National College of Ireland

PID control based Cluster Autoscaling for Kubernetes

MSc Research Project
Cloud Computing

Sagar Padhi
Student ID: x21245673

School of Computing
National College of Ireland

Supervisor: Rejwanul Haque

National College of Ireland
Project Submission Sheet
School of Computing

Student Name:	Sagar Padhi
Student ID:	x21245673
Programme:	Cloud Computing
Year:	2018
Module:	MSc Research Project
Supervisor:	Rejwanul Haque
Submission Due Date:	20/12/2018
Project Title:	PID control based Cluster Autoscaling for Kubernetes
Word Count:	8295
Page Count:	22

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

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

Signature:	Sagar Swarajya Padhi.
Date:	12th December 2024

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):	

PID control based Cluster Autoscaling for Kubernetes

Sagar Padhi
x21245673

Abstract

Containerized cloud based microservices are deployed using Kubernetes framework in recent approaches in the industry. Kubernetes offers in-built autoscaling tools like the Horizontal Pod Auto-scaler and Vertical Pod Auto-scaler for provisioning resources based on demand. This research proposes a Control-based autoscaling method implementing the Proportional, Integral and Derivative control theory (PID).the PID control logic is used to address the dynamic resource allocation in an efficient approach. In this research the scaling requirement and decision is aggregated by analysing CPU utilization ,Memory Utilization and Error requests. By passing these metrics into the PID control loop to obtain a stable value for autoscaling the hardware. Kubeadm and terraform are used for deployment and the achieve horizontal and vertical scaling. This research highlights the potential of the PID Auto-scaler to achieve better scaling performance in container based microservices by achieving better cost-effectiveness , response-time and performance. In Future the PID control logic can be implemented on various other metrics together or individually on any number of metrics then calculate an aggregated mean, later the scaling output can be further refined by applying advanced mean models to obtain more stable and effective results.

1 Introduction

A key technology in modern day cloud environment is containerization, which makes lightweight virtualization possible and makes it easier to use microservices-based designs. Applications can remain scalable in dynamic workloads with the help of containers, that are controlled by orchestration platforms like Kubernetes. The com- plexity of containerized environments, makes it extremely difficult to optimize resource allocation while maintaining system stability and cost effectiveness.Scaling technologies that are commonly used are Kubernetes' Horizontal Pod Autoscaler (HPA) and Vertical Pod Autoscaler (VPA). But these native autoscalers frequently have trouble keeping up with changes in workload, which results in poor use of resources and higher operat- ing costs.To improve autoscaling capabilities, recent developments have placed a major focus on combining control theory with predictive models Garces et al. (2020). The capacity of proportional- integral-derivative (PID) controllers to dynamically modify resource allocations depend- ing on real-time measurements has made them popular among these. In order to overcome the drawbacks of current autoscalers, this research presents a PID-based autoscaling architecture for Kubernetes environments. The suggested system dynamically scales container workloads by using measures like CPU utilization and request error, reducing the waste of resources while maintaining application performance. This report's next sections offers

a examination of the PID-based autoscaling system's development, deployment, and assessment, emphasizing how it could improve resource management in containerized cloud settings.

2 Related Work

Two helpful autoscaling features that are part of Kubernetes are the Vertical Pod Autoscaler (VPA) and the Horizontal Pod Autoscaler (HPA). These solutions modify resource allocations based on variables like CPU and memory utilization to guarantee that applications scale according to workload requirements. These autoscalers work well in simple situations, but they don't measure up other metrics, including error rates, latency, or specific application requirements that are taken into consideration in Tran et al. (2022). The drawbacks of native solutions, such as HPA and VPA, have been addressed by new techniques brought about by recent developments in Kubernetes auto-scaling Senjab et al. (2023).These innovative techniques aim to improve the performance, maximize resource use, and increase scalability of containerized applications.

2.1 State-of-the-Art Techniques in Container Scaling

Recent research on Kubernetes auto-scaling have developed techniques that overcomes the limitations of HPA and VPA. These new methods aim to increase containerized applications' performance, optimize resource usage, and enhance scalability. Machine learning techniques are used by AI-based auto-scaling systems to examine workload trends and forecast resource needs.As mentioned in Xing et al. (2019) the systems proactively allocate resources by forecasting future demands, which lowers latency and enhances system responsiveness during sudden spikes in workload. Neural network and reinforcement learning-based frameworks have the potential to handle difficult scaling situations.Predictive auto-scaling frameworks use historical data and real-time metrics to estimate resource requirements with accuracy. This enables proactive scaling strategies, ensuring that workload increases may be handled by the system without over-provisioning its resources as cited in Pahl et al. (2019)

CPU and memory utilizaton measurements are basic metrics that many Kubernetes scaling techniques use.state-of-the-art techniques enables the scope by including application-specific measurements like latency, error rates, and request throughput. These custom measurements help adjust scaling decision with application execution objectives by giving a more optimized scaling strategy.To obtain scaling decisions further, some methods use feedback loops and control theory principles like PID-based control systems. These mechanisms dynamically adjust scaling parameters based on real-time system behaviour, ensuring stable and efficient scaling.

2.2 Comparative Analysis of Auto-Scaling Techniques

In containerized systems, the variety of auto-scaling techniques shows approach to optimize resource management. These techniques focus on factors like operational stability, cost reduction, resource efficiency, and responsiveness. This section explains key methods and offers a comparative evaluation of their advantages and disadvantages. The development of container-based auto-scaling methods demonstrates new methods to improve system performance and maximize utilization of resources in cloud-native environments.

Kubernetes container scaling has seen a variety of developments, from native solutions like Vertical Pod Auto-scaler (VPA) and Horizontal Pod Autoscaler (HPA) to more creative approaches combining control theory and machine learning. In Beltre et al. (2019) Simple indicators like CPU and memory usage are the foundation of native Kubernetes auto-scalers, which provide simplicity but little flexibility for complicated workloads. By combining historical usage patterns and predictive algorithms, recent approaches—like resource-aware frameworks like RUBAS—improve scaling precision.Senjab et al. (2023) research from 2023 shows that customized metrics improve user experience, help with scaling decisions, and match resource allocation to the needs of particular applications.These methods increase operational complexity by requiring complex analytics and strong real-time monitoring.

Long Short-Term Memory (LSTM) networks and other machine learning models are used in adaptive AI strategies to predict resource demands and traffic patterns. Machine learning algorithms are used in adaptive AI-based methods to forecast the demand for resources. in Xing et al. (2019)the ability of Long Short-Term Memory (LSTM) neural networks to understand sequential data, such as workload patterns, makes them frequently utilized. These methods predict traffic surges before they happen, which lowers latency and prevents over-provisioning. Research shows that AI-based scaling frameworks significantly improve cost-efficiency and reaction times.By proactively scaling resources, these strategies lower latency and avoid over-provisioning.In order to make accurate scaling decisions, predictive auto-scaling uses historical data. Frameworks based on time-series analysis models, such as ARIMA or Prophet,are able to anticipate workload patterns and distribute resources in advance.Joshi et al. (2024) Because these frameworks rely on historical patterns, they may perform poorly during unexpected traffic surges yet well in settings with predicted workload changes. Compared to standard techniques that just consider CPU or memory utilization, incorporating specific metrics such as latency, error rates, and request volumes into scaling decisions offers a more efficient approach. Like in Xing et al. (2019) latency-aware scaling frameworks prioritize ensuring optimal response times to enhance user experience. However, these methods require real-time monitoring and complex metric monitoring, which can make operations more complicated.This approach achieves up to 15% better resource utilization but introduces slight computational overheads. As noted by Senjab et al. (2023), such methods are ideal for environments with predictable workload trends but struggle with abrupt spikes .

