National College of Ireland

Revolutionizing Cloud Management with AI-Powered Kubernetes Autoscaling Solutions

MSc Research Project
MSc Cloud Computing

Chethan Mundigehalla Prabhakar
Student ID: x23297395

School of Computing
National College of Ireland

Supervisor: Rashid Mijumbi

National College of Ireland

MSc Project Submission Sheet

School of Computing

Student Name:	Chethan Mundigehalla Prabhakar
Student ID:	X23297395
Programme:	MSc Cloud Computing **Year:** 2025
Module:	Research Project
Supervisor:	Rashid Mijumbi
Submission Due Date:	15-09-2025
Project Title:	Revolutionizing Cloud Management with AI-Powered Kubernetes Autoscaling Solutions
Word Count:	**9216 Page Count 24**

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: Chethan Mundigehalla Prabhakar

Date: 15-09-2025

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):	

Revolutionizing Cloud Management with AI-Powered Kubernetes Autoscaling Solutions

Chethan Mundigehalla Prabhakar
x23297395

Abstract

The proliferation of microservices and containerization, orchestrated predominantly by Kubernetes, has introduced significant challenges in cloud resource management. Traditional autoscaling mechanisms, such as the Horizontal Pod Autoscaler (HPA), are inherently reactive, adjusting resources only after performance metrics have already breached predefined thresholds. This reactive nature often causes periods of inefficient application performance or unnecessary over-provisioning of resources leading to buggy user experiences and excess operational costs. This work addresses the inefficiencies of the current autoscaling practices by designing, implementing, and evaluation a proactive, AI-based autoscaling solution for Kubernetes. The architecture we designed for this project implements a hybrid AI approach, using Machine Learning (ML) for workload forecasting, and Reinforcement Learning (RL) for rapid, intelligent, forward-looking scaling decisions. We designed a custom Kubernetes controller that encapsulates the AI pipeline using a Custom Resource Definition (CRD). An additional backoff controller was also included to ensure production-grade reliability - a large challenge when relying on distributed, AI systems - the backoff controller defaults back to standard HPA when there is low confidence in the AI model or instability in the system. Lastly an A/B testing controller automates the safe deployment and validation and promotion of new AI models. The evaluation on a Google Kubernetes Engine (GKE) cluster, demonstrated that the system was able to continually anticipate workload fluctuations, make smart scaling decisions to retain application performance and resource efficiency, and showed robust resilience. Overall the results confirm that a unified, AI-driven, approach can meaningfully push the state of cloud management further by fundamentally changing autoscaling from a reactive strategy, to a proactive optimization engine.

1 Introduction

The patronage cloud computing has fostered within the digital transformation age seems to be at an unprecedented speed regarding its mass adoption as an enabler of scalability, flexibility, and efficiency (Patel, 2023). In effect, organizations have been enabled to build and deploy highly complex applications globally. One of the greatest changes created through this transformation is in containerization and microservices architecture, with Kubernetes emerging as a de-facto standard for container orchestration (Kumar, Kaur, and Rana, 2025). Kubernetes abstracts underlying infrastructure in such a way that focus can be on application logic while the platform takes care of things like deployment, scaling, and operational tasks.

Autoscaling, the automatic allocation of computational resources in accordance with an application's changing workload, is one of the important aspects of managing applications in such a dynamic environment. The most well-known mechanism for this in Kubernetes is Horizontal Pod Autoscaler (HPA), which lets users scale up or down based on application instance (pod) replication depending on underlying metric observations for CPU or memory

usage. The HPA works fine for slow traffic changes, but its limitation is that it, by design, can react only after a metric traverses threshold; hence, it can cause performance degradation or even violations of service-level objectives (SLO) before the system responds (Sharma, 2025). Conversely, low thresholds to avoid practical issues can result in aggressive scaling and significant over-provisioning that raise operational costs.

An inherent inefficiency that reactive scaling can present is more than enough reason to compel organizations to adopt a more intelligent and proactive approach. The inclusion of Artificial Intelligence (AI) and Machine Learning (ML) within the management of cloud infrastructures has proven promising in terms of counteracting this scenario (Syed et al., 2025). With knowledge of historical data and learned workload patterns, AI models can then predict future resource needs so that the system scales in advance of peak demands before performance dips occur. This type of intelligent system development is investigated in this research. In fact, such a research gap is central to the study: the disunion caused by the traditional Kubernetes autoscaling's static and reactive characteristics and the dynamic and unpredictable characteristics of the workloads of modern applications.

The project investigates the following research question: **how can a building and implementation of a hybrid AI model, which comprises Machine Learning-based forecasting and Reinforcement Learning-based decision-making, be achieved to realize a proactive, resilient, and efficient autoscaling solution for workloads hosted on Kubernetes?**

To answer this question, the following research objectives were established:

1. To design and implement a custom Kubernetes controller using a Custom Resource Definition (CRD) to manage the lifecycle of an AI-powered autoscaler.
2. To develop a Machine Learning module capable of forecasting key application metrics (e.g., response time, CPU usage) based on real-time and historical data.
3. To implement a Reinforcement Learning module that consumes the ML forecasts to make optimal, proactive scaling decisions (scale up, scale down, or no change).
4. To architect and build a robust fallback system that ensures high availability by reverting to a traditional HPA if the AI system's performance or confidence degrades.
5. To evaluate the effectiveness of the complete system in a realistic cloud environment under dynamic load, assessing its proactivity, scaling accuracy, and resilience.

The primary contribution of this work is the creation and validation of a complete, end-to-end AI-powered autoscaling framework. Unlike studies that focus on a single aspect of AI-based scaling, this project integrates ML forecasting, RL decision-making, a custom Kubernetes operator, a resilient fallback mechanism, a comprehensive observability stack, and an automated A/B testing controller for safe model deployment. The result is a practical, production-ready blueprint for revolutionizing cloud resource management. Figure 1 provides a high-level overview of the system architecture.

Kubernetes has become the de-facto container orchestration platform, with the Horizontal Pod Autoscaler (HPA) widely adopted for scaling applications (Kubernetes, 2025). However, its reactive nature causes Service Level Agreement (SLA) violations under rapid workload fluctuations (Mishra et al., 2020). Recent studies emphasize the necessity of predictive and AI-driven scaling to minimize cost and ensure resilience (Syed et al., 2025; Vadisetty et al., 2024). This work contributes novelty by integrating online forecasting, Reinforcement Learning (RL)

decision-making, fallback mechanisms, and automated A/B testing into a unified Kubernetes controller, moving autoscaling from reactive to proactive.

1.1 Background of The Research

Linear Regression, a fundamental statistical method, models the relationship between dependent and independent variables by fitting a weighted linear function; in this project, an **online linear regression** approach was used, enabling incremental updates with each new data point, making it lightweight, adaptive to workload changes, and suitable for real-time Kubernetes control loops. Complementing this, **Q-learning**, a model-free Reinforcement Learning algorithm, was employed to optimize scaling policies by learning the long-term reward of state–action pairs using the update rule $[Q(s,a) \leftarrow Q(s,a) + \alpha\backslash Big\left[r + \gamma \max_{a'} Q\,(s',a') - Q(s,a)\backslash Big\right]$, where α is the learning rate, γ the discount factor, and rrr the reward; this allows the autoscaler to make proactive decisions such as scaling up, scaling down, or holding steady. To ensure safe model evolution, the system incorporated an **A/B Test Controller**, which enables canary deployments by running new AI models in parallel with the existing one under shadow traffic, comparing metrics like prediction accuracy, latency, and confidence, and only promoting the new model if it achieves a higher composite score. Together, these three components provide the predictive capability, intelligent decision-making, and safe continuous improvement necessary for a resilient AI-powered autoscaling framework.

