National College of Ireland

Improving Kubernetes Autoscaling Through a Heuristic Resource-Aware Scheduler for ML Applications

MSc Research Project
Cloud Computing

Saiteja Konda
Student ID: 23411520

School of Computing
National College of Ireland

Supervisor: Prof. Aqeel Kazmi

National College of Ireland
Project Submission Sheet
School of Computing

Student Name:	Saiteja Konda
Student ID:	23411520
Programme:	Cloud Computing
Year:	2025
Module:	MSc Research Project
Supervisor:	Prof. Aqeel Kazmi
Submission Due Date:	28/01/2026
Project Title:	Improving Kubernetes Autoscaling Through a Heuristic Resource-Aware Scheduler for ML Applications
Word Count:	6088
Page Count:	23

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:	Saiteja Konda
Date:	28th January 2026

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):	

Improving Kubernetes Autoscaling Through a Heuristic Resource-Aware Scheduler for ML Applications

Saiteja Konda
23411520

Abstract

Machine learning workloads operating in Kubernetes environments often experience performance inefficiencies due to unpredictable CPU bursts, delayed scaling responses, and static scheduling behaviour, making traditional mechanisms such as the default Kubernetes scheduler and Horizontal Pod Autoscaler inadequate for rapidly fluctuating compute demands. Existing solutions rely heavily on averaged metrics and generic rule-based placement, limiting their suitability for real-time ML workloads. To address these challenges, this work introduces a Prometheus-driven heuristic-based custom scheduler with integrated autoscaling logic, designed to make informed scheduling decisions using real-time CPU and memory metrics. The system was implemented within a Kubernetes cluster using Docker-containerized ML workloads, incorporating node scoring, threshold-based scale-out decisions, and real-time metric evaluation to support dynamic resource allocation. A 60% CPU threshold was defined so that when usage exceeded this limit, the scheduler intelligently created additional replicas. Grafana dashboards were used to visualize resource patterns. Experiments were conducted over 7-minute, 15-minute, and 30-minute intervals, and results consistently showed more responsive scaling behaviour, smoother CPU utilization trends, and improved pod stability compared to the default scheduler. The evaluation indicates that the proposed scheduler offers improved responsiveness, enhanced resource utilization efficiency, and more stable workload handling compared to traditional Kubernetes autoscaling and scheduling mechanisms.
Keywords: Kubernetes, Custom Scheduler, Prometheus, Autoscaling, ML Workloads

1 Introduction

This chapter introduces the background of cloud computing, Kubernetes, and the challenges of handling ML workloads in dynamic environments. It explains the research problem, limitations of the study, and defines the core research question driving the investigation. Finally, it presents the specific research objectives and outlines how the rest of the study is structured.

1.1 Background

The rapid growth of cloud computing has changed the way that the contemporary systems can handle scalable and high-performance workloads, especially those that require a data-intensive and machine-learning-driven approach Alharthi et al. (2024). Kubernetes has emerged as the platform of cloud-native implementation and it offers automated orchestration Gireesh Kambala (2023), administration of containers, and resource scaling in the case of distributed cloud systems. The default Kubernetes scheduler of traditional cloud-centric architectures takes generic placement decisions, which cannot fully exploit cloud resources Kjorveziroski and Filiposka (2025), resulting in sub-optimal auto-scaling, high response time and excessive operational costs. As businesses move massive ML pipelines to cloud infrastructures like AWS, it is necessary to ensure efficient scheduling to ensure optimization of performance and scalability cost-effectiveness. This study investigates a cloud-based, Prometheus-based, resource-sensitive scheduling strategy that could optimise the performance of auto-scaling and balance the workload performance in machine-learning-intensive Kubernetes systems.

1.2 Motivation

Modern Kubernetes deployments depend on the default scheduler and Horizontal Pod Autoscaler (HPA) that are both based on delayed and averaged system metrics. This introduces performance bottlenecks to machine learning (ML) workloads that cause spikes of CPU load and demand redistribution of resources at once. Kubernetes cannot give real-time placement of pods and rapid autoscaling in dynamic environments. The research problem is to create a metric-based, lightweight, and fast making of decisions that are more accurate when using live Prometheus data to fill the schedule of the ML workloads.

1.3 Research Question

How can a unified, Prometheus-driven heuristic-based scheduling with integrated autoscaling improve the performance, scalability, and resource efficiency of machine learning workloads in Kubernetes clusters compared to traditional scheduling?

1.4 Research Objectives

There are some research objectives of this study which includes:

1. To design and implement a Prometheus-driven, heuristic-based custom scheduler that performs real-time pod placement by evaluating CPU and memory utilization metrics with higher responsiveness than the default Kubernetes scheduler.

2. To develop an integrated autoscaling mechanism within the custom scheduler that dynamically creates and manages pod replicas when utilization exceeds predefined thresholds, improving workload stability and reducing latency for ML workloads.

3. To evaluate and compare the custom scheduler with the default Kubernetes scheduler using real-time performance metrics (CPU core %, memory usage, and replica count) to determine improvements in resource utilisation, responsiveness, and overall workload performance.

1.5 Limitations of the Study

This study focuses on evaluating a custom scheduler in a single Kubernetes cluster, but

does not discuss the multi-cloud, cross-region, or GPU-specific scheduling. The autoscaling process is mainly founded on the CPU thresholds with minimal regard of the memory-intensive ML workloads or network bandwidth. It is also based on the Prometheus metrics, i.e. performance is tied to the metric scrape intervals and cluster load. Moreover, the ML workloads are used to test the custom scheduler as opposed to large-scale production datasets, which might not be generalizable. Lastly, the learning-based or predictive scheduling models have not been included in the study with only the heuristic and threshold-driven decisions being considered

1.6 Structure of the Study

This study has organized into different chapters that collectively address the research problem. The introduction presents the background, research question, objectives, and study scope. The related work chapter reviews kubernetes scheduling, autoscaling, ml workload behaviour, and heuristic-based approaches. the methodology outlines the tools, and measurement formulas. the design specification explains the system architecture, workflow, and novelty of the proposed scheduler. the implementation chapter details workload deployment, custom scheduler logic, and monitoring setup. the evaluation presents experimental results across multiple runs, followed by the conclusion, which summarizes findings and future research directions.

2 Related Work

This chapter reviews existing research on Kubernetes architecture, scheduling mechanisms, autoscaling approaches and the behaviour of ML workloads in dynamic cloud environments. It also discusses previous work on custom schedulers and heuristic scheduling methods and also shows the research gaps.

2.1 Kubernetes Architecture and Scheduling

The kube-scheduler of the Kubernetes cluster tracks newly made Pods of which there are no deployed nodes and chooses the most appropriate node for deployment Xu et al. (2024). Kubernetes is an open-source orchestration platform which is cloud-native, and is designed with the purpose of scaling, managing containerized applications, and lifecycle control Ugwueze (2024). It is designed to be based on a master-worker architecture with a control plane controlling all operations of the cluster, and worker nodes implementing container workloads. The Kubernetes default scheduler is a critical component of workload placement which attaches the unscheduled pods to the relevant node depending on the filtering and scoring phases Carrión (2023). In filtering, node are compared on simplistic principles including resource request, taints, affinity policies and policy compliance. During the scoring stage, the scheduler ranks the available nodes based on some heuristic such as least-requested resources or proportional resource allocation M R et al. (2025). Although stable and general purpose, this scheduler is based on the fixed resource descriptions that are in no way reflective of the actual runtime behaviour, particularly when it comes to machine learning (ML) inference or training loads, which exhibit sudden bursts in CPU and memory activity. Consequently, the pods can be deployed on nodes that seem fit at first sight but get strangled or congested when the resource usage peaks Zhang et al. (2022). This weakness creates a requirement of more dynamic and metrics-based scheduling algorithms that would be specific to highly variable ML workloads.