To achieve effective resource optimization, hybrid approaches combine vertical scaling (adjusting the number of pods) with horizontal scaling (modifying resource allocations per pod).In Burroughs et al. (2021), This dual strategy provides flexibility and efficiency while managing a variety of workload patterns. While its potential, hybrid scaling requires coordination between the two scaling processes and complicated deployment.The difficulties of dynamic workloads in containerized settings are addressed by the twin strategy of combining resource optimization with rapid scaling. This approach guarantees that programs may adjust to changing demands while preserving effective resource usage by utilizing both horizontal and vertical scaling techniques. In addition to improving system speed, this well-rounded approach offers a scalable framework for microservices systems.

However, by employing time-series models to analyze past trends, Predictive Auto-Scaling inBurroughs et al. (2021) the system performs exceptionally well in settings with recurring workloads. Although it works well for consistent patterns, its unpredictable nature makes it necessary to use other strategies to manage traffic spikes.In Chang et al.

Scaling Method	Key Features	Advantages	Limitations	Best Use Cases
Adaptive AI-Based	Uses machine learning to predict resource demand	Reduces latency, proactively scales, cost-efficient	High computational cost, complexity in training	Dynamic workloads, e-commerce, gaming
Predictive Auto-Scaling	Analyzes historical data to forecast resource needs	Works well for predictable workloads, preemptive	Ineffective for highly unpredictable spikes	Batch processing, periodic tasks
Metrics-Driven Scaling	Integrates diverse custom metrics like error rates	Provides precise scaling, focuses on user experience	Complexity in implementation, requires robust monitoring	Latency-sensitive applications, APIs
Hybrid Scaling	Combines horizontal (pod count) & vertical (resource limits)	Achieves balanced scaling and resource efficiency	Implementation complexity, tuning challenges	Applications with varying load patterns
PID-Based Scaling	Uses feedback loops for dynamic adjustment	Stable response, prevents oscillations, adaptable	Requires precise tuning of PID parameters	High variability environments, unpredictable traffic
RUBAS	Enhances VPA with migration features	Efficient use of resources, minimizes container restarts	Overhead from migration operations	Dense, resource-heavy Kubernetes environments
Kubernetes Native (HPA/VPA)	Default Kubernetes auto-scalers	Easy to set up, works out-of-the-box	Limited metric options, slower response to changes	General scaling needs, low-complexity applications
Latency-Aware Scaling	Focuses on maintaining response time thresholds	Ensures high QoS, optimized user experience	Requires deep metric analysis, custom implementation	Web services, high-traffic environments

Figure 1: Comparative analysis of container based Auto-scaling techniques

(2017)by including application-specific data, like latency and error rates, into scaling decisions, metrics-driven scaling adds an extra level of accuracy. Although it requires strong and real-time monitoring capabilities, this improves the end-user experience cited in . In Shelar (2019) by combining horizontal and vertical scaling techniques, hybrid scaling strikes a balance between resource efficiency and speed. Although in Medel et al. (2018) this dual method necessitates complex implementation and coordination between scaling systems, it is especially appropriate for mixed workloads.PID-Based Scaling is ideal for workloads with varying demand since it stabilizes scaling processes and avoids oscillations thanks to feedback control techniques. However, precise parameter adjustment is important for maximizing its effectiveness. Medel et al. (2018)introduces container migration and dynamic allocation mechanisms to solve inefficiencies in Kubernetes' native VPA. Although a little resource-intensive, it is a reliable option for resource-intensive cloud environments because it increases overall utilization and lowers pod restarts.

The table 1 shows the importance of intelligent and hybrid scaling solutions that reduce the limitations of current approaches while combining their advantages. The proposal of an adaptive, PID-driven scaling model that maximizes resource consumption and improves operational stability is based on these observations.The analysis of different approaches shown in Deshpande (2021) have advantages and disadvantages of each of them. Although they provide significant computing demands, adaptive AI-based techniques perform exceptionally well in extremely dynamic contexts. While predictive frameworks function well with consistent workloads, they struggle with irregular traffic. Although it makes implementation more difficult, custom metric integration improves precision. Although stable, PID-based scaling needs to be carefully adjusted to avoid either overscaling or underscaling. It is clear from the analysis that no one approach performs better in every situation. Although it requires a lot of processing power, adaptive AI-based scaling provides better responsiveness. While predictive frameworks are stable for routine workloads, they break down when demand is erratic. Although they successfully combine strengths, hybrid techniques can be difficult to implement. Although RUBAS has small operating overheads, it shows notable gains in resource allocation efficiency.

2.3 Impact of Variables and Metrics in Scaling

Decisions on auto-scaling in containerized settings depend on accurate monitoring and analysis of metrics. Cost-effectiveness, excellent application performance, and resource efficiency are ensured by selecting the appropriate variables. This section reviews the important metrics that affect scaling choices, as well as the difficulties in choosing and applying them as mentioned in Joyce and Sebastian (2023)

The most often monitored metric for scaling choices is CPU usage. Increased workloads are frequently indicated by increased CPU consumption, which calls for more resources.According to studies, the choice of only CPU measurements may result in overscaling during temporary traffic surges, which may unnecessarily increase costs Joyce and Sebastian (2023).For instance, the Horizontal Pod Autoscaler (HPA) in Kubernetes uses CPU utilization levels to determine default scaling Balla et al. (2020).Memory utilization measurements are important for scaling memory-intensive systems, including data processing pipelines. According to Nguyen et al. (2020), improper memory scaling can lead to inefficient under-utilization of allocated resources or out-of-memory issues.For applications that interact with users, latency monitoring is important. In order to main-

tain Service Level Agreements (SLAs), longer response times indicate the requirement for more resources.For example, latency measurements are included into metrics-driven scaling strategies to guarantee quick response times during periods of high demand Jahan et al. (2020).

Throughput, defined as the rate of data processing or requests handled, is a vital metric for scaling batch processing systems and real-time streaming applications.Predictive scaling based on past workload patterns is frequently necessary to maintain consistent throughput Zhang et al. (2019) and Wei-guo et al. (2018).Beyond typical resource monitoring, custom application metrics like error rates or queue lengths offer important data.For example, in event-driven systems where asynchronous message processing drives workload surges, queue length metrics are essential (RUBAS, 2019).

Over- or under-scaling may result from choosing the incorrect metric. For example in Joseph and Chandrasekaran (2020) , focusing only on CPU use may fail to identify bottlenecks brought on by I/O activities or network limitations.Although fine-grained metrics increase the accuracy of scaling, they also demand additional resources for analysis and monitoring.high-frequency latency metric monitoring may result in additional overhead and affect system performance.In Jahan et al. (2020) decisions about scaling depend on real-time data, yet response may be decreased by processing and metric aggregation delays.Kubernetes' metrics server updates every 15 seconds, which might not adequately record sudden spikes in workload.