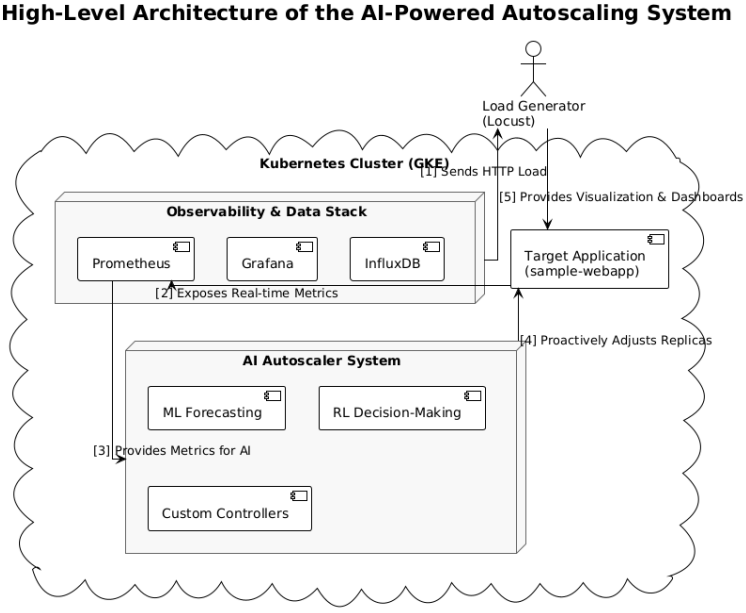

Figure 1: High-Level Architecture of the AI-Powered Autoscaling System

This report is structured as follows. Section 2 reviews related work in the fields of Kubernetes autoscaling and AI-driven cloud management. Section 3 outlines the research methodology employed in this project. Section 4 details the design and architecture of the proposed solution. Section 5 describes the implementation of the system on Google Kubernetes Engine. Section 6 presents a thorough evaluation of the system through a series of experiments. Finally, Section 7 concludes the report, summarizes the key findings, discusses limitations, and suggests avenues for future work.

2 Related Work

Researching and developing mechanisms for efficient resource management in cloud environments is a highly established field of research. This section will present a review of the literature, examining first traditional autoscaling methods in Kubernetes, followed by consideration of the use of AI, before finally highlighting the gaps within the existing research that this research aims to fill.

2.1 Traditional and Threshold-Based Autoscaling

The Horizontal Pod Autoscaler is the basis of Kubernetes autoscaling. The HPA tracks the resource usage metrics of a set of pods, usually some combination of CPU and memory, against a configurable user-defined target. If the average utilization exceeds that target, it scales out by adding more pod replicas. Conversely, if the utilization is way down below the target, it scales back and reduces replicas (Kubernetes, 2025). This model is simple to configure and understand, thus is widely used.

However, much is known about its limitations. The main complaint about these issues is with the HPA being reactive. It initiates scaling only when the system itself is under pressure, allowing time for performance degradation of the application. The lag thus caused can be significantly high for the HPA to be effective in some operational environments, as demonstrated by Mishra et al. (2020). The degradation in SLA could be worse if the lag is high and new container images have to be pulled before initializing the pods in a distributed environment. To avoid such scenarios, administrators are often forced to over-provision resources, going directly against the promise of paying only for what is used in a cloud environment.

Furthermore, the HPA heavily relies on simple and lagging indicators such as CPU utilization. An application whose performance is I/O bound or where CPU is one among several processors in a complex request pipeline may not be able to draw a linearly proportional relationship between CPU usage and actual performance. The next progressed kind of HPA is the Custom Metrics Autoscaler, where one can scale on application-specific metrics such as requests per second or queue length exposed via Prometheus. This could give a more direct scaling signal, but the logic remains reactive and threshold-based. The KEDA project takes this further to allow scaling based on events from multiple sources like Kafka queues or RabbitMQ, probably one of the few options remaining in triggering an event that has already occurred.

2.2 AI-Driven Cloud Infrastructure Management

The shortcomings of traditional approaches have generated a lot of interest in research into the application of innovative Artificial Intelligence (AI) techniques for managing cloud infrastructure, as AI could help move from reactive to predictive and proactive control (Vadisetty et al., 2024). That is the shift from reactive to predictive to proactive control will use historical data to learn patterns in order to make intelligent decisions about future resource needs.

Syed et al. (2025) provide an extensive survey of the topic titled "Artificial Intelligence as a Service (AIaaS)," detailing how AI models are being encapsulated as services to help with managing the complexities of cloud, fog, and edge environments. They note the particular trend of embedding intelligence in infrastructure management control planes directly. Sharma (2025)

also discusses the advancement of AI-driven cloud infrastructure with a focus on Kubernetes and server-less computing, where it is stated that AI will be needed to manage the complexities and varying scales of modern distributed systems (e.g., autoscaling, anomaly detection, resource scheduling). The body of work in this report directly maps to this goal of implementing AI as part of the Kubernetes control loop.

2.3 Machine Learning for Workload Forecasting

An integral part of any anticipatory autoscaler is the ability to predict future workloads. This has been the topic of a great deal of study. Time-series forecasting models are often used in this area. For instance, researchers have applied classical statistical models like ARIMA (Autoregressive Integrated Moving Average) and more advanced deep learning models like Long Short-Term Memory (LSTM) networks to predict future CPU utilization or request rates (Gari et al., 2021). LSTMs, in particular, are well-suited for capturing complex, non-linear patterns and long-term dependencies in workload data, making them a popular choice.

However, deploying and maintaining complex deep learning models can be resource-intensive and requires significant training data and time. An alternative approach, utilized in this research, is online learning. Online learning models, such as those from the River library (Montiel et al., 2021), learn incrementally from a stream of data. This makes them lightweight, highly adaptive to changing patterns (concept drift), and suitable for real-time environments where retraining large models frequently is impractical. The ability to detect concept drift, where the statistical properties of the workload change over time, is a key advantage, as it allows the system to recognize when its predictions are becoming unreliable.

2.4 Reinforcement Learning for Policy Optimization

Forecasting can tell "what" (the expected future load), but cannot tells "how" (the method of scaling). This is where Reinforcement Learning (RL) is very useful. RL is a form of machine learning where an agent learns to make sequential decisions in an environment to maximize a total reward. In the context of autoscaling, we can define the RL agent state by the current replicas, CPU usage, and memory usage, while the possible actions are scale_up, scale_down, and no_change. The reward function is defined in a way that adds positive value for what we want to achieve, e.g. low response time, low costs (Kaur and Aron, 2021). Q-learning, a popular model-free RL algorithm, can be used for this purpose, which learns a policy or Q-table, that defines the expected utility of taking a certain action in a certain state. Q-learning has been successfully applied to autoscaling virtual machines and containers in a several studies. For example, Shi et al. (2020) demonstrated a RL based autoscaler outperformed the traditional threshold based RL in both performance and cost. The outcomes of this study relies upon this concept by attaching feasible RL decision module to ml forecasting module; as a result,

2.5 Hybrid and Resilient Architectures

An AI-only system can be powerful, but could also have its own risks. AI models can be wrong – models are more prone to being wrong in scenarios that represent black swan events or significant concept drift. So a production-grade system must be robust. So the idea of a fallback system to a simple and predictable system when the AI fails is critical in ensuring reliability (Vadisetty et al., 2024). In this research, we implemented a fallback mechanism that monitors

the AI model confidence and health. When the model loses confidence, it swaps to a standard Kubernetes HPA deploy. We have created a hybrid system that leverages the intelligence of an AI system, while also retaining the reliability of the historic paradigms.

In addition, one challenge when work with AI models in production is effectively updating an AI model in production. New models can perform worse than the existing models. The A/B testing framework in this research has set out to address this problem with a canary deployment method. In the evaluation phase, we would deploy a new model while running slightly less than a percentage of live traffic through the new deployment while we run the existing deployment at the same instant. Its performance is objectively measured, and it is only promoted if it proves to be superior. This practice, common in software engineering, is adapted here for the safe lifecycle management of AI models in the infrastructure control plane.

In summary, the literature shows a clear progression from simple reactive autoscalers to more sophisticated AI-driven solutions. Research has explored ML for forecasting and RL for decision-making as separate components. However, there is a gap in the literature regarding a fully integrated, end-to-end system that combines these elements into a proactive control loop, complete with robust resilience mechanisms like a fallback controller and safe deployment strategies like A/B testing. This project directly addresses this gap by designing, implementing, and evaluating such a comprehensive system.