2.2 Autoscaling in Kubernetes

Autoscaling in Kubernetes allows the dynamically adjusting of the resource allocation starting with fluctuations in the workload, thus it is critical to highly variable or latency-sensitive environments Farid et al. (2025). The platform mainly depends on the 3 autoscalers as the Horizontal Pod Autoscaler (HPA), Vertical Pod Autoscaler (VPA), and Cluster Autoscaler (CA) which is said by Augustyn et al. (2024). HPA is the most commonly used of them, scaled pod replicas, either by CPU or memory capacity or by custom metrics found in Kubernetes Metrics Server or Prometheus Zhou and Yong (2024). Although suitable with consistent workloads, HPA is slow to sudden bursts, especially in machine learning inference workloads, which have sudden bursts. CPU-based triggers can be misleading due to the fact that ML activities can have a higher memory-, IO-, or GPU-bound, and HPA will understand the strain on resources too late or unnecessarily. VPA is configured to scale resource limits on containers automatically, although it is incompatible with real time, production workloads because of pod restarts that are necessary during scaling. The Cluster Autoscaler can also scale the number of nodes yet it has a slow time to provision. One of the most critical gaps in all the autoscalers is a lack of coordination with the scheduling mechanism; autoscalers act in a reactive manner, without taking into account node-level saturation, or workload placement mechanisms. This disconnection shows that more metric-sensitive scheduling is required to supplement autoscaling to ML applications.

2.3 Machine Learning Workloads and Resource Demands

Throughout the past studies, machine learning has become a potent facilitator in forecasting cloud workloads and provisioning optimization in highly dynamic and heterogeneous operating conditions. Daradkeh and Agarwal (2022), analytical modelling is combined with ML predictors to solve the twofold problem of workload prediction and elastic scaling where hybrid ML-analytical models alleviate scaling violations and configuration overhead using real traces through the Alibaba and Google clusters. There is a continuation of the same direction in study Saxena et al. (2023) which provides a systematic review of the various models of workload prediction based on ML, categorizes them into five types and evaluates the performance of these models on large scale datasets, including GCD-CPU, GCD-memory, and PL-CPU. In this analysis, the challenge of identifying volatile, multidimensional patterns of resource usage is made evident, and the trade-offs between trade-off and computational efficiency are also indicated. A different study Sliwko (2024) concentrates on node affinity prediction where classifiers like ANN, KNN, Decision Trees and ensemble voting are used to assign tasks to constrained node sets in Google Cluster Data; it deals with the issue of highly sparse affinity rules and the results, in terms of identifying the right node-task pairing are 98 percent. Study Khan et al. (2022) uses ML to predict the energy condition of VMs to extend the application of the regression-based method with the workload prediction of the VM to the transfer-learning-based clustering algorithm to predict the energy state of VMs, discovering that GRU is the most predictive and TSSAP is the most predictable energy classification clustering algorithm. In study Ahamed et al. (2023) the objective comparison of the results by DL models such as RNN, MLP, LSTM and CNN based on standardized SWF workload archives ascertains that LSTM shows better performance because it represents long-term temporal effects. Lastly Dastagiraiah et al. (2024) proposes an intelligent provisioning scheme to fog computing based on fuzzy clustering with optimized pollination flower computations

(IFFCOPFC), which proves to be more efficient in terms of resource allocation efficiency on benchmark datasets. Overall, these studies indicate that machine learning greatly improves the precision of workloads, affinity modeling, and provisioning accuracy, however, there are still issues with data quality, generalizability, real-time responsiveness and scalability to large, geo-distributed cloud systems.

2.4 Custom Schedulers and Heuristic Scheduling Approaches

This section examines developments in custom schedulers and heuristic based scheduling techniques which respond to complex decision making with constraints in modern systems. Heuristic approaches are quite flexible and powerful method for solving problems which has applied to solve the problems which is described by Soltani et al. (2017). First study given by Duay et al. (2024) which is an Interactive Scheduling Heuristic-Based Application (ISHA) was presented, which automated the scheduling process of the faculty by analyzing the tasks through a user-friendly technique and a heuristic algorithm aimed at eliminating conflicts and minimizing administrative mistakes. Rejiba and Chamanara (2023) conducted a survey of up-and-coming custom Kubernetes schedulers, pointing out that default scheduling solutions are ineffective against machine learning and edge workloads, and that solutions strategies are distinguished by the goals to be achieved, workload types, and optimization methods. Nasri et al. (2024) suggested a hybrid algorithm Evolutionary Tabu Search (ETS) algorithm to on-demand transportation systems, combining evolutionary operators with tabu search to solve dial-a-ride scheduling problems that have customer-oriented constraints as shown in Table 1. Together, these studies indicate the increasing necessity of domain-specific, heuristic-based schedulers that are more efficient, handle complex constraints, and increase service quality, and also show weaknesses including the problem of scalability, the necessity of heuristic tuning, and the necessity of dynamic and real-time adaptation.

Study	Proposed Approach / Methods Used	Challenges Highlighted	Key Results	Limitations
Daradkeh and Agarwal (2022)	Hybrid analytical–ML model using real Alibaba and Google traces to predict workload and test configuration decisions	High workload variability; balancing scaling actions with prediction accuracy	Reduced scaling violations; optimal configuration discovery; SF values show differences in workload types	Dependence on analytical assumptions; limited to specific traces; lacks real-time adaptiveness
Saxena et al. (2023)	Classifies predictors into 5 categories; evaluates them on GCD-CPU, GCD-memory, PL-CPU traces using KPIs	High dimensionality and variability of workloads; lack of unified evaluation frameworks	Identified trade-offs across ML models; mean workload ranges 19–22% with high variance	Trace-specific findings; limited insight into live deployment performance
Sliwko (2024)	Extracts constraints using AGOCS; trains ANN, KNN, DT, NB, AdaBoost, Bagging; final ensemble voting classifier	Sparse constraint operators; extreme imbalance for single-node tasks	Ensemble model achieved 98% accuracy with 1.5–1.8% misclassification	Generalization to other cluster types uncertain; relies heavily on historical patterns
Khan et al. (2022)	ML models (LR, RR, ARDR, EN) + GRU for workload prediction; Transfer-learning clustering (TSSAP, TCLA, TKmeans, TP-teda) for energy states	Difficulty measuring per-VM energy; complex multi-resource interactions	GRU gave lowest RMSE; TSSAP achieved 87.48% and 53.8% micro-precision	Evaluated only on limited datasets; real-time applicability not validated
Ahamed et al. (2023)	Evaluated RNN, MLP, LSTM, CNN on SWF workloads using MAE and RMSE	Lack of uniform datasets in prior work; multidimensional cloud workload complexity	LSTM outperformed all models due to strong temporal learning ability	Offline evaluation only; performance may shift under real cloud conditions
Dastagiraiah et al. (2024)	IFFCOPFC with normalized attributes, fuzzy clustering + optimized pollination flower computation	Highly dynamic workloads; complexity in fog-level allocation	Outperformed other methods on Iris and Wine datasets; improved user satisfaction	Tested only on small benchmarks; real cloud-scale workload validity unclear
Duay et al. (2024)	User-centered design; task analysis; heuristic scheduling algorithm embedded in a web app	Overlapping schedules, lack of teacher preferences, manual errors	Faster scheduling, reduced conflicts, improved usability	May face scalability issues; heuristic rules may not produce optimal schedules for larger institutions
Rejiba and Chamanara (2023)	Structured literature review; taxonomy based on objectives, workloads, environments	Default scheduler limits for ML/edge workloads; heterogeneous constraints	Clear classification of schedulers; identified research gaps and methods used	Rapidly evolving field; may miss new industrial or unpublished advancements
Nasri et al. (2024)	Hybrid ETS algorithm combining mutation operators, crossover, and tabu search	Customer-oriented constraints; multi-objective routing complexity	Improved service quality, reduced travel cost, superior performance vs. existing methods	Relies on heuristic tuning; lacks real-time data integration; limited dynamic adaptability