While numerous auto-scaling strategies have significantly improved resource allocation in Kubernetes and containerized environments, several limitations persist. This section identifies these gaps and proposes a framework that integrates the strengths of existing methods with a PID-based approach for enhanced scaling efficiency and precision.

Gaps in Current Auto-Scaling Techniques

Many techniques rely on reactive scaling, responding to metrics after thresholds are breached. This delay can lead to resource wastage or degradation of application performance during surges in workloads Aderaldo et al. (2019). The Kubernetes Horizontal Pod Autoscaler (HPA) reacts to CPU or memory spikes, which may not align with the actual application needs during latency-sensitive operations Deshpande (2021) . While most auto-scaling algorithms focus on CPU and memory utilization, they fail to incorporate critical application-specific metrics such as latency, error rates, or I/O bottlenecks. RUBAS enhanced Kubernetes Vertical Pod Autoscaler (VPA) by introducing resource migration, but still struggled to account for latency metrics Mulubagilu Nagaraj (2020) . Combining horizontal and vertical scaling provides robust solutions but remains complex to implement. Current hybrid models often lack adaptability to diverse workload patterns.The hybrid model in RUBAS showed improvements but faced challenges in balancing rapid scaling with resource optimization Balla et al. (2020) . Adaptive methods, particularly those leveraging AI/ML, often incur high computational and financial costs during training and real-time analysis. Such overheads are unsuitable for cost-sensitive applications.Example: AI-based frameworks demonstrated improved performance, but required significant resources for model training and inference Mavridis and Karatza (2017).

3 Methodology

This section explains a structured methodology to design and evaluate a PID-based autoscaling mechanism for Kubernetes based container environments.The limitations of existing scaling techniques are addressed and the research focuses on combining diffrent dynamic metric-driven scaling with theoretical and experimental approach. The methodology is supported by prior research, as mentioned in the related work, and aims to propose a novel and adaptive solution to resource management.The experiments are conducted in a controlled Kubernetes cluster environment, which is used as a testing enviroment for real-time scenarios. The infrastructure includes Docker as contianer service and Kubernetes version 1.24 with 2 node cluster deployed on a cloud platform.CPU Utilization-Captures workload intensity and acts as a primary scaling variable.Request Latency (ms) means user experience and system responsiveness.Error Rate ensures reliability and identifies bottlenecks.To ensure consistency in scaling calculations, all metrics are normalized within predefined thresholds.The PID control mechanism formed the core of the autoscaler. It calculates scaling decisions by dynamically adjusting resources based on real-time deviations from target values

Proportional Term (P)calculates as the difference between observed and target values. Integral Term (I) adds the error correction over time to address continious deviations. Derivative Term (D) Reacts to sudden workload fluctuations to stabilize the system.

$$PID\_Output = K_p \cdot e(t) + K_i \cdot \int_0^t e(\tau)d\tau + K_d \cdot \frac{de(t)}{dt}$$

$$P = K_p \cdot e(t)$$

$$I = K_i \cdot \int_0^t e(\tau)d\tau$$

$$D = K_d \cdot \frac{de(t)}{dt}$$

$$Scaling\_Action = \text{ceil}(PID\_Output)$$

Joshi et al. (2024)An important development in adaptive resource management is the integration of Proportional-Integral-Derivative (PID) control logic into Kubernetes' autoscaling algorithms. Current autoscaling techniques, such the Horizontal Pod Autoscaler (HPA), mostly use threshold-based rules, which might result in scaling decisions that are reactive and occasionally ineffective. By continuously modifying scaling operations in response to real-time system performance measurements, the PID controller, on the other hand, provides a more advanced method.

This control signal is translated into accurate changes in the number of active pods in Kubernetes autoscaling, allowing the system to react dynamically to variations in workload.PID control's use in container orchestration has been investigated in recent research. In Shelar (2019)to estimate resource use and maintain desired performance levels the PID approach combines PID control with predictive analysis using the Auto-Regressive Integrated Moving Average (ARIMA) model.Shelar (2019) Comparing this technique to conventional threshold-based techniques, it has been shown to enhance response times and CPU utilization.The ability of PID control to offer a balanced and

flexible response to fluctuating workloads is what makes its integration with Kubernetes autoscaling more effective. The PID-based method improves system stability and resource efficiency by continuously tracking performance measures and making real-time resource adjustments.Each metric is assigned PID parameters (Kp, Ki, and Kd) that are iteratively tuned to achieve a balance between scaling speed and system stability. The aggregated PID outputs determined the required number of pods for scaling up or down.The methodology includes simulation of dynamic workload patterns in form of application image replica to evaluate the adaptability of the proposed method.To achieve stability and precision in scaling decisions, the PID control loop has to include the previous error term. The procedure makes sure that the derivative component responds proportionately to the rate of change in the observable metrics by storing and using the most recent error value. By using an iterative feedback mechanism, the autoscaler may predict workload patterns and avoid oscillation or overcorrective behavior. Additionally, the loop maintains error evaluation continuity, allowing the integral term to gradually correct for persistent deviations. As a result, the PID controller keeps a responsive and balanced response, making sure that resource allocation closely matches real-time demands while reducing unnecessary scaling operations.

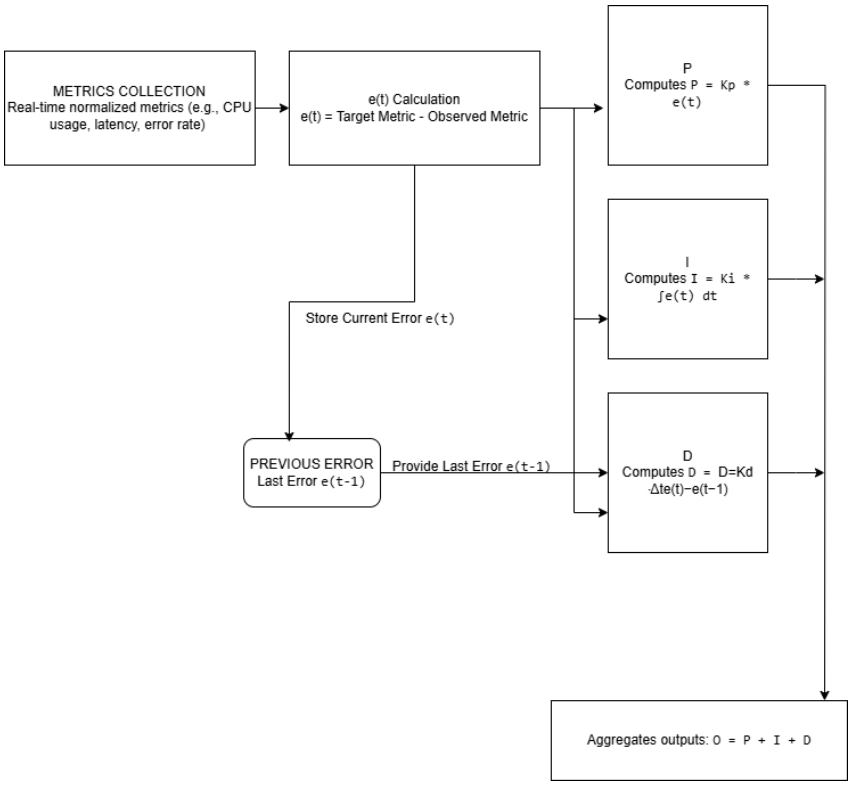

Figure 2: PID logic loop

The Auto-scaler logic computes a scaling output as shown in Fig.2 in by passing the metric value obtained into the Kp term first calculating the proportional term by comparing it with a predefined threshold value set prior. The same is done with other metrics as well but the error calculated at kp is stored to use in the next iteration as a previous error. this acts as a stabilizer for the scaling output preventing it from generating skewed results.