Table 1: Summaries from literature review

Author(s), Year	**Focus Area**	**Approach/Method**	**Key Findings**	**Limitations/Gaps**
Kubernetes, 2025	Kubernetes HPA mechanism	Threshold-based reactive scaling	Simple and widely adopted; scales based on CPU/memory usage	Reactive, can cause SLA breaches; unsuitable for complex workloads
Mishra et al., 2020	Evaluation of HPA	Performance testing under varying loads	Demonstrated lag in scaling under sudden load changes	Encourages over-provisioning to avoid lag
Vadisetty et al., 2024	AI-driven cloud resilience	AI-based proactive scaling & fallback	Improved reliability through predictive control and failover	Limited focus on integrated ML+RL
Syed et al., 2025	AIaaS for cloud/fog/edge	Survey of AI embedding in infrastructure	Trends towards predictive/proactive autoscaling	Lacks deep integration with Kubernetes CRDs

Gari et al., 2021	Workload forecasting	LSTM for CPU/request rate prediction	Captures non-linear temporal patterns effectively	High resource/training requirements
Montiel et al., 2021	Online ML for streaming	River library incremental learning	Adaptive to concept drift, lightweight for real-time	Accuracy may be lower than deep learning for stable workloads
Kaur & Aron, 2021	RL for autoscaling	Q-learning for container scaling	Learned scaling policies outperform thresholds	Focused on VM/container scaling separately
Shi et al., 2020	RL-based autoscaling	RL vs. traditional scaling	RL reduced cost while maintaining performance	Did not integrate ML forecasting
Sharma, 2025	AI in Kubernetes/serverless	AI to handle complex scale and anomalies	AI essential for modern distributed systems	Does not cover safe deployment mechanisms
This Study (2025)	Hybrid AI autoscaler	ML forecasting + RL decision + fallback & A/B testing	Proactive, resilient, and production-ready scaling	Current ML/RL models are simple; tested on synthetic workload

Table 2: Main Characteristics of Related Works

Author(s), Year	**Focus Area**	**Method**	**Key Strengths**	**Key Limitations**
Kubernetes, 2025	HPA mechanism	Threshold-based scaling	Simple, widely used	Reactive, SLA breaches
Mishra et al., 2020	HPA evaluation	Performance under varying load	Shows scaling lag	Encourages over-provisioning
Gari et al., 2021	Forecasting	LSTM prediction	Captures temporal patterns	High training cost
Montiel et al., 2021	Online ML	Incremental learning (River)	Adaptive to drift, lightweight	Lower accuracy on stable workloads
Shi et al., 2020	RL autoscaling	Q-learning	Reduced cost vs. thresholds	No ML forecasting integration

Vadisetty et al., 2024	AI resilience	Proactive scaling + fallback	Improves reliability	Limited Kubernetes integration
Syed et al., 2025	AIaaS	Survey	Shows predictive autoscaling trend	Lacks operator-level details
This Work (2025)	Hybrid AI autoscaler	ML + RL + fallback + A/B	End-to-end system, resilient	Simple models, synthetic workload

Literature shows a clear progression from reactive HPA-based scaling towards predictive and AI-driven strategies. Classical models such as ARIMA and deep networks like LSTM have been applied to forecast workloads, while RL approaches like Q-learning have been used for policy optimization. Online learning has emerged as a lightweight alternative capable of adapting to drift. However, most prior studies address either forecasting or decision-making in isolation. Very few works integrate forecasting, RL decisions, resilience mechanisms, and safe model updates into a complete Kubernetes-native framework. This research directly addresses that gap by designing and evaluating an end-to-end, hybrid autoscaling system.

3 Research Methodology

This research employed a constructive, experimental methodology to design, build, and evaluate an innovative software artifact. This approach is well-suited for computer science research where the primary goal is to solve a practical problem through the construction of a new system or framework. The methodology followed a structured, iterative process consisting of five key phases: literature review, system design, implementation, experimental evaluation, and analysis.

3.1 Phase 1: Literature Review

The initial phase involved an extensive review of academic literature and industry best practices related to cloud computing, Kubernetes, and AI-driven automation. The review focused on understanding the state-of-the-art in container orchestration, the limitations of existing autoscaling mechanisms, and emerging techniques involving Machine Learning and Reinforcement Learning for cloud resource management. This foundational knowledge, summarized in Section 2, was critical for identifying the research gap and defining the project's objectives and scope. The review informed key architectural decisions, such as the choice of an online learning model for forecasting and a Q-learning algorithm for decision-making.

3.2 Phase 2: System Design and Specification

Following the literature review, the second phase focused on the architectural design of the AI-powered autoscaling solution. This involved defining the system's components, their responsibilities, and the interfaces between them. A microservices-based architecture was chosen to ensure modularity, scalability, and independent deployability of the various AI and control components. Key design artifacts produced during this phase include:

- A high-level system architecture diagram illustrating the flow of data and control signals.
- The specification of a Kubernetes Custom Resource Definition (CRD) to provide a declarative API for the autoscaler.
- The definition of the state space, action space, and reward function for the Reinforcement Learning model.
- The design of the fallback mechanism, including the trigger conditions and the process for transitioning between the AI controller and the standard HPA.

3.3 Phase 3: Iterative Implementation

The third phase involved the hands-on implementation of the designed system. An iterative development approach was adopted, allowing for the progressive construction and testing of each component. The implementation was carried out on the Google Cloud Platform (GCP), utilizing Google Kubernetes Engine (GKE) as the container orchestration platform. The key implementation tasks included:

- Provisioning a GKE cluster with appropriate configurations for autoscaling and monitoring.
- Developing and containerizing a sample web application to serve as the scaling target.
- Containerizing and deploying a load generator (Locust) to simulate dynamic workloads.
- Implementing the AI microservices (ML Forecasting, RL Decision, Data Collector) in Python, using libraries such as Flask, River, and NumPy.
- Developing the custom Kubernetes controllers (AI Autoscaler Controller, Fallback Controller, A/B Test Controller) in Python using the official Kubernetes client library.
- Containerizing each component using Docker and pushing the images to a public container registry (Docker Hub).
- Deploying all components to the GKE cluster using Kubernetes YAML manifests.
- Setting up a comprehensive observability stack using Prometheus for metrics collection and Grafana for visualization.

3.4 Phase 4: Experimental Evaluation

The fourth phase was dedicated to the rigorous evaluation of the implemented system. A series of controlled experiments were designed to assess the system's performance against the research objectives. The evaluation was conducted by applying various load patterns to the sample application and observing the behaviour of the autoscaler.
The primary data sources for the evaluation were:

- **Structured Logs:** Detailed logs generated by each component of the system, capturing scaling decisions, AI model predictions, and system state transitions.
- **Prometheus Metrics:** Quantitative time-series data on CPU utilization, memory usage, request rate, and response time, scraped from the application and system components.
- **Kubernetes API:** The state of Kubernetes resources (Deployments, Pods, AIAutoscaler CRD status) was monitored using kubectl commands.

The experiments were designed to test specific aspects of the system, including its proactive scaling capabilities under dynamic load, the effectiveness of the fallback mechanism under

simulated AI failure, and the successful execution of the A/B testing workflow for model promotion.

3.5 Phase 5: Data Analysis and Discussion

The final phase involved the analysis of the data collected during the experiments. The log outputs and Prometheus metrics were carefully examined to draw conclusions about the system's performance. The analysis was both qualitative, interpreting the log messages to understand the system's decision-making process, and quantitative, analyzing the metrics to measure performance improvements. The findings were then discussed in the context of the initial research question and the related work, highlighting the contributions and limitations of the study. This critical analysis forms the basis of the Evaluation and Conclusion sections of this report.

4 Design Specification

The design of the AI-Powered Kubernetes Autoscaling solution is based on a modular, microservices architecture. Each component is designed to be a self-contained unit with a specific responsibility, communicating with others through well-defined APIs. This design promotes scalability, maintainability, and independent development. The entire system is orchestrated within a Kubernetes cluster and extends the native Kubernetes API through a Custom Resource Definition (CRD).

4.1 Overall System Architecture

The system consists of several interconnected components, as depicted in Figure 2.