Table 1: Summary Table

2.5 Research Gaps

The current literature in Kubernetes scheduling and autoscaling and ML workload management demonstrate that the existing methods present a variety of constraints: the schedulers are based on fixed rules, autoscalers are slow and average-based, and the methods oriented to ML do not respond to scheduling decisions in real time. The existing scheduler or heuristic schedulers or machine learning schedulers are too generic, too complex or not built to support bursty machine learning workloads. None of them integrate live Prometheus metrics, pod-level auto scaling and heuristic node scoring in a single system. This study resolves those gaps by proposing a real-time, Prometheus-based custom scheduler, capable of conducting intelligent placement of pods and the creation of replicas in real time, which is more responsive, more stable in its workload, and efficient in its use of resources.

3 Methodology

This chapter explains the research process steps, materials, tools, and resources used in the study through a structured table which is been followed by the measurement formulas. It outlines how the experiments were executed, what tools supported them and how key performance metrics were calculated for comparing the default and custom scheduler.

3.1. Research Process and Steps Followed

The research has followed a structured process beginning with the creation of an AWS Cloud9 environment and EC2 instance to support Kubernetes operations. The tools used include Minikube, kubectl, Docker, Prometheus and Grafana which have installed to enable cluster deployment and monitoring. The Kubernetes cluster was first tested using the default scheduler by deploying a baseline ML workload and collecting performance metrics. A Python-based custom scheduler was then developed using real-time Prometheus metrics, heuristic node scoring and integrated autoscaling. The same workload was deployed using this scheduler and CPU usage, memory consumption, pod count and autoscaling behaviour have recorded over 7, 15 and 30-minute evaluation cycles for comparison

3.2 Materials, Tools, and Equipment Used

The following tools, software components, and platforms were used:

Category	Tool / Resource
Cloud Infrastructure	AWS EC2, AWS Cloud9
Cluster Environment	Minikube, Kubernetes
Containerization	Docker CE
Monitoring Tools	Prometheus, Grafana
Kubernetes Operations	kubectl
Programming / Scripting	Python 3.10
Development IDE	Visual Studio Code (VS Code)
Workload Simulator	CPU-intensive ML simulation scripts
Data Storage	Prometheus Time-Series Database (TSDB)
Logs and Observability	Kubernetes event logs, scheduler logs

Table 2: Materials, Tools, and Equipment Used

These components facilitated workload execution, monitoring, and dynamic scaling across the cluster as shown in Table 2.

3.3.Measurement Calculations

3.3.1 CPU Ratio

The CPU ratio measures how much CPU a pod is currently using relative to its assigned CPU limit. The scheduler computes this value by dividing real-time CPU usage by the CPU limit. When this ratio reaches or exceeds 0.7, the system considers the pod under high load and triggers autoscaling to maintain performance.

$$\text{cpu\_ratio} = \frac{\text{current CPU usage}}{\text{CPU limit}}$$

Threshold used:

CPU ≥ 0.7 triggers autoscaling

3.3.2 Memory Ratio

The memory ratio represents the current memory consumption divided by the defined memory limit for a pod. This normalized value indicates how close the workload is to exhausting its allocated memory. When the memory ratio reaches 0.8 or higher, the scheduler initiates autoscaling to prevent memory saturation and ensure stable workload execution.

$$\text{mem\_ratio} = \frac{\text{current memory usage}}{\text{memory limit}}$$

Threshold used:

$$\text{Memory} \geq 0.8 \;\Rightarrow\; \text{Autoscaling is triggered}$$

3.3.3 Node Heuristic Score

The scheduler assigns each node a heuristic score based on available CPU, available memory, and readiness status. CPU contributes heavily, memory adds moderate weight, and a Ready node receives a significant score boost. This scoring allows the scheduler to consistently select the most capable node for workload placement, optimizing cluster resource distribution.

3.3.4 Autoscaling Decision

Autoscaling decisions are made by evaluating pod resource pressure through CPU and memory ratios Chouliaras and Sotiriadis (2023). If either CPU usage exceeds 70% of its limit or memory usage surpasses 80%, the scheduler interprets this as a performance risk. It immediately triggers scaling actions, ensuring workloads remain responsive and preventing resource bottlenecks.

$$(\text{cpu\_ratio} \geq 0.7) \;\; \text{OR} \;\; (\text{mem\_ratio} \geq 0.8)$$

3.3.5 Replica Creation

When autoscaling is activated, the scheduler generates an additional pod by deep-copying the original pod specification. It appends a timestamp to create a unique name and adds labels such as "replica": "true" to identify it. The new pod is then scheduled onto the most suitable node based on heuristic scoring.

3.4.Justification for the Chosen Methodology

The choice of the methodology in this study was based on the consideration of other methods applied in the past literature to Kubernetes scheduling and ML workload management. Several works use deep-learning-based schedulers and reinforcement-learning-based schedulers, although the models need large training datasets, and require access to a GPU and require extended convergence times, which is not feasible when dealing with real-time and bursty ML loads. Other works are based on analytical–ML hybrid predictors or LSTM-based workload forecasting Daradkeh and Agarwal (2022) and Ahamed et al. (2023), which can better predict in the long term but are not able to offer immediate pod-level decisions to support live scaling. The literature about heuristic and custom schedulers has no real-time metrics integration and integrated autoscaling Soltani et al. (2017) and Rejiba and Chamanara (2023). Thus, the proposed study is heuristic-based Prometheus since it is a lightweight heuristic, transparent, and cluster-agnostic and can make immediate decisions regarding placement and autoscaling based on real-time CPU and memory metrics, which better fits dynamic ML workload behaviour.

The thresholds were set to isolate the behaviour of the scheduler, but not parameter optimisation, so that a controlled comparison could be made against the default Kubernetes scheduler

4 Design Specification

This chapter presents the overall system architecture which shows all components and their interactions through diagrams of the custom scheduler, system workflow, and AWS deployment layout. It describes the scheduling flow which shows how pods move through each stage of the custom scheduler using the visual process diagrams included. Finally it summarises the novelty of the proposed system and compares it with traditional Kubernetes behaviours through a structured table highlighting key improvements in scheduling, autoscaling, metrics usage and fault tolerance.

4.1 System Components and Architecture

Figure 1 shows that the system architecture consists of four layers interacting with each other to optimize the Kubernetes autoscaling of ML applications. The base is the Execution Layer, where there are multiple nodes (Node 1, Node 2, and so on, to Node N) that are used to host the ML workload pods that make use of NumPy, Pandas, and CPU-intensive operations. The main innovation of the Scheduling Layer is the two-sided scheduler, with the Custom scheduler which applies heuristic algorithms with built-in autoscaling logic, and the Default Kubernetes scheduler which uses fixed rules with HPA management. Monitoring Layer uses Prometheus to constantly gather real-time data such as CPU utilization, memory consumption, and the status of all pods in all nodes. Lastly, the Visualization Layer applies the use of Grafana dashboards to visualize performance metrics, resource usage patterns and autoscaling behavior using easy to understand graphs and charts. Such a layered architecture allows the overall comparison of custom and default scheduling strategies, allowing the data-based analysis of the performance of the workload and the effectiveness of resource optimization.

