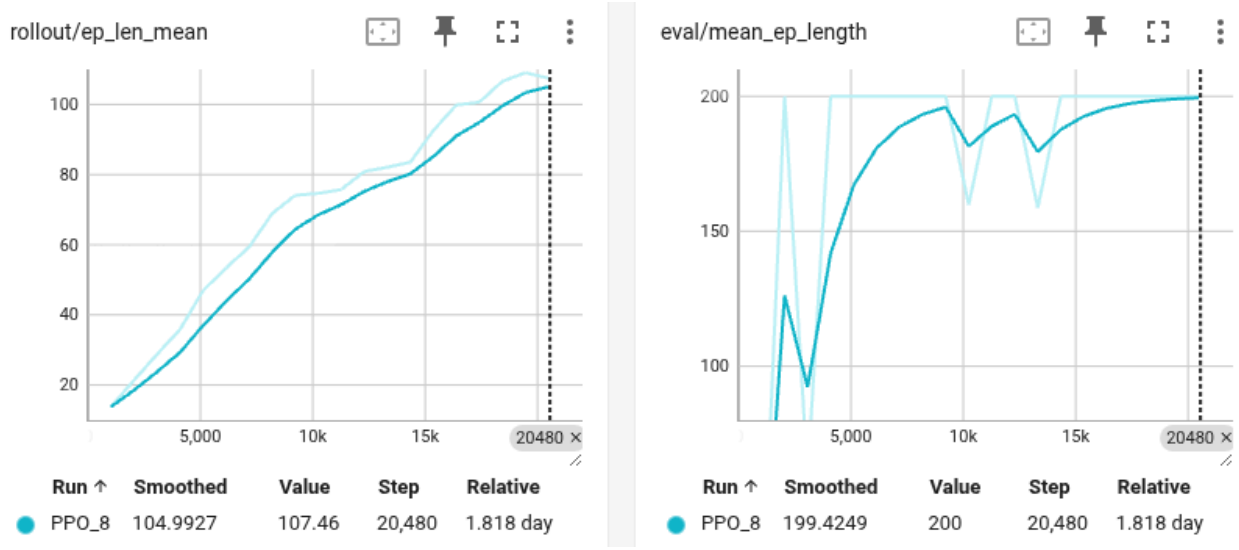

Figure 8. Mean Episode Graphs of model without stress

These graphs measure how long the agent can manage the system without causing a failure. The graph on the left (rollout/ep_len_mean) shows a clear, steady increase, indicating the agent is learning to avoid critical errors during training.

The graph on the right (eval/mean_ep_length) is even more compelling. It shows that the agent quickly learns to survive for the maximum possible duration (200 steps) during evaluation. This demonstrates that the final learned policy is highly stable and robust, successfully operating the system without violating the critical performance boundaries that would terminate an episode.