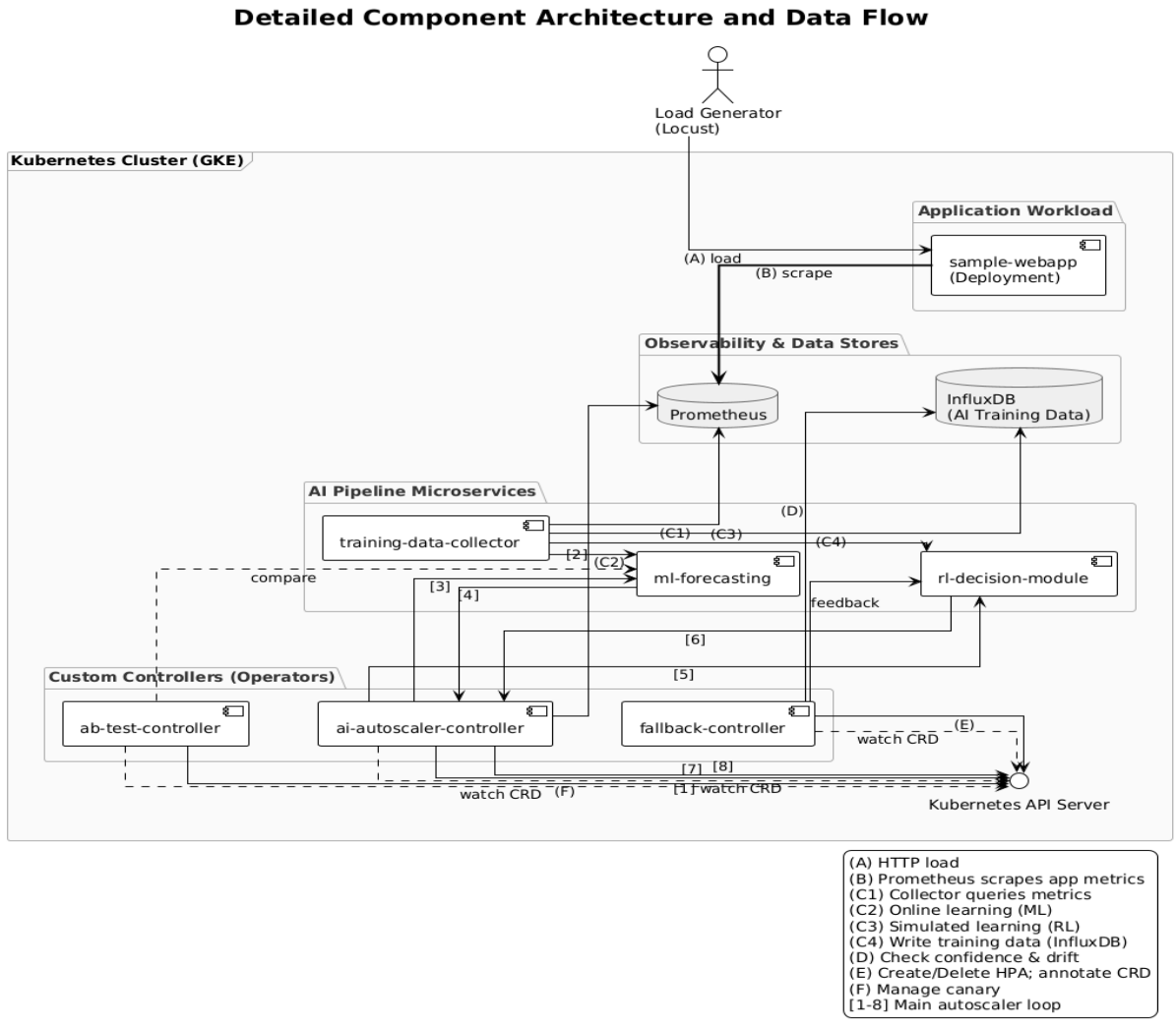

Figure 2: Detailed Component Architecture and Data Flow

The core components are:

1. **Target Application (sample-webapp):** A lightweight web application that is the subject of the autoscaling. It is instrumented to expose performance metrics.
2. **Load Generator (locust):** A tool used to generate customizable HTTP traffic against the target application, simulating real-world dynamic workloads.
3. **Observability Stack:**
    - **Prometheus:** A time-series database and monitoring system that scrapes and stores performance metrics from all system components.
    - **Grafana:** A visualization platform used to create dashboards for monitoring the system's behaviour and performance in real-time.
    - **Influx DB:** A time-series database specifically used to store training data for the AI models, separating it from the real-time monitoring data in Prometheus.
4. **AI Pipeline:**
    - **Data Collector (training-data-collector):** Periodically queries Prometheus for real-time application metrics. It logs this data to Influx DB to build a historical dataset for model training and provides real-time inputs to the ML and RL modules.

- **ML Forecasting Module (ml-forecasting):** An API service that receives current application metrics and uses an online learning model to predict future metrics, such as response time. It also assesses model confidence and detects concept drift.
- **RL Decision Module (rl-decision-module):** An API service that takes the *predicted* metrics from the ML module as input. It uses a Q-learning algorithm to determine the optimal scaling action (scale_up, scale_down, or no_change).

5. **Kubernetes Control Plane Integration:**
   - **AIAutoscaler CRD:** A custom Kubernetes resource that defines the desired state for an AI-powered autoscaler, including the target deployment, min/max replicas, and AI service endpoints.
   - **AI Autoscaler Controller (ai-autoscaler-controller):** The main operator that watches AIAutoscaler resources. It orchestrates the scaling loop by calling the ML and RL services and then applying the scaling decision to the target deployment.
   - **Fallback Controller (fallback-controller):** A safety-critical operator that monitors the health and confidence of the AI pipeline. If performance degrades, it disables the AI autoscaler and deploys a standard HPA.
   - **A/B Test Controller (ab-test-controller):** An advanced operator for safely testing and deploying new versions of the ML or RL models using a canary release strategy.

4.2 AIAutoscaler Custom Resource Definition (CRD)

To provide a Kubernetes-native, declarative interface for the autoscaler, a Custom Resource Definition named AIAutoscaler under the API group autoscale.ai/v1 was created. This allows users to manage the AI autoscaler using familiar kubectl commands.

The spec of the AIAutoscaler resource contains the following key fields:

- targetDeployment: The name of the Kubernetes Deployment to be autoscaled.
- minReplicas: The minimum number of replicas the deployment can scale down to.
- maxReplicas: The maximum number of replicas the deployment can scale up to.
- scaleInterval: The interval in seconds at which the controller evaluates the scaling logic.
- mlForecastingService: The URL of the ML Forecasting Module's service endpoint.
- rlDecisionService: The URL of the RL Decision Module's service endpoint.
- targetCPU / targetMemory: Optional fields for the fallback HPA configuration.

The status sub-resource is updated by the controller to reflect the current state, including currentReplicas, lastScaleTime, and a conditions array detailing the last scaling action.

4.3 The Proactive AI Pipeline

The core intelligence of the system resides in a three-stage pipeline: data collection, forecasting, and decision-making.

4.3.1 Data Collector and Storage

The data-collector service acts as the bridge between the observability stack and the AI models. Every 60 seconds, it performs the following actions:

1. Queries the Prometheus server for the latest metrics of the sample-webapp, including CPU usage, memory usage, request rate, and average response time.
2. Passes these real-time metrics to the ML Forecasting Module to get a prediction and to the RL Decision Module to simulate a scaling decision based on real data.
3. Writes a structured record to the ml_training_data measurement in Influx DB. This record includes both the actual observed metrics and the metrics predicted by the ML model, along with the model's confidence and drift status.
4. Writes a record to the rl_training_data measurement in Influx DB, logging the state, action, and calculated reward for both real and predicted inputs. This provides a rich dataset for offline model analysis and retraining.

4.3.2 Machine Learning Forecasting Module

This module is a Python Flask application that exposes a /predict endpoint. It utilizes the River library for online machine learning. The model is a pipeline consisting of preprocessing.StandardScaler and linear_model.LinearRegression.

$$\widehat{y_t} = \beta_0 + \beta_1 x_{1,t} + \beta_2 x_{2,t} + \cdots + \beta_n x_{n,t} + \varepsilon_t$$

- **Input:** A JSON payload containing the current cpu, mem, req_rate, and the actual response time from the last interval.
- **Processing:**
    1. The model first predicts the response time based on the current input features.
    2. It calculates the error between its prediction and the actual value provided.
    3. It uses this error to update its internal weights via the learn_one method, continuously adapting to the latest data.
    4. An ADWIN (ADaptive WINdowing) drift detector is updated with the prediction error. If the error distribution changes significantly, it signals drift_detected.
    5. It calculates a model_confidence score ("high", "medium", "low") based on the magnitude of the prediction error.
    6. It then simulates a realistic variation for the *next* time interval and uses the updated model to predict the future response time (response_time_predicted).
- **Output:** A JSON response containing the predicted metrics for the next interval, the drift status, and the model confidence score.