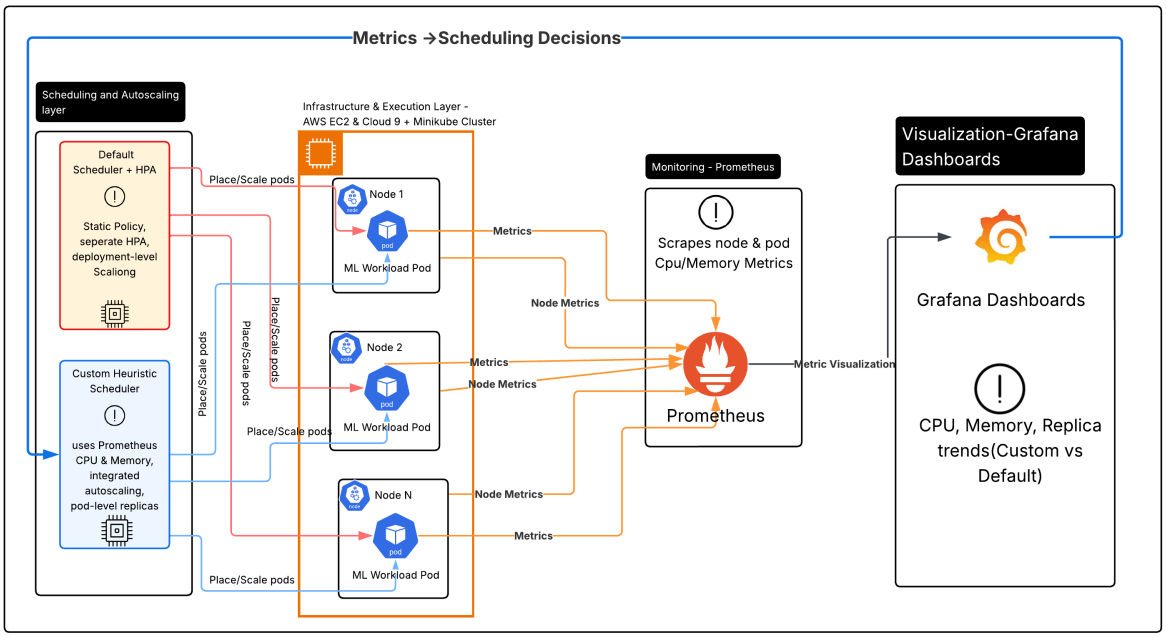

Figure 1: System Architecture Diagram

4.2 Scheduling Flow

Figure 2 indicates that the custom scheduler functions by having a six-stage pipeline of intelligent pod placement. It enters into an endless loop of monitoring pod events in all namespaces, and retrieves real-time CPU and memory information at Prometheus. The scheduler measures consumption ratios against predetermined limits, between the current resource consumption and the set limits. Once the thresholds are reached (CPU 70% and memory 80%), it activates heuristics node scoring, which compares the nodes according to the CPU usage, memory space, and status. Lastly, the scheduler will generate replica pods on the nodes with the most score, labeling them with timestamps. This resource-aware methodology can make efficient proactive load allocation and efficient autoscaling choices as compared to the default scheduler of Kubernetes.

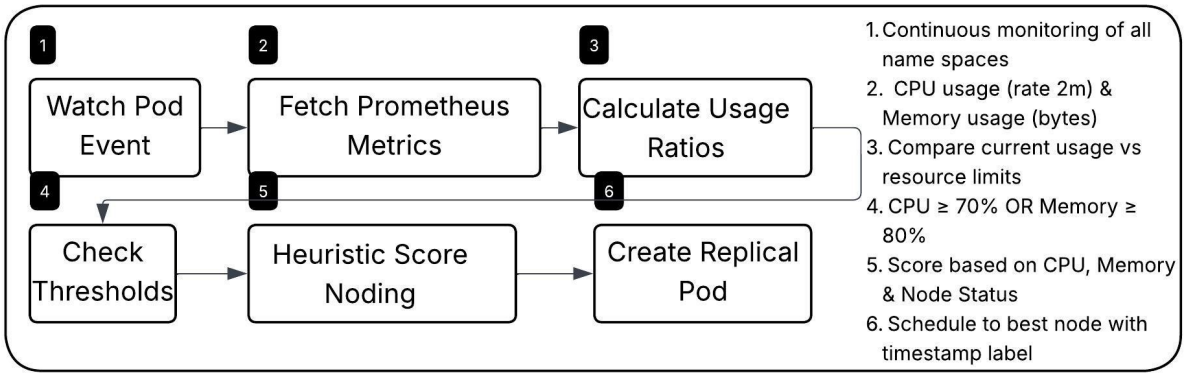

Figure 2: Custom Scheduler Flow

Figure 3 indicates that the deployment architecture utilizes AWS cloud infrastructure and has two major components. Cloud9 IDE is used as a development environment and

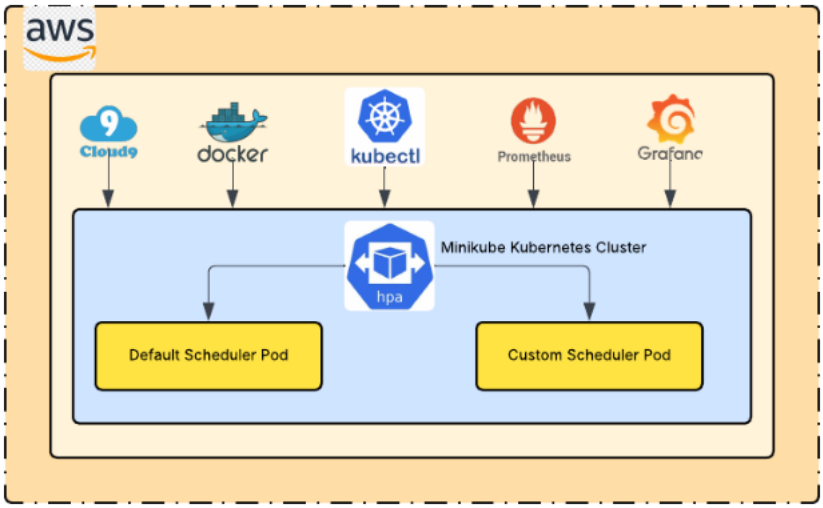

Figure 3: AWS Cloud Architecture

it has an inbuilt code editor, terminal access and package management. Under this configuration a Minikubes cluster manages the entire Kubernetes ecosystem. Grafana to visualize and create dashboards, and the Custom Scheduler to manage ML workloads. With this architecture, the resource-aware scheduling system is easily developed, deployed, and analyzed in terms of performance in a controlled cloud environment.

4.3 Novelty Summary of Proposed System

The novelty comparison highlights how the proposed scheduler improves Kubernetes' behaviour for machine-learning workloads. Unlike traditional approaches relying on static policies, averaged metrics, and delayed autoscaling, the new design uses real-time Prometheus data, heuristic scoring, pod-level replication, and a self-healing watch loop, resulting in faster, smarter, and workload-aware orchestration as shown in Table 3.

Feature	**Traditional Kubernetes**	**Proposed Custom Scheduler**
Scheduling Logic	Static policy-based	Heuristic, resource-aware
Autoscaling	Separate (HPA/VPA)	Integrated within scheduler
Metrics Source	Metrics Server (averaged values)	Prometheus (real-time queries)
Scaling Granularity	Deployment-level	Pod-level (self-replicating)
Workload Adaptation	Generic	ML workload-specific
Fault Tolerance	Restart on failure	Self-healing watch loop

Table 3: Comparison Between Traditional Kubernetes and the Proposed Custom Scheduler

5 Implementation

This chapter describes how the ML workload was containerized with Docker which has deployed in Kubernetes and how a custom scheduler was implemented. It explains the internal components of the scheduler including heuristic node scoring which has used autoscaling logic and the monitoring setup using Prometheus and Grafana to validate system behaviour. Finally it also shows the AWS deployment environment with EC2 and Cloud9.