Static Workloads- Baseline tests with consistent workload levels to measure resource efficiency. **Dynamic Workloads-** Simulated traffic spikes and drops to assess responsiveness. **Hybrid Workloads-** Periodic and random workload changes combined to test scalability and stability under complex conditions in Burroughs et al. (2021) following that, the collected data is subjected to detailed analysis to evaluate the performance of the PID-based autoscaler. Key performance indicators (KPIs) included .Time taken for scaling decisions to be executed and reflected in cluster performance. Efficiency of resource allocation in maintaining target performance levels. Frequency of scaling oscillations, indicates stability.Statistical analysis tools were used to compare results across scenarios just like in Baresi et al. (2021), highlighting the effectiveness of the proposed approach. Several challenges are encountered during the research like sudden metric spikes impacted scaling precision. This was resolved using moving averages and smoothing techniques.Identifying optimal PID values required multiple iterations to balance stability of the Auto-Scaler logic.Scenarios with extremely low or high workloads resulted in the implementation of thresholds to prevent excessive scaling actions.The proposed methodology provides a framework for developing and validating a PID-based autoscaling mechanism in Kubernetes clusters. By using real-time metrics and adaptive control logic the goal is to optimize resource allocation, enhance responsiveness, and ensure reliability under dynamic workloads.

4 Design Specification

The design specification describes the entire architecture for implementing a custom autoscaling mechanism in Kubernetes based container environments that is based on PID. This design includes essential tools and technologies to guarantee application performance under dynamic workloads and achieve effective resource utilization and smooth execution of the auto scaling operation.

4.1 Infrastructure Setup

EC2 t2.medium instances are used as the worker nodes and control plane in the Kubernetes cluster, which is hosted on Amazon Web Services (AWS). These instances offer a balance between processing power and cost-effectiveness, their configuration of two virtual CPUs and four gigabytes of RAM these requirements are needed for kubernetes framework to run. To ensure fault tolerance and high availability, the worker nodes and control plane are placed in the same availability zones. Terraform is used to provide and manage the EC2 instances, allowing for faster infrastructure deployment and easy integration in the auto scaling logic.The collected metrics are forwarded to Prometheus, which serves as the central monitoring and metrics aggregation system. Prometheus ensures that metrics from all nodes are consistently available for analysis for this case Prometheus is setup on the same server as the control plane but this can be implemented on a separate server focused on metrics collection.Every worker node in the Kubernetes cluster has a lightweight Daemon Set installed. In order to collect performance measurements, the Daemon Set feature makes sure that a certain pod is operating on each node.The daemon set kind yaml file makes sure it gets executed on each and evry pod in the metadata collection section of the pod.Due to this feature it is possible to collect precise metrics from each pod as they are deployed.This Daemon set file then triggers an

execution call to the main PID cluster auto scaler script.The PID controller, which is implemented as a custom Python script called PID-autoscaler.py, is the central component of the design. This script computes scaling decisions by processing the real-time metrics that Prometheus has collected.

The PID-based scaling logic is core to the system's functionality. It uses proportional components to react to immediate deviations, integral terms to address cumulative errors, and derivative terms to avoid downtime in sudden workload surges or drops. The algorithm translates the PID output into a positive or negative scaling value, deciding whether additional pods should be provisioned or existing ones removed for nodes as well. By integrating real-time metrics from daemon set and control theory based logic to calculate a sacling value, the model adapts dynamically, making it suitable for unpredictable microservice workloads.Deployment management is handled using Terraform scripts, which automate the Vertical scaling actions determined by the PID logic. These scripts integrate with Kubernetes APIs to add or remove pods or nodes in response to scaling decisions. The use of Terraform ensures faster and efficient management of infrastructure changes, along with the operational requirements of modern cloud-native container based environments.

.

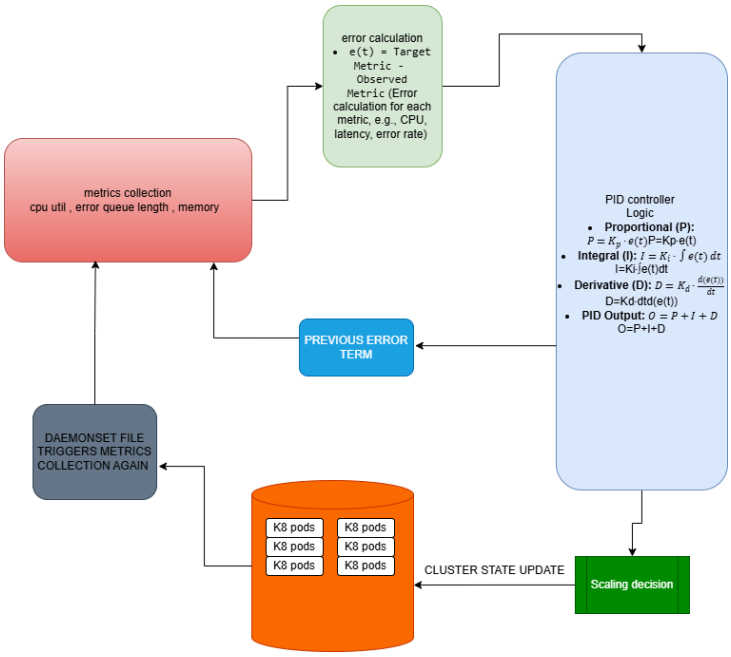

Figure 3: PID autoscaler process flow

4.2 Scaling management

The necessary number of pods is determined by the scaling value that the PID logic generates.The PID logic decides how many more pods are needed to balance the workload if the workload is 150% greater than the present capacity with a target CPU utilization of 50%.Using Kubeadm, an administrative tool for Kubernetes, horizontal scaling is carried out using the PID output. Kubeadm dynamically adds or removes pods by interacting with the Kubernetes API as shown in Fig 3 and Fig 4. This guarantees that even in the