Project Poster

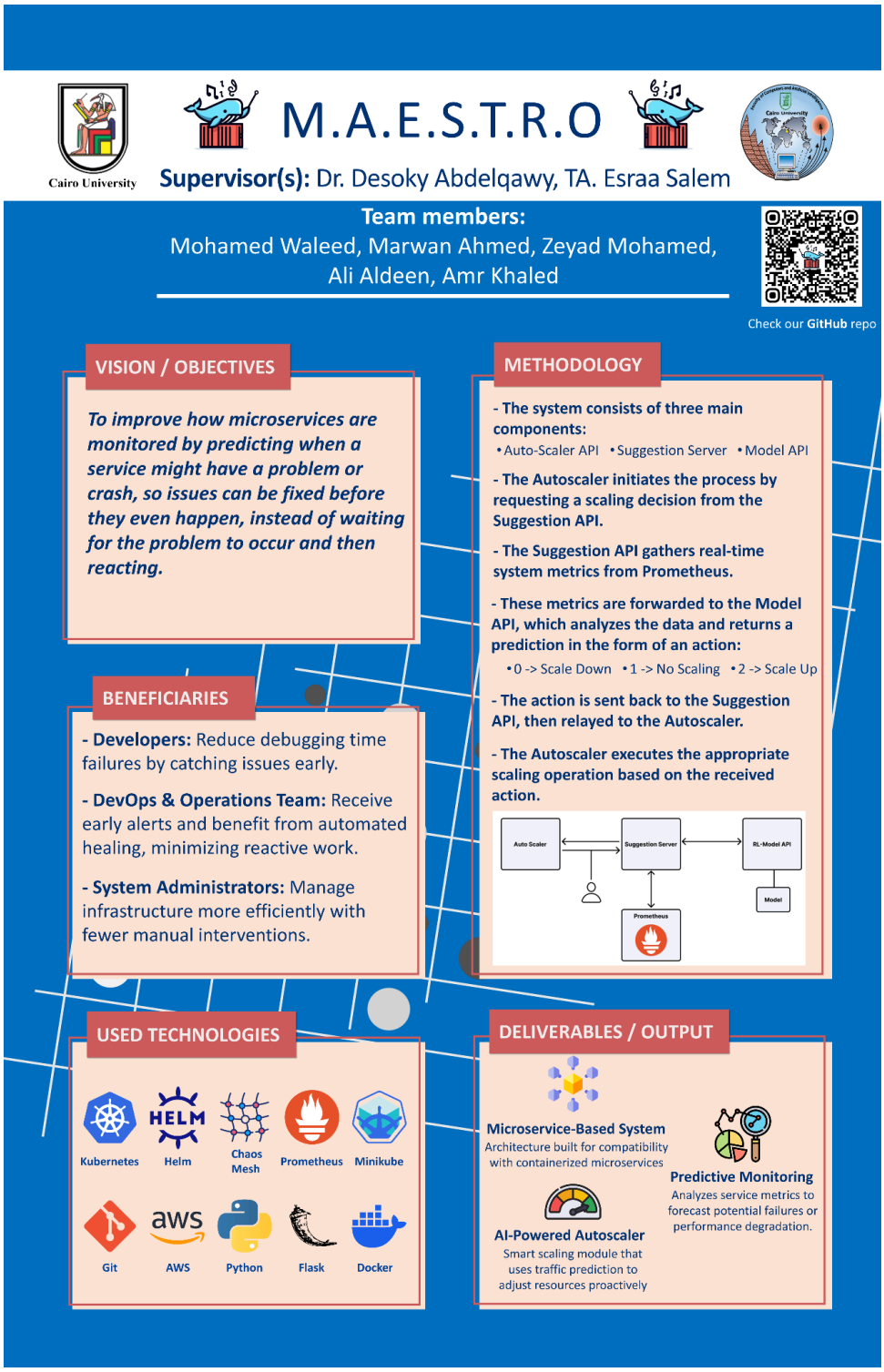

Figure 9. Poster

Appendix A: System Installation Guide

In this appendix, we demonstrate how to recreate the system implemented in this thesis locally and how to make modifications to implement future work.

A.1 Prerequisites

The system requires the installation of components to run a kind cluster, including:

Software Requirements

• **Docker**: Install Docker to enable containerization and management of containers. Instructions for installing Docker can be found on the official Docker documentation.

• **Minikube**: Minikube is an open-source tool that enables the quick setup of a local Kubernetes cluster on macOS, Linux, and Windows systems. It creates a single-node cluster within a virtual machine or container on your local machine, providing a convenient environment for development, testing, and learning purposes.

• **kubectl**: Install kubectl, the Kubernetes command-line tool, to interact with the Kubernetes cluster. Follow the kubectl installation guide for instructions.

• **Python**: Ensure Python is installed on your system. You can download it from the official Python website.

Hardware Requirements

Ensure your system meets the following hardware requirements:

- **CPU**: At least 4 CPU cores.
- **Memory**: At least 8 GB of RAM.
- **Disk Space**: At least 20 GB of free disk space.

A.2 Installation

The installation process is divided into three main stages: deploying the monitoring stack, deploying the target application, and deploying the MAESTRO autoscaling components.

Step 1: Clone the Project Repository

First, clone the repository containing all the necessary Kubernetes YAML files.

```
> git clone git@github.com:Alialdin99/Microservice-failure-prediction-and-self-healing.git
> cd Microservice-failure-prediction-and-self-healing
```

Step 2: Deploy the Monitoring Stack (Prometheus)

MAESTRO relies on Prometheus to collect the metrics that form the agent's observation state. We will use the official kube-prometheus-stack Helm chart, which provides a comprehensive monitoring solution.

```
  # 1. Add the Prometheus community Helm repository
> helm repo add prometheus-community https://prometheus-community.github.io/helm-charts
> helm repo update

  # 2. Install the kube-prometheus-stack
  # This chart deploys Prometheus, Grafana, and the necessary metric exporters.
> helm install prometheus prometheus-community/kube-prometheus-stack --namespace monitoring
  --create-namespace
```

Step 3: Deploy the Target Application & Services

This step deploys the sample application that MAESTRO will autoscale. This deployment includes the application itself, a sidecar for latency metrics, and a Kubernetes Service to expose it.

```
# Apply the manifest file for the target application
> kubectl apply -f manifests/target-app/deployment.yaml
```

This will create a Deployment, Service, and any other necessary resources for the application you want to scale.