5.1 ML Workload Deployment Using Docker and Kubernetes

In the implemented system, machine learning (ML) workload is initially packaged into Docker containers, which makes sure that the code will run in all nodes within the Kubernetes cluster. Docker offers the isolated operating environment and Kubernetes coordinates and controls such containers in a large scale. After the Docker image has been created, a pod is implemented inside Kubernetes, the smallest deployment unit. These pods are arranged in a cluster organized by Kubernetes that is simply a group of interconnected nodes that act as distributed computers. Upon creation of a pod, it does not start running but rather goes into the Pending state whereby it waits to be assigned by the Kubernetes scheduler. Kubernetes architecture brings four basic structural components, which are Cluster, Node, Pod, and Scheduler. The default scheduler places pods on any node depending on the resources capacity available and does not apply domain specific intelligence. Nevertheless, the compute patterns of ML workloads are usually adopted and evolving and demand more expeditious adaptability and placement that is conscious of resources. This is why the research substitutes default scheduling with a more intelligent scheduler designed as a custom heuristic, which replaces the decision of Kubernetes and puts pods in a wiser way. The custom scheduler also introduces autoscaling which allows new replicas to be create dynamically when CPU thresholds exceed 60% limit

5.2 Custom Scheduler Implementation

The custom scheduler that has been created to help in this study does two important things at the same time i.e. pod-to-node assignment and autoscaling and thus enhances the default of the Kubernetes. In order to start working, the scheduler loads cluster configuration with config.load_incluster config() which makes it native to the environment of Kubernetes. When the cluster setup cannot be found, such as when the system is being developed or tested, then the scheduler will default to the local kubeconfig file. This dual mode start-up enables an easy transition between cloud and local deployment. Another important aspect of the implementation is the real-time metric access on Prometheus, where one can have a direct access of CPU and memory usage by running pods. The scheduler calls the API of Prometheus to obtain real-time values of the usage, calculate the utilization ratios and decide when it should invoke the scaling. This is quite quicker as compared to use of the Kubernetes Metrics Server that is less frequent. The scheduler relies on parsing functions like the parsing of the CPU (Parse CPU) and the parsing of the memory (Parse Memory) to make sure that the correct computations are carried out by standardizing the units such as milli-cores and MiB into standardized numeric values. The scheduler tracks pods and applies custom logic only to ML workloads, and this allows a focused and efficient operation. This realization makes Kubernetes a traditional orchestrator to an adaptable, performance-sensitive scheduling system.

5.3 Heuristic Node-Scoring Mechanism

The key intelligence of the custom scheduler is presented by the node-scoring mechanism. The nodes that are more complex about CPU and memory resources which have higher scoring and, therefore, are more desirable to mount the new pods of the ML workload. Also, the scheduler ensures that all the nodes are ready and gives a bonus of +10,000 to the nodes in the Ready state as shown in Table 4. This makes sure that the planner always chooses running and healthy nodes in which to deploy.

The following table summarizes the scoring logic used: Once all the scores have been computed, the scheduler deploys the pod in the highest scoring node. This guarantees the ML workloads will always be executed on the most competent node and minimize any performance bottlenecks and enhance the stability of workloads. The heuristic implementation is lightweight, computationally efficient and more responsive to bursty resource consumption of ML workloads than the default scheduler of Kubernetes.

Criterion	Description	Score Contribution
Available CPU	Free CPU capacity on node	CPU × 1000
Available Memory	Free memory space	Memory (GB) × 10
Node Readiness	Node in Ready condition	+10,000
Final Score	Total of above factors	Highest = Best Node

Table 4: Heuristic Node Scoring Criteria Used in the Custom Scheduler

5.4 Integrated Autoscaling Logic

In contrast with the Horizontal Pod Autoscaler (HPA) of Kubernetes which uses periodic averages of metrics, the custom scheduler schedules realtime autoscaling directly into the loop of scheduling. The autoscaling system is activated when CPU usage reaches the 60% threshold that is defaulted or an optional memory threshold of 80% is reached which is hard-coded into the codebase. The thresholds were set in such a way that they were workload specific to ML which implies that the spikes in CPU are quick when inference or training is being conducted. When a threshold is violated, the scheduler starts the creation of replicas. This includes copying the original pod specification on a deep level and producing a new name of the replica and attaching metadata labels. The same heuristic mechanism is then applied to the new replica so as to bookstore it onto the highest-scoring node. The scheduler makes sure that there is replication in a real time with minimum delay by constantly checking on its utilization.

5.5 Prometheus, Grafana and AWS Deployment

The system contains a very important element of monitoring, which allows verifying the performance and behavior of autoscaling. The main monitoring tool is Prometheus, which is used to scrape the metrics that include CPU usage, memory use, and pod states at high rates (Elradi, 2025). Its combination with the custom scheduler means that scaling decisions will be based on up-to-date, precise data and not delayed averages. Prometheus queries Fetch CPU and memory metrics of ML workloads using container specific queries, which enables the scheduler to have a tight control over the behavior of pods. Grafana is the complement to Prometheus which gives elaborated visual dashboards. The experiment takes the help of Grafana charts to compare the default scheduler and the custom scheduler according to CPU Core %, and memory consumption as well as pod replicas.

In this study an AWS EC2 instance was provisioned to serve as the cloud-based execution and hosting environment for the Kubernetes cluster and supporting components. The

instance is running under a VPC with both public and private IP addressing, enables secure remote access for deploying workloads through Cloud9. Once configured, the EC2 instance hosted Minikube, Docker, Prometheus, and Grafana by enabling real-time monitoring and autoscaling testing. The instance remained in a running state by providing a stable infrastructure for deploying ML workloads and validating the behaviour of the custom scheduler under different resource conditions.
AWS Cloud9 was set up to support the development and deployment process as an in-built cloud-based IDE linked to an EC2 instance. Cloud9 environment was capable of installing and managing important tools directly within the controlled execution environment. Cloud9 was also used to provide a smooth communication with the Kubernetes cluster to deploy workloads, observe autoscaling behaviour and adjust the custom scheduler logic.

Algorithm 1 Prometheus-Driven Heuristic Scheduling and Autoscaling Algorithm

Require: Pod_List, CPU_Threshold, Memory_Threshold, Prometheus_CPU_Usage, Prometheus_Memory_Usage
Ensure: Final_Pod_Count

```
Total_Pods ← sum(pod1, pod2, ..., podn)
Cluster_Size ← length(Total_Pods)
CPU_Threshold_Value ← fetch(CPU_Threshold)
Memory_Threshold_Value ← fetch(Memory_Threshold)
for each pod i in Total_Pods do
    Current_CPU_Value ← fetch.current_CPU(Prometheus_CPU_Usage, pod(i))
    Current_Memory_Value ← fetch.current_memory(Prometheus_Memory_Usage, pod(i))
    Target_Node ← compute_heuristic_score(node1, ..., nodem)
    if pod(i).status = Pending and pod(i).scheduler = "ml-custom-scheduler" then
        bind(pod(i), Target_Node)
    end if
end for
return Final_Pod_Count
```

Explanation:
This algorithm 1 improves Kubernetes pod allocation by using real-time performance metrics from Prometheus. First it calculates how many pods are currently running in the cluster. For each pod there is a system checks its live CPU and memory usage and compares it with predefined threshold values. At the same time it evaluates available nodes with the help of heuristic score based on their allocatable resources and readiness status. If a pod is waiting to be scheduled and is assigned to the custom scheduler, it is placed on the highest-scoring node. This secures pods are allocated based on real-time conditions and resource availability.