event of unexpected spikes in traffic, application performance is maintained as set the pid logic tries to settle it according the threshold value set prior.The system uses Terraform to provision extra EC2 instances or decommission unused nodes when the workload exceeds the capacity of the current node pool. Reliability of scaling actions are guaranteed by Terraform's interface with AWS. The cluster can manage workloads that beyond the pod-level scaling capabilities enabling horizontal as well as vertical scaling. The design also prioritizes scalability at the cluster level by achieving beyond pod-level autoscaling. When pod scaling alone is insufficient to meet demands, the system integrates with Kubernetes cluster autoscalers to provision additional nodes dynamically. This hierarchical scaling capability ensures that the framework remains effective even in environments application demands exceeds the capacity of the existing node pool.The system is designed to integrate with various cloud providers and container orchestration platforms, making it versatile for deployment across diverse operational contexts.while the implementation is focused on Kubernetes, the principles and algorithms can be adapted for other platforms like Docker Swarm or Apache Mesos with minimal modifications. This flexibility ensures that the design remains relevant as containerization and orchestration technologies evolve.The design includes a feedback loop for continuous stabalizing and meeting the PID threshold value. Performance data from scaling actions are logged to further aggregate the PID parameters and decision range over time. This adaptive tuning feature ensures that the system evolves with the workload patterns, maintaining its effectiveness as application demands change.Security and fault tolerance have also been addressed in the design. The system employs role-based access control (RBAC) for its Terraform scripts and Kubernetes interactions, ensuring that scaling operations are executed securely. Additionally, redundancy mechanisms are used into the monitoring and scaling processes to handle failures.Fallback strategies used here is ,if a primary scaling action fails due to network latency or API errors.It enhances the resource management capabilities of Kubernetes and provides a compatible and secure solution for solving the challenges of modern, dynamic workloads. By implementing PID control, hybrid scaling, and continuous feedback collection , the design ensures optimal performance and resource utilization,for scalable containerized applications in cloud-native environments.

5 Implementation

The implementation of the proposed PID-based autoscaler includes a structured approach, beginning with the development of core components and culminating in the deployment and testing of the system within a Kubernetes environment.

5.1 Autoscaler work flow

The first step is creating the metrics collection pipeline. A lightweight daemon set is deployed to each Kubernetes node, enabling the continuous monitoring of metrics such as CPU utilization, memory usage, and error rates in this case. The daemon set uses Prometheus exporters to collect these metrics and store them .This data is used for calculating real-time scaling output values, ensuring the system responses fast to workload fluctuations.The PID controller logic is implemented as a Python script running in a Kubernetes pod. This script periodically fetches metrics data using Prometheus APIs and applied the PID control algorithm to calculate the scaling factor. The PID parameters (Kp, Ki, and Kd) were fine-tuned during the testing phase to ensure stability

and responsiveness under different workload conditions.These values are aggregated by testing the logic of the repeatedly and obtained a fair constant value Kp-0.1, Ki-0.01 and kd-0.05.These constants determine the weight of each component in the PID logic. The output of the PID controller determines the number of pods required to maintain target performance levels, with an additional buffer of +1 pod to handle sudden traffic spikes.For executing scaling decisions, the system integrates with Terraform to automate pod and node scaling. The PID script passes scaling actions to Terraform through a secure API endpoint, making communication better between the autoscaler and the cluster infrastructure. Terraform configurations are pre-defined to enable dynamic scaling, including the creation of new nodes when pod requirements exceeded the current cluster capacity.In this case there is no range of limit for the autoscaler yet as it is tested in a very controlled environment but a range of limits can be defined in the terraform to avoid accidental scaling and use of unused instances. Role-Based Access Control (RBAC) policies are applied to secure Terraform's access, ensuring that scaling operations adhered to cluster security protocols and only the owner with correct authentication keycan create more nodes. The PID controller enhances the system's efficiency by providing a robust decision-making mechanism for scaling as the constant check for scaling is compared with actual utilization every second and react to needed circumstances in a quicker resposne time. Unlike static or rule-based scaling approaches, the PID controller dynamically adjusts scaling actions based on real-time metrics. The Proportional (P) component reacts to the current error by determining how far the system is from the target state. The Integral (I) component accumulates past errors, addressing steady-state deviations and ensuring that the system doesn't underperform or over-utilize resources in the long term. The Derivative (D) component predicts future trends by analyzing the rate of change of the error, allowing the system to respond to rapid workload fluctuations faster.The DaemonSet and PID controller create a combination of repeated checks for needed value while considering the previous solved error. The DaemonSet ensures timely metric collection across the cluster, providing the data needed for PID logic to calcualte scaling decisions. The PID controller processes these metrics to provide precise, adaptive scaling values as no. of pods +1. This combination results in an autoscaling mechanism that is efficient, responsive and resource-conscious, effectively managing dynamic workloads in real-time container based environments.

The workflow and architecture of a PID-based custom autoscaler in a Kubernetes container based enviroment is shown in the diagram Fig 4. Through the combination of Kubernetes and Terraform, it explains how metrics get collected, processed, and used to calculate the scaling decisions and no. of pods to be allocated or de-allocated, while providing effective resource management.The pods are the main components which contain containerized applications image files in the worker node segment. The application workloads are managed by these pods, and their resource consumption is regularly monitored by the metadata pod. Every worker node in the cluster will have a particular metadata collecting pod installed due to the Daemon Set file deployed. Key performance parameters like CPU utilization, request delay, and error rates are collected by this metadata collection pod. After being prepared as JSON objects, these measurements are transmitted to the control plane for processing.The control plane will contain the main PID-Autoscaler python file which will be called and executed by daemon set file, in short the pid autoscaler script is executed repeatedly as soon as there is any diffrence between the pre-defined scaling threshold value and current value.Additionally, every worker node has a kubelet, a Kubernetes component in charge of pod management. The kubelet re-