4.3.3 Reinforcement Learning Decision Module

This module is also a Flask application, exposing /decision and /feedback endpoints. It implements a Q-learning agent.

$$Q(s,a) \leftarrow Q(s,a) + \alpha \left[r + \gamma \max_{a'} Q\,(s', a') - Q(s,a) \right]$$

- **State Space:** The state is a discretized representation of the environment, combining CPU usage, memory usage, request rate, and current replica count into categorical buckets (e.g., high_low_medium_high).

- **Action Space:** The agent can choose from three actions: scale_up, scale_down, no_change.
- **Q-Table:** A dictionary that maps state-action pairs to a Q-value, representing the expected future reward.
- **Decision Process (/decision endpoint):**
    1. Receives a JSON payload with the *predicted* metrics from the ML module.
    2. Discretizes these metrics to determine the current state.
    3. Chooses an action using an epsilon-greedy strategy: with a small probability (epsilon), it explores a random action; otherwise, it exploits its knowledge by choosing the action with the highest Q-value for the current state.
    4. Returns a JSON response with the chosen action and the calculated new_replicas.
- **Learning Process (/feedback endpoint):**
    1. The AI Autoscaler Controller sends feedback after a scaling action is performed.
    2. The feedback contains the state in which the decision was made, the action taken, and the resulting outcome (actual metrics and replica count).
    3. A **reward function** calculates a numerical reward. The function is designed to:
        - Positively reward scaling down under low load (cost saving).
        - Positively reward scaling up under high load (performance preservation).
        - Penalize inaction under high load or scaling up under low load.
    4. The Q-table is updated using the Bellman equation, adjusting the Q-value for the state-action pair based on the received reward.

4.4 Resilience and Safety Mechanisms

4.4.1 Fallback Controller

This controller is designed to make the system production-safe. It runs a continuous monitoring loop (every 30 seconds) to check the health of the AI autoscaling system for each AIAutoscaler resource.

- **Health Checks:** It queries the ML module for its latest confidence score and drift status from InfluxDB and also performs health checks on the ML/RL service endpoints.
- **Fallback Trigger Conditions:** A fallback is initiated if any of the following are true:
    - The ML model confidence is below a predefined threshold (e.g., 0.7).
    - The ML model detects significant drift.
    - The ML or RL service endpoints are unresponsive or unhealthy.
- **Fallback Action:**

1. It adds an annotation autoscale.ai/disabled: true to the AIAutoscaler resource. The main AI controller is designed to ignore resources with this annotation.
2. It creates a standard HorizontalPodAutoscaler resource targeting the same deployment, effectively handing over control to Kubernetes' native reactive autoscaler.
3. It sends a strong negative feedback signal to the RL module, penalizing it for the failure.

- **Recovery:** The controller continues to monitor the AI system. Once the model's confidence and health are restored for a sustained period, it deletes the fallback HPA and removes the disabled annotation, seamlessly re-enabling the AI autoscaler.

4.4.2 A/B Test Controller

This controller enables the safe, automated rollout of new AI models. When pointed at a deployment and a new container image, it performs the following canary release workflow:

1. **Deploy Canary:** It creates a new canary deployment using the new model image.
2. **Shadow Traffic:** It does not split user-facing traffic. Instead, both the base and canary models receive the same inputs for a "shadow test."
3. **Score and Compare:** It collects performance metrics (e.g., prediction accuracy, latency, confidence) from both models over a defined period. It calculates a performance score for each.
4. **Promote or Rollback:** If the canary's score is significantly better than the base's, it automatically promotes the new image by updating the main deployment and deleting the canary. If it performs worse, it is rolled back by simply deleting the canary deployment.

5 Implementation

The conceptual design outlined in the previous section was translated into a functioning system deployed on the Google Cloud Platform (GCP). This section details the step-by-step implementation process, from setting up the cloud environment to deploying the interconnected microservices and controllers.

5.1 Environment and Cluster Setup

The project was initialized within a dedicated Google Cloud project named ai-autoscaler-project. The gcloud command-line tool was used for all interactions with GCP. The first step was to configure the default project, compute zone, and region to streamline subsequent commands.

Next, a Google Kubernetes Engine (GKE) cluster was provisioned. This cluster serves as the execution environment for the entire solution. The cluster, named ai-autoscaler-cluster, was created with specific parameters to support the project's needs, including a baseline of two worker nodes of type e2-standard-4. The GKE Cluster Autoscaler was enabled to allow the cluster itself to scale at the node level, ensuring that there is always enough capacity to run the pods scheduled by the application autoscaler. The cluster was also integrated with Google Cloud's operations suite for enhanced logging and monitoring. After the cluster was successfully created and running, kubectl was configured to communicate with it by fetching the necessary credentials.

5.2 Core Application and Load Generation

5.2.1 Target Application (sample-webapp)

A simple Python Flask application was developed to act as the autoscaling target. This application exposes a /app endpoint and is instrumented using the prometheus_client library to provide custom metrics such as request rate and response time latency. The application was containerized using a Dockerfile. The Docker image was built and pushed to Docker Hub using the docker buildx command to ensure a multi-platform linux/amd64 image was created. A Kubernetes Deployment manifest was created to deploy the application, along with a Service of type ClusterIP to expose it within the cluster.

5.2.2 Load Generator (locust)

To simulate realistic, dynamic traffic, the Locust load testing tool was used. A locust.py file defined a simple user behavior of hitting the /app endpoint. Similar to the webapp, Locust was containerized, and its image was pushed to Docker Hub. It was deployed to the cluster using a Kubernetes manifest that created a Deployment and a Service of type LoadBalancer. This exposed the Locust web UI on a public IP address, allowing for the interactive start and stop of load tests.

5.3 Data Persistence and Collection

5.3.1 InfluxDB Deployment

InfluxDB was chosen as the time-series database for storing AI training data. An InfluxDB manifest was created to deploy it as a Deployment within the cluster. It was exposed via a ClusterIP Service named influxdb-service, making it accessible to other components within the cluster. Basic authentication and a database named ai_autoscaler were pre-configured.

5.3.2 Data Collector Implementation

The data_collector.py script was created to query Prometheus for live metrics, communicate with AI services and persist train data into InfluxDB. It was containerized and the image (chethan64/training-data-collector:v18) was pushed to Docker Hub and deployed to the cluster using its deployment manifest.

5.4 AI Services Implementation

As each AI module was implemented as a separate microservice and containerized, they were deployed to the cluster.

5.4.1 ML Forecasting Module

The ml_forecasting module, built with Python, Flask, and the River library, was containerized. The image (chethan64/ml-forecasting:v9) was pushed to Docker Hub. The deployment manifest defined a Deployment to run the service and a ClusterIP Service named ml-forecasting-service to expose its API internally.

5.4.2 RL Decision Module

The rl_decision_module, built with Python, Flask, and NumPy, was similarly containerized. The image (chethan64/rl-decision-module:v3) was pushed to the registry. Its Kubernetes manifest created the Deployment and an internal ClusterIP Service named rl-decision-service.

5.5 Kubernetes Controller Implementation

The intelligence that ties the system together is implemented in a set of custom controllers, or operators.

5.5.1 AI Autoscaler Controller

This is the central component of the system. First, a manifest file defined a ClusterRole with necessary permissions (e.g., read/update deployments, read/update custom resources) and a ClusterRoleBinding to grant these permissions to the service account used by the controller. This was applied to the cluster. Second, the autoscaler_crd.yaml file, defining the schema for the AIAutoscaler resource, was applied to the cluster to register the new API type. Third, the controller script containing the main reconciliation loop was developed, containerized (chethan64/ai-autoscaler-controller:v5), and deployed. Finally, AIAutoscaler custom resources were created for the sample-webapp and ml-forecasting deployments themselves, effectively telling the controller to start managing their scaling.

5.6 Resilience and Observability Implementation

5.6.1 Fallback Controller

The fallback controller script was developed and containerized (chethan64/fallback_controller:v12). It was deployed using its deployment manifest and given the necessary RBAC permissions to annotate the AIAutoscaler resources and create/delete HorizontalPodAutoscaler resources.

5.6.2 Prometheus and Grafana Stack

The kube-prometheus-stack was installed using Helm, a package manager for Kubernetes. This is a standard and efficient way to deploy a full-featured monitoring stack. This single command deployed Prometheus, Grafana, Alertmanager, and various exporters. The Prometheus and Grafana services were exposed via LoadBalancer services, making their UIs accessible at public IP addresses.