Step 4: Deploy the MAESTRO Autoscaling System

Now, deploy the three core components of the MAESTRO system. The manifests are structured to create the Deployments, Services, and necessary ServiceAccounts with RBAC permissions.

```
# 1. Deploy the RL Model Server
# This pod hosts the trained PPO model.
> kubectl apply -f manifests/maestro/rl-model-server.yaml
## 2. Deploy the Suggestion Server
# This service acts as the intermediary.
> kubectl apply -f manifests/maestro/suggestion-server.yaml
 3. Deploy the Custom Autoscaler Controller
# This is the agent that interacts with the K8s API.
# It includes the necessary Role and RoleBinding for scaling deployments.
> kubectl apply -f manifests/maestro/autoscaler-controller.yaml
```

After applying these manifests, all components of the MAESTRO system are deployed.

A.3 Verifying the Installation

To ensure that all components are running correctly, you can check the status of the pods in the default (or relevant) namespace.

```
# 1. Deploy the RL Model Server
# This pod hosts the trained PPO model.
> kubectl get pods

# Expected output should show all pods in the 'Running' state:
# NAME                                        READY   STATUS    RESTARTS   AGE
# target-app-deployment-xxxxxxxxxx-xxxxx      2/2     Running   0          5m
# rl-model-server-deployment-xxxxxxxx-xxxxx   1/1     Running   0          2m
# suggestion-server-deployment-xxxxxxx-xxxxx  1/1     Running   0          2m
# autoscaler-controller-deployment-xxxxxx-xxxxx 1/1   Running   0          1m
```

You can also view the logs of the autoscaler controller to see it making decisions.

```
# First, get the name of the autoscaler pod
> kubectl get pods | grep autoscaler-controller

# Then, view its logs
> kubectl logs -f <autoscaler-controller-pod-name>
```

A.4 Generating Traffic (Optional)

To see the autoscaler in action, you need to generate load on the target application. The repository includes a manifest to deploy a k6 traffic generator pod.

```
# Apply the k6 traffic generator job
> kubectl apply -f manifests/tools/k6-traffic-generator.yaml
```

References

[1] Schulman, J., et al. "Proximal Policy Optimization Algorithms." arXiv preprint arXiv:1707.06347. 2017. URL: https://arxiv.org/abs/1707.06347 (cit. on pp. X, Y).

[2] Sutton, R. S., and Barto, A. G. Reinforcement Learning: An Introduction. MIT Press, 2018. (cit. on p. X).

[3] Kubernetes Authors. "Horizontal Pod Autoscaling." Kubernetes Documentation. Accessed: [Date], 2024. URL: https://kubernetes.io/docs/tasks/run-application/horizontal-pod-autoscale/ (cit. on p. X).

[4] Mao, H., et al. "Resource Management with Deep Reinforcement Learning." Proceedings of the 15th ACM Workshop on Hot Topics in Networks. 2016. URL: https://dl.acm.org/doi/10.1145/2999572.2999582 (cit. on p. X).

[5] The CleanRL Authors. "CleanRL: High-quality Single-file Implementations of Deep Reinforcement Learning Algorithms." GitHub Repository. 2023. URL: https://github.com/vwxyzjn/cleanrl (cit. on pp. X, Y).

[6] The Gymnasium Development Team. "Gymnasium: An API for Reinforcement Learning." GitHub Repository. 2024. URL: https://gymnasium.farama.org/ (cit. on p. X).

[7] Prometheus Authors. "Prometheus: Monitoring system & time series database." Prometheus Documentation. Accessed: [Date], 2024. URL: https://prometheus.io/docs/introduction/overview/ (cit. on pp. X, Y).

[8] Chaos Mesh Authors. "Chaos Mesh: A Chaos Engineering Platform for Kubernetes." Chaos Mesh Documentation. Accessed: [Date], 2024. URL: https://chaos-mesh.org/docs/ (cit. on p. X).

[9] K6 Authors. "k6: An open-source load testing tool for engineering teams." K6 Documentation. Accessed: [Date], 2024. URL: https://k6.io/docs/ (cit. on p. X).

[10] Grinberg, M. Flask Web Development: Developing Web Applications with Python. O'Reilly Media, 2018. (cit. on p. X).

[11] Kubernetes Authors. "Kubernetes Documentation." Accessed: [Date], 2024. URL: https://kubernetes.io/docs/home/ (cit. on p. X).

[12] Amazon Web Services, Inc. "Elastic Kubernetes Service (EKS)." AWS Documentation. Accessed: [Date], 2024. URL: https://aws.amazon.com/eks/ (cit. on p. X).

[13] Roy, G., et al. "FIRMA: An Intelligent Fine-grained Resource Management for Serverless Computing." Proceedings of the ACM/IFIP/USENIX Middleware Conference. 2020. URL: https://dl.acm.org/doi/10.1145/3423211.3425680 (cit. on p. X).

[14] Al-Dhuraibi, Y., et al. "Compass: A Proactive Autoscaling Framework for Cloud Services." IEEE Transactions on Cloud Computing. 2021. URL: https://ieeexplore.ieee.org/document/9355419 (cit. on p. X).