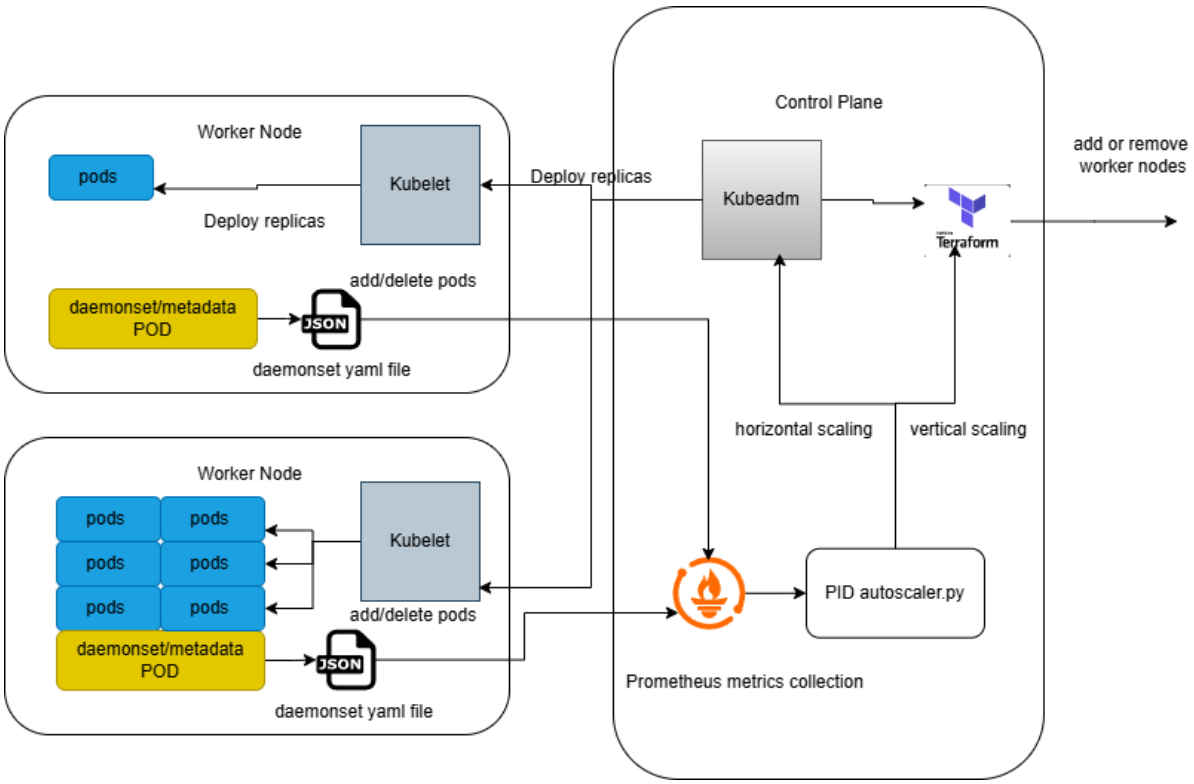

Figure 4: PID Autoscaler Architecture diagram

ports the status of the pods that the autoscaler has deployed to the control plane and makes sure they are operating as planned. Decisions about scalability are made at the control plane, which is the main component of the design. Every worker node's performance metrics are gathered by Prometheus, which then aggregates them for analysis. The PID-based autoscaler script then receives these metrics. PID control terms are used by the autoscaler script to determine scaling choices. The derivative component reduces sudden spikes in workload, the integral component takes into account continuous deviations over time, and the proportional component responds to current deviations from goal values. Whether scaling actions, such adding or removing pods, are necessary is decided by the PID logic.Scaling choices at the pod level are implemented by Kubeadm in the control plane. When scaling operations include node additions or deletions, Terraform is used to manage the infrastructure. By ensuring that worker nodes are provided or decommissioned as necessary. Terraform helps to match the workload demands with the cluster's capacity.The DaemonSet starts the workflow by gathering worker node metrics. Prometheus receives these measurements, combines them, and sends them to the PID autoscaler. After analyzing the metrics, the autoscaler determines whether to scale up or down. Kubeadm modifies the amount of pods deployed for horizontal scaling. Terraform adds or removes nodes for node-level or vertical scaling. This system runs constantly, tracking metrics and processing them in a feedback loop to ensure the cluster adapts dynamically to changing workloads.

The final phase of implementation involves testing the autoscaler in a controlled production environment. variety of workloads with varying patterns in form of deploying replica sets of the demo application written in Go languge are generated to evaluate the

system's responsiveness, stability, and efficiency. Key performance indicators such as scaling latency(delay), resource utilization(extra unsed pods), and error rates (amount of errors generated due to insufficient pods) are collected to use it in the autoscaler's calculation. Feedback from these tests shows further refinements to the PID parameters and scaling logic. The implementation of the PID-based autoscaler is a multi-staged process involving metric collection, control logic development, automation of scaling actions, and repeated checks for stability. The system is designed to integrate with Kubernetes, providing a flexible and adaptive solution for resource management in containerized environments. By using both horizontal and vertical scaling strategies, the implementation ensures optimal resource utilization and high application performance across dynamic workload generated in various scenarios.The architectural flow begins with the metrics collection stage, where a Kubernetes DaemonSet is deployed across the cluster to gather real-time data. This DaemonSet collects critical performance metrics such as CPU utilization, memory usage, error rates, and request latencies from all nodes within the cluster. The collected metrics are then parsed to Prometheus, a monitoring and alerting toolkit that acts as the central repository for storing and managing these metrics. Prometheus enables efficient querying and analysis of the data, providing the basis for subsequent decisions and can be used in future to collect advance application specific rquirements to achive more accurate scaling results.Once the metrics are stored in Prometheus as objects, they are fetched by a custom PID controller implemented in a Python file which contains the PID logic algorithm. This controller processes the metrics to determine deviations between the current and desired threshold values and tries to reach to the target value set initially. Using proportional, integral, and derivative calculations, the PID controller generates an output value that denotes the scaling requirement a positive or negative number denoting the need of scalling up or scaling down the system. The controller evaluates whether the current workload necessitates scaling up, scaling down, or maintaining the existing resource allocation.The output from the PID controller is then fed into Terraform scripts , which automate the necessary scaling operations. These scripts interact with the underlying infrastructure to modify the Kubernetes cluster configuration. Depending on the scaling requirement, the scripts adjust either the number of pods in the cluster or the resource allocation of individual containers.The Kubernetes cluster, equipped with the new configuration, proceeds to implement horizontal scaling by increasing or decreasing the number of pods based on workload demands. This shows that the system can effectively handle variations in requests for service while maintaining performance and efficiency. Also, vertical scaling adjusts the resource limits of individual containers, optimizing the utilization of CPU and memory resources within the existing pods.The target can be changed accordingly to maximize the hardware use.

5.2 Daemon set feature

The Daemon Set plays a critical role in this architecture by ensuring that metrics are collected consistently and accurately across all nodes within the Kubernetes cluster. Since a Daemon Set runs a copy of a specified pod on each node, it provides a uniform mechanism to gather key performance metrics such as CPU utilization, memory usage, error rates, and request latencies and other metrics directly from the source. This decentralized and synchronized data collection minimizes latency in retrieving metrics and ensures that the system has an up-to-date understanding of cluster-wide resource utilization.the Daemon Set simplifies the deployment of monitoring tools. Instead of manually configuring mon-

itoring tools on each node, the Daemon Set ensures that the agents are automatically deployed and maintained, reducing administrative overhead. This automation is especially valuable in dynamic cloud environments, where nodes may frequently scale up or down.

6 Evaluation of PID Autoscaler

This section evaluates the performance, effectiveness, and flexibility of the suggested PID-based autoscaling mechanism in a different case scenarios. The primary finding of this research is that adding PID control logic to Kubernetes autoscaling improves system responsiveness and resource usage under changing workloads. In order to compare the suggested approach with other autoscaling strategies such as Horizontal Pod Autoscaler (HPA), Vertical Pod Autoscaler (VPA), and other advanced frameworks, the experiments have been carefully developed to replicate real-world application scenarios, including both controlled and unpredictable traffic patterns.This section aims to address important research goals, such as stability under varying workloads, resource allocation efficiency, and adaptability to hybrid scaling requirements, by utilizing a combination of theoretical insights and experimental setups. Because the experiments are based on strict testing procedures and controlled scaling actions, the outcomes are guaranteed to be both practically and statistically significant. This section describes a thorough summary of the advantages and disadvantages of the PID-based strategy using specific performance measures including CPU utilization, request latency, and scaling speed.Real-time metric collecting and smooth execution of scaling actions are made possible by tools like Terraform and Prometheus which increase the response time of the operation in total. To make it easier to understand how the system behaves in various situations, the findings are displayed using tables and then compared in graphs in discussion. This section concludes by highlighting the research's useful applications, providing useful data for both research and industry practices in cloud containerized systems.