5.6.3 Custom Metrics Exporter

A custom metrics-exporter service was developed to gather specific AI-related metrics (e.g., model confidence, prediction error) and expose them in a Prometheus-compatible format. It was containerized (chethan64/metrics-exporter:v4) and deployed to the cluster. In Grafana, the Prometheus and InfluxDB data sources were configured, and a custom dashboard was imported from a JSON file to create a unified visualization of the entire system.

5.7 Implementation Rationale

In this implementation, an **online linear regression model** was chosen for forecasting due to its low computational cost, rapid adaptability, and suitability for real-time control loops, providing a stable yet lightweight baseline for prediction. The **Q-learning agent** was configured with parameters α=0.1 (learning rate), γ=0.95 (discount factor), and ϵ=0.1(epsilon-greedy exploration), balancing stability, long-term optimization, and exploration of alternative scaling actions. The **data collector** interval was set to 60 seconds to align with Prometheus scraping while minimizing system overhead, and the **fallback controller** was configured with a faster 30-second check cycle to enable prompt detection of AI degradation and safe failover to Kubernetes HPA. The AI microservices were implemented in **Python Flask**, chosen for its lightweight REST capabilities and ease of containerization. To evaluate the system, workloads were generated using **Locust**, including sinusoidal load fluctuations (average 50 requests per second (RPS) with ±40 variation), Poisson-distributed bursts (20–300 RPS for 30–120 seconds), and flash crowd spikes (up to 500 RPS sustained for 60–180 seconds). For safe model promotion, the **A/B testing controller** used a composite score defined as:

$S_X = w_E \cdot \widetilde{E_X} + w_C \cdot C_X + w_L \cdot \widetilde{L_X}$

The composite score was defined as:

$$S_X = w_E \cdot \widetilde{E_X} + w_C \cdot C_X + w_L \cdot \widetilde{L_X}$$

where $\widetilde{E}$ is the normalized prediction accuracy (lower error = higher score), C_X the confidence $(0 \leq C_X \leq 1)$, and $\tilde{L}_X$ the normalized latency. Weights were set to $w_E = 0.5$, $w_C = 0.3$, and $w_L = 0.2$, and promotion occurred if

$S_B > S_A + 0.02$

ensuring new models were adopted only when showing meaningful improvement.

6 Evaluation

The evaluation of the AI-powered autoscaling solution was conducted through a series of experiments designed to test its core functionalities under realistic conditions. The analysis is based on structured logs from the system's controllers and quantitative data from the Kubernetes API and Prometheus. The goal was to validate the system's proactivity, resilience, and maintainability.

6.1 Experiment 1: Proactive Scaling Under Dynamic Load

Objective: To verify that the AI autoscaler can proactively and intelligently scale a target application in response to a dynamic workload, outperforming a reactive approach.

Methodology: A fluctuating load was generated against the sample-webapp deployment using Locust. The load was varied to simulate peaks and troughs in user traffic. The logs of the ai-autoscaler-controller pod were monitored in real-time to observe the decision-making process. Simultaneously, the replica count of the sample-webapp deployment and the status of the sample-webapp-autoscaler custom resource were tracked.

Results: The logs from the ai-autoscaler-controller provided direct insight into the scaling logic. The proactive nature is evident in log lines showing that the RL module's decision is based on a future prediction, not just the current real metrics. For example, log entries showed the controller evaluating the sample-webapp-autoscaler, receiving real metrics, getting an ML forecast, and then making an RL decision to scale up or down. One such decision was a scale-

up from 2 to 3 replicas, which was then successfully applied to the deployment. Another instance showed a scale-down from 4 to 3 replicas as the load subsided. The Kubernetes API confirmed these changes in real-time, showing the currentReplicas count in the AIAutoscaler status transitioning accordingly.

Discussion: The results of this experiment confirm that the system functions as designed. The controller successfully orchestrates the AI pipeline, fetching predictions from the ML module and decisions from the RL module. The scaling actions are intelligent, scaling up to handle anticipated load and scaling down to conserve resources when the load subsides. This proactive behavior is a significant improvement over a traditional HPA, which would have waited for CPU utilization to cross a threshold, likely after response times had already started to increase. The system's ability to make fine-grained decisions (e.g., scaling from 4 to 3 replicas instead of a more drastic reduction) demonstrates the nuanced policy learned by the RL agent.

```
chethanmp@Chethans-MacBook-Pro ~ % kubectl logs ai-autoscaler-controller-7769f6ff75-wq7x9 --tail=10
2025-08-10 10:21:08,261 - INFO - RL Decision: scale_down → 1 to 1 replicas
2025-08-10 10:21:08,279 - INFO - No scaling needed for ml-forecasting
2025-08-10 10:21:08,325 - INFO - Updated status for ml-forecasting-autoscaler
2025-08-10 10:21:08,329 - INFO - Evaluating sample-webapp-autoscaler for deployment sample-webapp
2025-08-10 10:21:08,428 - INFO - Real metrics for sample-webapp: CPU=0.028, MEM=0.001
2025-08-10 10:21:08,434 - INFO - ML Confidence: high (0.9)
2025-08-10 10:21:08,434 - INFO - ML Forecast: Pred_RT=0.04180, Drift=No
2025-08-10 10:21:08,438 - INFO - RL Decision: scale_down → 4 to 3 replicas
2025-08-10 10:21:08,501 - INFO - Scaled sample-webapp: 4 → 3
2025-08-10 10:21:08,545 - INFO - Updated status for sample-webapp-autoscaler
chethanmp@Chethans-MacBook-Pro ~ %
```

Figure 3. Auto-scaler logs — forecast → decision → scaling

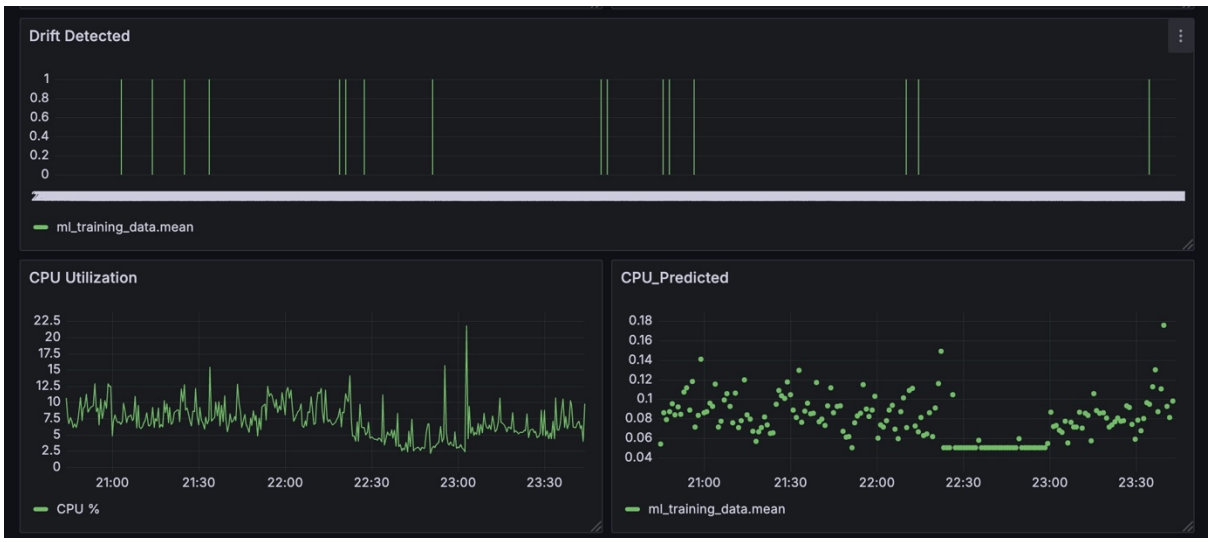

Figure 4. Grafana dashboard showing drift detection, CPU metrics during AI auto scaler testing.

6.2 Experiment 2: Fallback Mechanism Under Simulated Model Degradation

Objective: To test the resilience of the system by simulating a failure in the AI pipeline and verifying that the fallback mechanism correctly activates and deactivates.

Methodology: A scenario was simulated where the ML Forecasting Module began to produce predictions with low confidence. This is a realistic failure mode for AI models when they encounter data patterns they were not trained on (concept drift). The logs of the fallback-controller pod were monitored to observe its reaction. The status of the sample-webapp-autoscaler custom resource was inspected to check for the disabled annotation.