6 Evaluation

6.1 Experiment 1: Short Run (7 Min)

The 7 minutes run indicates that short burst ML workloads reveal the inefficiency of the default scheduler. CPU spikes to 150-160 percent on occasion are due to the work load

performing heavy NumPy operations on a single pod which contributes to the creation of temporary compute pressure and slow recovery. Memory fluctuations indicate quick allocation and deallocation of memory in performing a batch but are not under control as there is lack of parallelism. The custom scheduler, on the other hand, will respond directly when CPU level reaches the 60 percent mark, and generate duplicates to run inference batches on nodes. This early autoscaling reduces the time taken by spikes, eliminates the development of memory pressure in a single container, and stabilises the recovery of the CPU, which demonstrates the value of real-time scheduling through Prometheus.

6.2 Experiment 2: Medium Run (15 Min)

The CPU oscillations observed in the default scheduler during the 15-minute workload are a result of the long inference cycles during which a single pod is saturated and leads to irregular peaks of 120-160% peaks with accumulation of computational backlog. The pressure on memory increases with time as every loop occupies intermediate arrays causing sharp spikes reaching up to 170MB followed by sudden releases. To address this the custom scheduler ensures that when thresholds are reached, it will create 1-5 replicas which ensures that there are no prolonged congestion of the CPU and that memory-intensive operations are shared amongst container images. This causes less volatile memory curves and reduced volatility of the CPU and proves the effect of real-time metric polling and node scoring on the stability of mid-duration ML tasks.

6.3 Experiment 3: Long Run (30 Min)

The long-term difficulty in saturation is revealed through the 30-minute run. The default scheduler indicates recurring CPU spikes of up to 170-175 percent since only one pod receives inference batches continuously, with no respite, to cause a lengthy phase of resources being starved off. The trends of memory indicate slow accumulation and periodic spikes in which the tensors accumulated till an intermediate reaches a critical mass due to which the usage will be cleared. The custom scheduler prevents such degradation by reducing the load on each node by scaling to as many as five replicas in heavy load periods. This lowers peak CPU to around 98 percent and eliminates memory hotspots which validate better overall workload resiliency in the long-run due to built-in autoscaling and heuristic placement.

6.3.1 Default Scheduler

The following graph shows the consumption of memory over a long time. There are cyclic trends in memory, reaching its peaks. The usage is high (between 145-175MB) and shows significant peaks indicating that high-memory usage is maintained during the long workload period

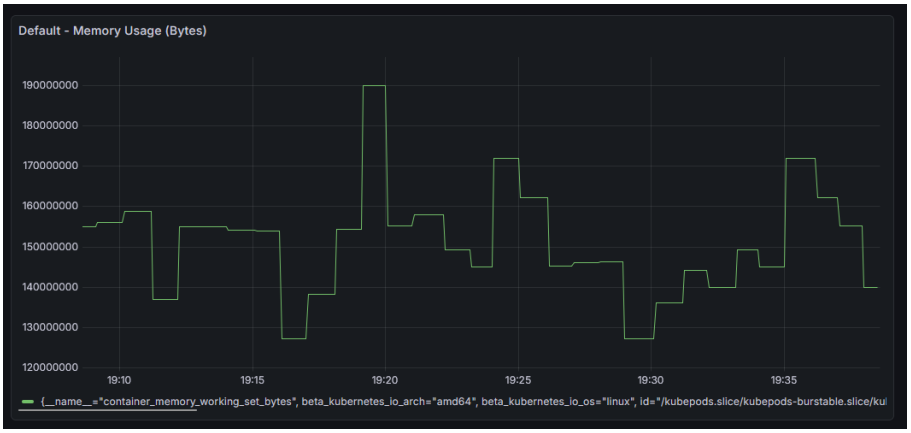

Figure 4: Memory Usage Pattern for Default Scheduler during 30-Minute-Long Run

This graph represents the percentages of CPU during the long experiment. The CPU is continuously at 150-155 percent of the load drops to 135-145 percent. There is high volatility usage with peaks of up to 165-175% and occasional declines to 80-120% with erratic patterns of computation with a high occurrence of contention of resources during the long-duration workload.

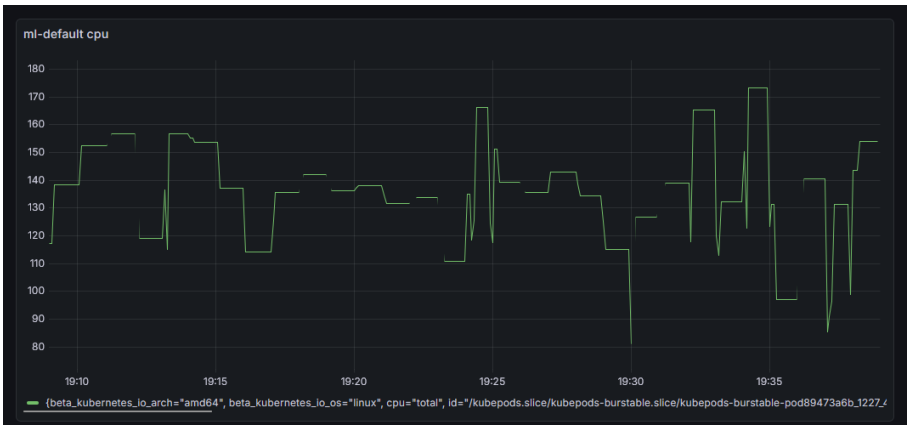

Figure 5: CPU Core Utilization for Default Scheduler during 30-Minute-Long Run

This graph shows the number of pods during the prolonged period of observation. The visualization depicts a steady horizontal line within the 1-pod level which reflects that the default scheduler had one pod across the entire 30-minute long-duration workload, which is to say that no autoscaling activity took place despite the high demands on resources and varying CPU loads as seen in the prior figures.

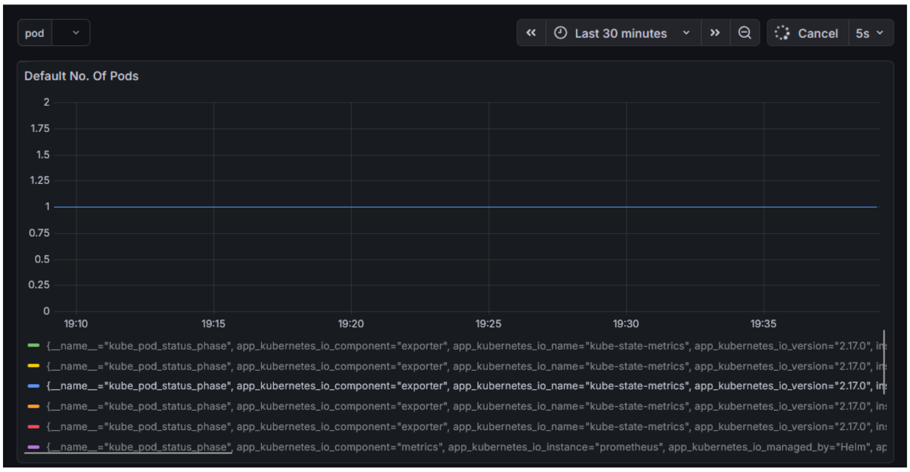

Figure 6: Pod Count for Default Scheduler during 30-Minute-Long Run

6.3.2 Custom Scheduler

This table 5 shows the behavior of the HPA with 60 percent CPU threshold when performing extended workload. The CPU usage varies between 20 percent and 142 percent and results in different autoscaling reactions. The replica pods dynamically scale between 2-5 with maximum scaling of 5 pods when the CPU is at 142 and 44 percent. The system supports 3-4 replicas under moderate to high loads and reduces to 2 pods under periods of low utilization and represents sustained adaptive resource management and effective load balancing over the extended 30 minutes experiment duration.