6.1 Evaluating PID-Based Autoscaler with Static Workloads against the HPA/VPA

In this experiment, a controlled static workload scenario is generated by the demo application replica sets to compare the PID-based autoscaler to the Kubernetes Horizontal Pod Autoscaler (HPA) and Vertical Pod Autoscaler (VPA). In order to demonstrate the results of the PID-based strategy in a static workload enviroment, the important metrics evaluated are CPU utilization, the number of scheduled pods, and system response time. Version 1.24 of Kubernetes is installed on the Kubernetes cluster used in this experiment. The workload is hosted on a single node with a target CPU usage threshold of 50% and an initial pod count of one. The workload is static as the amount of cpu usage and memory usage can be locked in the replica set deployment file, ensuring no fluctuation in demand to test the autoscalers' behavior under static conditions.This involves creating replica sets, starting with one pod, and observing how both autoscalers respond by adding or removing pods as necessary. Additionally, the experiment does node-level scaling to determine how efficiently resources are allocated and deallocated.

Time (Minutes)	PID Autoscaler CPU Util (%)	HPA/VPA CPU Util (%)	Pods Scheduled (PID)	Pods Scheduled (HPA/VPA)
2	40	40	1	2
5	45	45	2	3
10	48	48	2	4
15	50	50	3	5

Figure 5: Results obtained to record amount of resources allocated for a static workload

6.2 Response to Sudden Workload Spike

This experiment compares the PID-based autoscaler's response time and resource efficiency to that of the HPA/VPA in the case of a sudden spike in workload.the spike is generated by using a different script to generate multiple traffic request. The workload is steady at 40% CPU usage at the start of the experiment. The spike is simulated by deploying replica sets at different frequency, which results in a 120% increase in CPU utilization. The threshold CPU utilization for both autoscalers is set at 50%. One pod manages a consistent workload at 40% CPU usage at the start of the experiment. Real-world traffic surges are replicated by sequentially deploying replica sets with increasing request frequencies in order to replicate a sudden workload spike. In order to assess the PID-based autoscaler's scaling efficiency in comparison to HPA/VPA, metrics like the number of pods added and stabilization time are tracked.

Increase in Workload (Initial % → Spike %)	Pods Added/Removed (PID)	Time Taken (PID)	Pods Added/Removed (HPA/VPA)	Time Taken (HPA/VPA)
40% → 120%	+2	20 seconds	+3	40 seconds
120% → 60%	-1	25 seconds	-2	45 seconds
60% → 30%	-1	15 seconds	-1	30 seconds
30% → 100%	+3	22 seconds	+4	50 seconds
100% → 40%	-2	18 seconds	-3	35 seconds

Figure 6: Results obtained to record response time in scaling operation

6.3 Calculating Resource Wastage

This experiment compares the efficiency of HPA/VPA with the PID-based autoscaler under various workloads in order evaluate over-provisioning pods. A consistent workload of 30% CPU usage controlled by a single pod starts the test. Increasing over time, the CPU utilization peaks at 80% and then drops out at 50%. At each interval, metrics are recorded, including the number of pods scheduled and the amount of resource waste.The HPA/VPA adds three pods, leaving one extra pod, whereas the PID system schedules two pods when the workload exceeds 60% at the 5-minute mark. Measurable resource waste results from this pattern's repeating at other intervals.

Time (Minutes)	CPU Utilization (%)	Pods Scheduled (PID)	Pods Scheduled (HPA/VPA)	Excess Pods (HPA/VPA)	Resource Wastage (HPA/VPA)
2	30	1	1	0	0%
5	60	2	3	1	~33%
10	80	3	4	1	~25%
15	50	2	3	1	~20%

Figure 7: Results obtained after deploying irregular workload

6.4 Discussion

The experiments provide a detailed comparison between the PID-based autoscaler and HPA/VPA methods in Kubernetes environments. By analyzing metrics such as CPU utilization, time taken for scaling, resource efficiency, and pods scheduled, the evaluation highlights the superiority of the PID-based approach in dynamic scaling scenarios.In a static workload scenario, the PID-based autoscaler demonstrated efficient scaling with minimal resource wastage. The experiment began with a consistent workload at 40% CPU utilization, gradually increasing to 50% over 15 minutes. During this time, the PID-based autoscaler scheduled pods incrementally as needed, avoiding over-provisioning. In contrast, the HPA/VPA scheduled additional unnecessary pods, leading to increased resource wastage. For example, at the 15-minute mark with 50% CPU utilization, the PID-based method scheduled three pods compared to five pods by HPA/VPA, reducing excess allocation by 40%.

The PID-based system added two pods in 20 seconds when the workload increased from 40% to 120%, whereas HPA/VPA added three pods in 40 seconds, demonstrating the PID's superior resource use and quicker responsiveness.The PID-based method deleted one pod in 25 seconds during a workload reduction from 120% to 60%, but HPA/VPA removed two pods in 45 seconds, indicating slower and less effective scaling on the part of HPA/VPA.The PID controller's adaptive nature made it possible for it to effectively stabilize the system, preserving peak performance while reducing excessive resource use. These findings are consistent with studies showing threshold-based scaling techniques, such as HPA/VPA, are not very effective in managing sudden workload fluctuations.The examination of resource waste provided important information on how effective both

approaches are. The number of extra pods planned by HPA/VPA in comparison to the ideal number needed was used to calculate resource waste. HPA/VPA planned three pods at 5 minutes with 60% CPU consumption, whereas PID scheduled two pods, wasting 33% of the available resources. HPA/VPA scheduled three pods at 15 minutes with 50% CPU utilization, but PID scheduled two pods, resulting in a 20% waste of resources. These results show that the PID-based approach can reduce operating expenses and improve resource efficiency while maintaining appropriate scaling without over-provisioning.

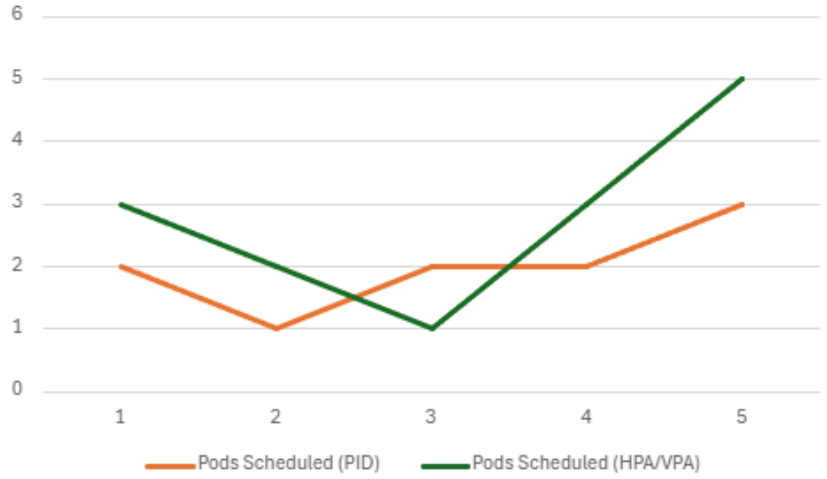

Figure 8: Comparison of Pods Scheduled Over Time: PID-based Autoscaler vs. HPA/VPA

The graph in Fig 8 shows how many pods are planned over time for both HPA/VPA and the PID-based autoscaler. First, over-provisioning occurs because the HPA/VPA routinely plans more pods than the PID-based autoscaler. The PID-based autoscaler continues to scale more steadily and effectively over time, but the HPA/VPA shows a more pronounced increase in pods, which may indicate resource waste. This illustrates how effectively the PID technique reduces needless scaling activities.