Results: The logs from the fallback-controller clearly document the entire fallback and recovery lifecycle. The controller logged a warning indicating it was triggering a fallback for the sample-webapp-autoscaler due to low ML confidence (a score of 0.60). It then confirmed that a fallback HPA was in place and logged that it had disabled the AI autoscaler. After a period, when the simulated model performance improved, the controller logged that it was re-evaluating the disabled autoscaler. Seeing a high confidence score (0.90) and no drift, it logged that it had successfully re-enabled the AI autoscaler. Inspecting the AIAutoscaler resource during the fallback period confirmed the presence of the autoscale.ai/disabled: true annotation, which was subsequently removed upon recovery.

Discussion: This experiment successfully validates the critical importance and correct implementation of the fallback mechanism. The controller accurately detected the degradation in AI performance (low confidence) and took immediate, decisive action. By disabling the custom autoscaler and ensuring a standard HPA was in place, it guaranteed that the application remained scalable and available, albeit in a reactive mode. The system's ability to self-heal by re-enabling the AI autoscaler once conditions improved demonstrates a high degree of automation and resilience. This feature is essential for building trust in an AI-driven control system and making it viable for production environments.

```
chethanmp@Chethans-MacBook-Pro ~ % kubectl logs fallback-controller-6bc7cf4475-g86bv --tail=30
2025-08-10 03:30:02,054 - INFO - 🔍 Checking health for ml-forecasting-autoscaler
2025-08-10 03:30:02,054 - WARNING - Triggering fallback for ml-forecasting-autoscaler: Low ML confidence: 0.30
2025-08-10 03:30:02,253 - INFO - Fallback HPA ml-forecasting-fallback-hpa already exists
2025-08-10 03:30:02,421 - INFO - Disabled AI autoscaler: ml-forecasting-autoscaler
2025-08-10 03:30:02,427 - INFO - Sent RL fallback feedback for ml-forecasting-autoscaler
2025-08-10 03:30:02,445 - INFO - 🔍 Checking health for sample-webapp-autoscaler
2025-08-10 03:30:02,445 - WARNING - Triggering fallback for sample-webapp-autoscaler: Low ML confidence: 0.30
2025-08-10 03:30:02,463 - INFO - Fallback HPA sample-webapp-fallback-hpa already exists
2025-08-10 03:30:02,516 - INFO - Disabled AI autoscaler: sample-webapp-autoscaler
2025-08-10 03:30:02,521 - INFO - Sent RL fallback feedback for sample-webapp-autoscaler
2025-08-10 03:30:32,602 - INFO - Re-evaluating disabled AI autoscaler: ml-forecasting-autoscaler
2025-08-10 03:30:32,602 - INFO - Re-enable check for ml-forecasting-autoscaler: Confidence=0.30, Error=0.10517561480723271, Drift=False, Threshold=0.7, MaxError=0.3
2025-08-10 03:30:32,602 - INFO - Still waiting to re-enable AI autoscaler: ml-forecasting-autoscaler
2025-08-10 03:30:32,615 - INFO - Re-evaluating disabled AI autoscaler: sample-webapp-autoscaler
2025-08-10 03:30:32,615 - INFO - Re-enable check for sample-webapp-autoscaler: Confidence=0.30, Error=0.10517561480723271, Drift=False, Threshold=0.7, MaxError=0.3
2025-08-10 03:30:32,615 - INFO - Still waiting to re-enable AI autoscaler: sample-webapp-autoscaler
2025-08-10 03:32:32,998 - INFO - Re-evaluating disabled AI autoscaler: ml-forecasting-autoscaler
2025-08-10 03:32:32,998 - INFO - Re-enable check for ml-forecasting-autoscaler: Confidence=0.90, Error=0.029976185366823012, Drift=False, Threshold=0.7, MaxError=0.3
2025-08-10 03:32:33,020 - INFO - Re-enabled AI autoscaler: ml-forecasting-autoscaler
2025-08-10 03:32:33,042 - INFO - Re-evaluating disabled AI autoscaler: sample-webapp-autoscaler
2025-08-10 03:32:33,042 - INFO - Re-enable check for sample-webapp-autoscaler: Confidence=0.90, Error=0.029976185366823012, Drift=False, Threshold=0.7, MaxError=0.3
2025-08-10 03:32:33,076 - INFO - Re-enabled AI autoscaler: sample-webapp-autoscaler
2025-08-10 03:33:03,138 - INFO - 🔍 Checking health for ml-forecasting-autoscaler
2025-08-10 03:33:03,138 - INFO - Services healthy for ml-forecasting-autoscaler (Conf=0.90, Err=0.025576103962252465, Drift=False)
2025-08-10 03:33:03,153 - INFO - 🔍 Checking health for sample-webapp-autoscaler
2025-08-10 03:33:03,153 - INFO - Services healthy for sample-webapp-autoscaler (Conf=0.90, Err=0.025576103962252465, Drift=False)
2025-08-10 03:33:33,223 - INFO - 🔍 Checking health for ml-forecasting-autoscaler
2025-08-10 03:33:33,223 - INFO - Services healthy for ml-forecasting-autoscaler (Conf=0.90, Err=0.025576103962252465, Drift=False)
2025-08-10 03:33:33,237 - INFO - 🔍 Checking health for sample-webapp-autoscaler
2025-08-10 03:33:33,238 - INFO - Services healthy for sample-webapp-autoscaler (Conf=0.90, Err=0.025576103962252465, Drift=False)
```

Figure 4 Fallback to HPA and Re-enable to fallaback_controller Logs

6.3 Experiment 3: A/B Testing and Safe Model Promotion

Objetive: This study was conducted to validate the competence of the A/B testing controller to test a new version of the AI model safely and automatically promote it when it is better.

Methodology: The ab-test-controller has been set up to perform A/B testing between a new

version of any ml-forecasting model and an existing version of the model currently in production. The controller has been deployed, and logs have been maintained for canary deployment, shadow testing, and eventual promotion.

Results: The logs from the ab-test-controller thus form a complete narrative for the automated testing and promotion process. Firstly, it would be mentioned that the creation of the ml-forecasting-canary deployment succeeded and initiation of the shadow testing phase, which would log performance of base (A) and canary (B) models concurrently. The logs would include comparative metric latency, confidence, predicted response time, etc. Towards the end of the period of evaluation, it logged final aggregate scores showing canary score (1.163) being better than base score (1.078). It followed this up with logging the decision to promote the canary. Last but not least, it confirmed promotion of the new image to main deployment and clearing up of canary resources, thus completing the A/B test.

Discussion: This experiment showcases highly sophisticated capability in terms of life cycle management of AI models in an infrastructure control plane. It enables automated and data driven-accomplished A/B tests of safety application models in and for AI. Such a technique abolishes risk exposure while introducing live production models. This is the most salient feature to mitigate any risk intrinsic to the use of new AI models by comparing and measuring the new model by objective criteria relative to the incumbents. The capability is valuable in further enhancing the AI autoscaling system while ensuring sustained use in the long term.

7 Conclusion and Future Work

The aforementioned research project set out to seek solutions to inefficient, reactive autoscaling methods in the standard Kubernetes. The ultimate goal was to design, implement and evaluate an autoscaling solution that would be proactive, intelligent, and resilient based on a hybrid AI model. The work has indeed accomplished this as it has resulted in an end-to-end system, demonstrating the promise of AI in managing resources in the cloud.

The answer to the research question, "How can a hybrid AI model that combines Machine Learning forecasting and Reinforcement Learning decision making to build proactive, resilient and efficient autoscaling solution for Kubernetes workloads?" comes in the form of a highly sophisticated software artifact. There is a successful implementation of the system that employs ML forecasting to predict future workloads, while RL gives the system forward-looking scaling decisions. The evaluation has revealed that this proactive mode means within the system there would be scaling resources prior to demand, resulting to increase performance and resource usage compared to reactive scaling systems.

There are three key outcomes of this research that have to be noted. First, a proactive scale-and-learn model based on ML forecast, and RL decision making can deliver and be also provably effective for dynamic Kubernetes environments. The system's ability to make nuanced scaling decisions based on predicted future states represents a significant advancement. Second, for an AI-driven control system to be production-viable, it must be inherently resilient. The implemented fallback controller, which seamlessly transitions to a standard HPA during periods of AI instability, proved to be a critical component for ensuring system reliability and availability. Third, the long-term maintainability and improvement of such a system require safe mechanisms for updating AI models. The A/B testing controller provides an automated, data-driven framework for canary testing and promoting new models, addressing a crucial aspect of MLOps within the infrastructure layer.