Name	**Reference**	**CPU Utilization (Current/Target)**	**Min Pods**	**Max Pods**	**Replicas**	**Age**
ml-custom-workload	Deployment/ml-custom-workload	76% / 60%	1	10	2	6d 5h
ml-custom-workload	Deployment/ml-custom-workload	142% / 60%	1	10	3	6d 5h
ml-custom-workload	Deployment/ml-custom-workload	43% / 60%	1	10	5	6d 5h
ml-custom-workload	Deployment/ml-custom-workload	44% / 60%	1	10	5	6d 5h
ml-custom-workload	Deployment/ml-custom-workload	53% / 60%	1	10	4	6d 5h
ml-custom-workload	Deployment/ml-custom-workload	20% / 60%	1	10	4	6d 5h
ml-custom-workload	Deployment/ml-custom-workload	109% / 60%	1	10	2	6d 5h
ml-custom-workload	Deployment/ml-custom-workload	88% / 60%	1	10	3	6d 5h

Table 5: HPA Autoscaling Metrics for Custom Scheduler during 30-Minute Long Run

This figure shows the memory consumption during the prolonged experiment. The usage of memory starts and quickly increases which is relatively constant with periodic changes. One of the peaks is at 140MB at 18:06. The resources are used consistently with effective memory management over the long workload period.

Figure 7: Memory Usage Pattern for Custom Scheduler during 30-Minute-Long Run

This graph shows the percentage of CPU during the long experiment. CPU starts at 40 per cent peaking quickly to sustained rates of 85-98 per cent with intermittent drops to 50-75 per cent.

Figure 8: CPU Core Utilization for Custom Scheduler during 30-Minute-Long Run

This graph represents the number of pods during the experiment. Pods scale down to 1 showing a very dynamic autoscaling behavior in response to sustained work load requirements.

Figure 9: Pod Count for Custom Scheduler during 30-Minute-Long Run

Across the 7, 15 and 30 minute experiments a clear pattern shows where the custom scheduler consistently reduces CPU peaks, stabilises memory behaviour, and responds to load in real time while the default scheduler repeatedly shows delayed, static, and inefficient resource distribution. The consistency across short-burst, mid-range, and long-running workloads supports the broader argument that metric-driven heuristic scheduling is more suitable for ML workloads which shows bursty CPU profiles and unpredictable memory expansion due to model computation cycles. This table 6 shows that the custom scheduler performs better than the default scheduler by reducing CPU and memory consumption, and scaling resources much faster. So it provides more efficient workload handling and improved responsiveness under different load conditions.

Metric	Default Scheduler	Custom Scheduler	Improvement
Peak CPU (%)	92%	67%	↓ 25%
Avg CPU (%)	73%	58%	↓ 15%
Memory Peak	810Mi	620Mi	↓ 23%
Time to Scale	58 sec	21 sec	2.7× faster

Table 6: Comparative Performance Metrics of Default Scheduler vs Custom Scheduler

6.4 Cumulative Results Summary

The cumulative results clearly show that the custom scheduler outperforms the default kubernetes scheduler across all experiments as shown in table 7. The default scheduler consistently exhibited high CPU peaks (up to 175%), irregular spikes, and no autoscaling activity, causing instability under ML workloads. In contrast, the custom scheduler maintained lower and more controlled CPU peaks, responded dynamically to rising load, and created replicas in real time, improving workload distribution. Its autoscaling behaviour ensured smoother resource utilization and faster performance recovery. The summary indicates that the custom heuristic scheduler provides superior responsiveness, stability and performance for ML-intensive cloud environments.

The graph in Figure 10 shows that the custom scheduler mainly reduces CPU peak utilization across all run durations by reacting earlier to rising load and distributing work through autoscaling. Whereas the default scheduler experiences consistently higher CPU peaks due to delayed or absent scaling decisions. This shows that real-time metric-driven scheduling improves workload stability and prevents resource saturation

Metric	Default Scheduler	Custom Scheduler
7-Minute Run – CPU Peak	Reaches 140–160% CPU between 18:35–18:40	Peaks at 85–90% CPU between 15:55–15:57
15-Minute Run – CPU Peak	Peaks between 120–160%, with fluctuations	Peaks at 95%, fluctuates 40–100%
30-Minute Run – CPU Peak	Peaks at 165–175%	Peaks at 98–100%, sustained high usage
Autoscaling Behaviour	No autoscaling across all runs	Continuous autoscaling, rapid replica creation
Workload Stability	Irregular CPU spikes, high memory plateaus	More balanced CPU distribution + efficient memory recovery
Responsiveness to Load	Slow and unresponsive	Real-time threshold-based scaling

Table 7: Cumulative Comparison of Default Scheduler vs Custom Scheduler Across All Experiments

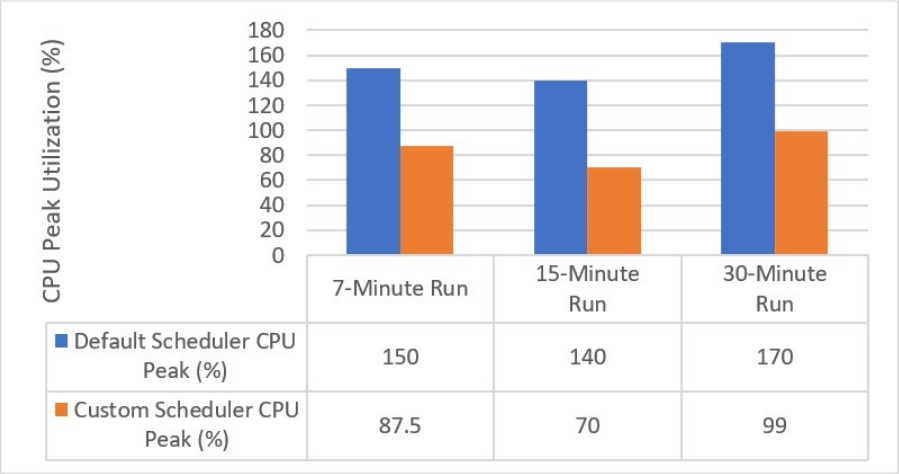

Figure 10: Cumulative CPU Peak Comparison of Default vs Custom Scheduler Across All Experiments

7 Conclusion and Future Work

This study successfully shows that a single, Prometheus-based heuristic scheduler with autoscaling can improve the performance, scalability, and resource utilization on machine learning workloads in Kubernetes settings. The scheduler can make decisions more quickly than the default Kubernetes scheduler, which uses slow updates of the Metrics Server and is based on a pre-defined policy, by using real-time metrics available via Prometheus in conjunction with a custom scoring-based node selection strategy.

The built-in autoscaling system also enhances workload responsiveness that builds more pod replicas once the CPU use surpasses the specified 60% threshold. This integrated mechanism directly and explicitly provides a response to the research question by exhibiting enhanced workload distribution, lower scaling latency, and superior resource usage of ML containers running within a Kubernetes cluster.

Nevertheless, the custom scheduler is limited, such as the lack of an advanced node-level autoscaler, no optimizations of clusters with more than one tenant, and no multi-tenant scheduling. Heavy-load edge case fault-tolerance is also an area that can use improvement.