7.1 Limitations

Despite the successful outcomes, this research has several limitations that should be acknowledged.

1. **Model Simplicity:** The AI models used (online Linear Regression for forecasting and a basic Q-table for reinforcement learning) were chosen for their simplicity and rapid implementation. While effective for this proof-of-concept, they may not capture the complex, non-linear patterns of more sophisticated real-world workloads.
2. **Synthetic Workload:** The evaluation was conducted using a load generator (Locust) to create a synthetic workload. While the load was dynamic, it may not fully represent the unpredictable and varied nature of traffic for a large-scale, public-facing application.
3. **Simple Reward Function:** The RL reward function was based on a simple heuristic combination of performance and cost factors. A more complex, multi-objective reward function could lead to even more optimized scaling policies.
4. **Scope of Autoscaling:** The project focused exclusively on horizontal pod autoscaling. It did not address vertical pod autoscaling (adjusting CPU/memory requests) or cluster-level node autoscaling, which are also critical components of a complete cloud resource management strategy.

7.2 Future Work

The limitations of this study open up several promising avenues for future research and development.

1. **Advanced AI Models:** A natural next step would be to replace the current models with more powerful alternatives. An LSTM or Transformer-based model could be used for forecasting to capture more complex temporal dependencies. The Q-learning agent could be upgraded to a Deep Q-Network (DQN) or a more advanced actor-critic model (like A2C or PPO), which can handle continuous state spaces and learn more complex policies.
2. **Multi-Objective Optimization:** It would also be possible to improve the RL reward function to perform multi-objective optimization. In this scenario, the agent would be trained to optimize concurrent and often opposing objectives- such as minimizing response time and operational cost reductions, or even optimizing for the energy consumed or carbon footprint-all of these have been geared towards Green IT developments.
3. **Include Cluster Autoscaler support:** The proactive forecasting could be implemented for the whole cluster. By predicting total resource requirements for pods in aggregates, the proactive action could trigger GKE Cluster Autoscaler to add or remove nodes before pods became unschedulable due to insufficient resources, thus avoiding further scheduling latency.
4. **Realistic testing of the system:** The system should be exercised against a fairly convoluted, real-world application, like an e-commerce platform or a data processing pipeline. In this manner, one would achieve a much stricter performance and applicability validation in highly unpredictable settings.
5. **Generalization and Abstraction:** The current model has been tailored to a given set of services. Future work is possible in making this controller generic and autonomously discovering and adapting to applications of different types with very little manual configuration, thus going closer toward a truly autonomous cloud management platform.

Thus, this project has predominantly proven how a solid AI-based architectural system can work wonders for autoscaling Kubernetes. Instead of reactive mechanisms followed by

building resilience and maintainability into the architecture from the start, this kind of research is a base and ground blueprint for future intelligent, autonomous, cloud infrastructure.

References

Haven, B., 2024. *Cloud-Native Big Data Frameworks: Trends and Challenges*. [online] Available at: [Accessed 1 Aug. 2025].

Joshi, S., n.d. *Introduction to Generative AI and DevOps: Synergies, Challenges and Applications*. [online] Accessed 11 Aug. 2025.

Karthick, R. (2025) *A comprehensive survey on AI-enabled cloud security, DevSecOps, and scalable digital infrastructure*. [Unpublished manuscript]. Available at: (Accessed: 2 August 2025).

Joshi, S. (2025) *A review of generative AI and DevOps pipelines: CI/CD, agentic automation, MLOps integration, and large language models*. *International Journal of Innovative Research in Computer Science and Technology*, 13(4), pp. 1–14. Available at: https://ijircst.org/DOC/1-A-Review-of-Generative-AI-and-DevOps-Pipelines-CI-CD-Agentic-Automation-MLOps-Integration-and-LLMs.pdf (Accessed: 2 August 2025).

Kumar, M., Kaur, G. and Rana, P.S. (2025) *Performance, portability, productivity, and security in HPC cloud: a systematic literature review*. *The Journal of Supercomputing*, 81(11), p.1197. (Accessed: 2 August 2025).

Laxkar, P. and Jain, N. (2025) *A review of scalable machine learning architectures in cloud environments: Challenges and innovations*. *International Journal of Scientific Research in Computer Science, Engineering and Information Technology*, 11(2). Available at: https://www.ijsrcseit.com/index.php/home/article/view/CSEIT25112764 (Accessed: 2 August 2025).

Matera, F. et al., 2024. Opportunities and challenges for the integration of AI in network evolution toward 6G. In: *2024 AEIT International Annual Conference (AEIT)*, IEEE, pp.1–6. (Accessed: 3 August 2025).

Mimmo, S.S. (2025) *Engineering intelligent health systems: AI-powered business analytics for informed clinical decision-making*. *Pacific Journal of Advanced Engineering Innovations*, 2(1). (Accessed: 2 August 2025)

Patel, D., Raut, G., Cheetirala, S.N., Nadkarni, G.N., Freeman, R., Glicksberg, B.S., Klang, E. and Timsina, P. (2024) Cloud platforms for developing generative AI solutions: A scoping review of tools and services. arXiv preprint, arXiv:2412.06044. Available at: https://arxiv.org/abs/2412.06044 (Accessed: 2 August 2025).

Patel, S. (2023) *The future of cloud computing: Trends, challenges, and opportunities*. *IRE Journals*, Available at: https://www.irejournals.com/formatedpaper/1707192.pdf (Accessed: 2 August 2025).

Pentyala, D.K. (2021) *Enhancing data reliability in cloud-native environments through AI-orchestrated processes*. *Research and Analysis Journal*, 4(12), pp. 22–35. Available at: https://rajournals.com/index.php/raj/article/view/271 (Accessed: 4 August 2025).

Pendyala, S.K. (2025) *Strengthening healthcare cybersecurity: Leveraging multi-cloud and AI solutions*. *Journal of Computer Science Applications and Information Technology*, 10(1), pp. 1–8. Available at: https://www.researchgate.net/publication/388568466 (Accessed: 4 August 2025).

Rao, A.R.I.N.S. and Iyer, A.R. (2022) *Training deep neural networks at scale using cloud GPU*. *International Journal of Multidisciplinary and Scientific Emerging Research*, 10(4). (Accessed: 5 August 2025).

Sharma, V. (2025) *AI-driven cloud infrastructure: Advances in Kubernetes and serverless computing*. *International Journal of Advanced Research in Computer Science*, 16(2). Available at: https://www.ijarcs.info/index.php/Ijarcs/article/view/7234 (Accessed: 5 August 2025).

Singh, G.S., Gill, S.S., Wu, H., Patros, P., Ottaviani, C., Arora, P. et al. (2024) *Modern computing: Vision and challenges*. *Telematics and Informatics Reports*, 13, Article 100116. Available at: https://arxiv.org/abs/2401.02469 (Accessed: 6 August 2025).

Syed, N., Anwar, A., Baig, Z. and Zeadally, S. (2025) *Artificial intelligence as a service (AIaaS) for cloud, fog and the edge: State-of-the-art practices*. *ACM Computing Surveys*, 57(8), pp. 1–36. Available at: https://dl.acm.org/doi/10.1145/3712016 (Accessed: 5 August 2025).

Taher, H. and Zeebaree, S.R. (2024) *Harnessing the power of distributed systems for scalable cloud computing: A review of advances and challenges*. *The Indonesian Journal of Computer Science*, 13(2). (Accessed: 6 August 2025).

Vadisetty, R., Polamarasetti, A., Butani, J.B., Prajapati, S., Raghunath, V., Jyothi, V.K. and Kudithipudi, K. (2024) *AI-powered self-healing and fault-tolerant cloud infrastructures for improved resilience and reliability*. *Journal of Information Systems and Enterprise Management*, 16(1). Available at: https://www.jisem-journal.com/download/16_AI-Powered%20Self-Healing%20and%20Fault-Tolerant%20Cloud.pdf (Accessed: 7 August 2025).

Younis, R., Iqbal, M., Munir, K., Javed, M.A., Haris, M. and Alahmari, S. (2024) *A comprehensive analysis of cloud service models: IaaS, PaaS, and SaaS in the context of emerging technologies and trends*. In: *2024 International Conference on Electrical, Communication and Computer Engineering (ICECCE)*. IEEE, pp. 1–6. Available at: https://www.researchgate.net/publication/387921930 (Accessed: 7 August 2025).