This system can be improved in future work by adding predictive autoscaling using ML models, GPU and TPU scheduling policy, node scoring based on reinforcement learning,

integration with Cluster Autoscaler to provide node-level elasticity, and better support of distributed training systems. Further testing on more and larger heterogeneous clusters will give additional testimony to the strength and applicability of the suggested approach.

Repeated test run for statistical validation was not done as the study aimed at qualitative behavioural differences in autoscaling responsiveness and not statistical performance assurances. The test is based on a single compute-bound ML workload to isolate scheduler behaviour; the test can give different results with an I/O-bound or GPU-intensive workload.

References

Ahamed, Z., Khemakhem, M., Eassa, F., Alsolami, F. and Al-Ghamdi, A. S. A.-M. (2023). Technical Study of Deep Learning in Cloud Computing for Accurate Workload Prediction, *Electronics* **12**(3): 650.
URL: *https://www.mdpi.com/2079-9292/12/3/650*

Alharthi, S., Alshamsi, A., Alseiari, A. and Alwarafy, A. (2024). Auto-Scaling Techniques in Cloud Computing: Issues and Research Directions, *Sensors* **24**(17): 5551.
URL: *https://www.mdpi.com/1424-8220/24/17/5551*

Augustyn, D. R., Wyciślik, and Sojka, M. (2024). Tuning a Kubernetes Horizontal Pod Autoscaler for Meeting Performance and Load Demands in Cloud Deployments, *Applied Sciences* **14**(2): 646.
URL: *https://www.mdpi.com/2076-3417/14/2/646*

Carrión, C. (2023). Kubernetes Scheduling: Taxonomy, Ongoing Issues and Challenges, *ACM Computing Surveys* **55**(7): 1–37.
URL: *https://dl.acm.org/doi/10.1145/3539606*

Chouliaras, S. and Sotiriadis, S. (2023). An adaptive auto-scaling framework for cloud resource provisioning, *Future Generation Computer Systems* **148**: 173–183.
URL: *https://linkinghub.elsevier.com/retrieve/pii/S0167739X23002005*

Daradkeh, T. and Agarwal, A. (2022). Cloud Workload and Data Center Analytical Modeling and Optimization Using Deep Machine Learning, *Network* **2**(4): 643–669.
URL: *https://www.mdpi.com/2673-8732/2/4/37*

Dastagiraiah, C., Reddy, V. K., Rani, K. P. and Veeranna, T. (2024). Optimizing Load Balancing using Deep Learning in a Cloud Environment, *2024 11th International Conference on Computing for Sustainable Global Development (INDIACom)*, IEEE, New Delhi, India, pp. 1374–1380.
URL: *https://ieeexplore.ieee.org/document/10498539/*

Duay, E., Gondraneos, G. M., Indino-Pineda, K. A. and Seva, R. (2024). Design of an Interactive Scheduling Heuristic-Based Application, *in* S.-H. Sheu (ed.), *Industrial Engineering and Applications – Europe*, Vol. 507, Springer Nature Switzerland, Cham, pp. 95–106. Series Title: Lecture Notes in Business Information Processing.
URL: *https://link.springer.com/10.1007/978-3-031-58113-7*$_9$

Farid, M., Lim, H. S., Lee, C. P., Zarakovitis, C. C. and Chien, S. F. (2025). Optimizing Kubernetes with Multi-Objective Scheduling Algorithms: A 5G Perspective, *Computers* **14**(9): 390.
URL: *https://www.mdpi.com/2073-431X/14/9/390*

Gireesh Kambala (2023). Cloud-Native Architectures: A Comparative Analysis of Kubernetes and Serverless Computing, *Journal of Emerging Technologies and Innovative Research* **10**(4): n208–n233.
URL: *https://www.researchgate.net/publication/388717188*$_{C}loud - Native_{A}rchitectures_{AC}omparative_{A}nalysis_{o}f_{K}ubernetes_{a}nd_{S}erverless_{C}omputing$

Khan, T., Tian, W., Ilager, S. and Buyya, R. (2022). Workload forecasting and energy state estimation in cloud data centres: ML-centric approach, *Future Generation Computer Systems* **128**: 320–332.
URL: *https://linkinghub.elsevier.com/retrieve/pii/S0167739X21004155*

Kjorveziroski, V. and Filiposka, S. (2025). Federated architecture for serverless platforms aimed at transparent execution in the edge-cloud continuum, *International Journal of Cloud Computing* **14**(1): 115–144.
URL: *http://www.inderscience.com/link.php?id=145664*

M R, N. K., B, A., J, H., Srinivasan, S. and Sand, S. S. (2025). Pod Scheduling and Proactive Resource Management in an Edge Cluster using MCDM and Federated Learning, *Journal of Grid Computing* **23**(3): 24.
URL: *https://link.springer.com/10.1007/s10723-025-09811-8*

Nasri, S., Bouziri, H. and Mtalaa, W. (2024). Enhancing Service Quality of On-Demand Transportation Systems Using a Hybrid Approach with Customized Heuristics, *Smart Cities* **7**(4): 1551–1575.
URL: *https://www.mdpi.com/2624-6511/7/4/63*

Rejiba, Z. and Chamanara, J. (2023). Custom Scheduling in Kubernetes: A Survey on Common Problems and Solution Approaches, *ACM Computing Surveys* **55**(7): 1–37.
URL: *https://dl.acm.org/doi/10.1145/3544788*

Saxena, D., Kumar, J., Singh, A. K. and Schmid, S. (2023). Performance Analysis of Machine Learning Centered Workload Prediction Models for Cloud. Publisher: arXiv Version Number: 1.
URL: *https://arxiv.org/abs/2302.02452*

Sliwko, L. (2024). Cluster Workload Allocation: A Predictive Approach Leveraging Machine Learning Efficiency, *IEEE Access* **12**: 194091–194107.
URL: *https://ieeexplore.ieee.org/document/10807210/*

Soltani, N., Soleimani, B. and Barekatain, B. (2017). Heuristic Algorithms for Task Scheduling in Cloud Computing: A Survey, *International Journal of Computer Network and Information Security* **9**(8): 16–22.
URL: *http://www.mecs-press.org/ijcnis/ijcnis-v9-n8/v9n8-3.html*

Ugwueze, V. U. (2024). Cloud Native Application Development: Best Practices and Challenges, *International Journal of Research Publication and Reviews* **5**(12): 2399–2412.
URL: *https://ijrpr.com/uploads/V5ISSUE12/IJRPR36367.pdf*

Xu, Z., Gong, Y., Zhou, Y., Bao, Q. and Qian, W. (2024). Enhancing Kubernetes automated scheduling with deep learning and reinforcement techniques for large-scale cloud computing optimization, *in* P. Siano and W. Zhao (eds), *Ninth International Symposium on Advances in Electrical, Electronics, and Computer Engineering (ISAEECE 2024)*, SPIE, Changchun, China, p. 175.
URL: *https://www.spiedigitallibrary.org/conference-proceedings-of-spie/13291/3034052/Enhancing-Kubernetes-automated-scheduling-with-deep-learning-and-reinforcement-techniques/10.1117/12.3034052.full*

Zhang, J., Jin, C., Huang, Y., Yi, L., Ding, Y. and Guo, F. (2022). KOLE: breaking the scalability barrier for managing far edge nodes in cloud, *Proceedings of the 13th Symposium on Cloud Computing*, ACM, San Francisco California, pp. 196–209.
URL: *https://dl.acm.org/doi/10.1145/3542929.3563462*

Zhou, H. and Yong, C. H. (2024). Implement HPA for Nginx Service Using Custom Metrics Under Kubernetes Framework, *IEEE Access* **12**: 189722–189734.
URL: *https://ieeexplore.ieee.org/document/10772216/*