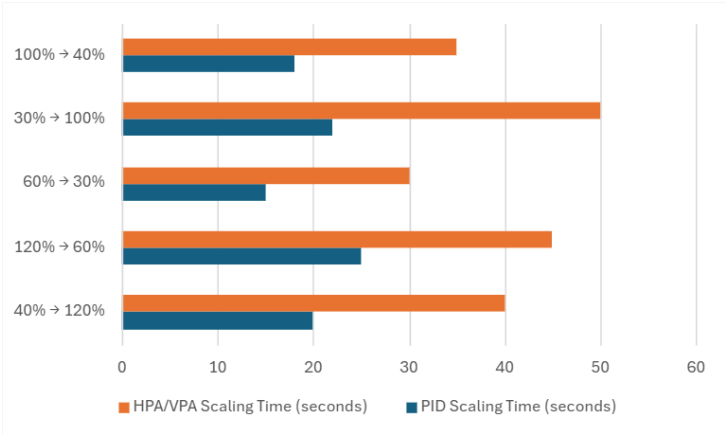

Figure 9: Comparison of Scaling Response Times: PID-based Autoscaler vs. HPA/VPA under Varying Workload Changes

The bar chart in Fig 9 compares the scaling reaction times of HPA/VPA with the PID-based autoscaler under different workload fluctuations. It demonstrates that the PID-based autoscaler continuously responds to changes in workload more quickly and scales more quickly in all situations. The HPA/VPA, on the other hand, takes a lot longer, particularly when there are major increases or decreases in workload. This demonstrates how well the PID-based strategy responds to changing workload needs and cuts down on decision-making delays related to scaling.

The comparative costs across various time periods are shown in this bar graph Fig 10.

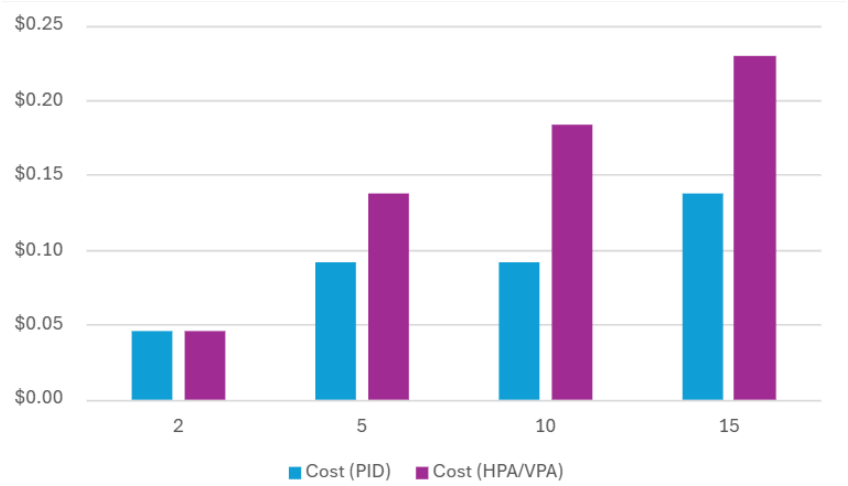

Figure 10: Cost Comparison of PID-Based Autoscaler vs. HPA/VPA Over Time

Because of its effective resource allocation, the PID-based autoscaler continuously shows lower costs when the costs are computed based on resource utilization over time. When the workload stabilizes at two minutes, the expenses of both systems are almost equal. The PID-based autoscaler is still more cost-effective, but as the workload increases (at 5, 10, and 15 minutes), the HPA/VPA costs increase greatly because of the over-provisioning of resources.

7 Conclusion and Future Work

This study proposes and evaluates a PID-based autoscaling framework in order to overcome the problems of the default Kubernetes Horizontal Pod Autoscaling (HPA) and Vertical Pod Autoscaling (VPA) methods. Reducing overprovisioning and underprovisioning by constant target checks and increasing resource allocation efficiency also ensuring cost-effectiveness without sacrificing application performance were the main goals. The effectiveness of the PID-based method in accomplishing these objectives was confirmed by a series of controlled experiments that compared it to similar autoscalersexecuting a dynamic workload scenario.The key findings show that the PID-based autoscaler is more resource-efficient and responds to workload variations more quickly. Especially when the demand is dynamic, it continuously avoids the resource waste seen in HPA/VPA. The PID-based system scaled up and down quickly and precisely in situations of unexpected workload spikes, eliminating unnecessary pods and preserving target utilization. According to the cost study, this action not only decreased operating expenses but also guaran-

teed a steady performance level, which is essential for preserving Quality of Service (QoS).

There are some limitations to the research. It can be difficult to balance responsiveness and stability in highly unpredictable conditions since the PID tuning process is sensitive and requires iterative calibration. Also, since CPU utilization acted as the main scaling indicator in this study, future research might investigate the incorporation of other metrics, including disk I/O and network bandwidth, to develop a more comprehensive scaling mechanism appropriate for a range of cloud microservice workloads.The PID-based model is economically feasible for cloud service providers looking to optimize resource management since it can be easily integrated into current Kubernetes cloud environments. It is a strong option for handling modern cloud-native apps because of its capacity to flexibly adjust to changes in workload.Automating PID parameter tweaking with machine learning methods to increase system flexibility is one possible research topic. Also, this framework's applicability might be expanded by integrating it into memory-intensive microservices or stateful applications. The effectiveness of scaling decisions could be further improved by investigating hybrid strategies that combine PID control with predictive scaling based on previous patterns.

In conclusion, this study has successfully demonstrated how a PID-based autoscaler may be used to successfully overcome the drawbacks of conventional autoscaling techniques. It creates the foundation for creative developments in Kubernetes resource management by finding a balance between performance, responsiveness, and affordability. It also creates new opportunities for investigation and findings of new approches in cloud-native enviroments.

References

Aderaldo, C. M., Mendonça, N. C., Schmerl, B. and Garlan, D. (2019). Kubow: an architecture-based self-adaptation service for cloud native applications, *Proceedings of the 13th European Conference on Software Architecture - Volume 2*, ECSA '19, Association for Computing Machinery, New York, NY, USA, p. 42–45.
URL: *https://doi.org/10.1145/3344948.3344963*

Balla, D., Simon, C. and Maliosz, M. (2020). Adaptive scaling of kubernetes pods, *NOMS 2020 - 2020 IEEE/IFIP Network Operations and Management Symposium*, pp. 1–5.

Baresi, L., Hu, D. Y. X., Quattrocchi, G. and Terracciano, L. (2021). Kosmos: Vertical and horizontal resource autoscaling for kubernetes, *in* H. Hacid, O. Kao, M. Mecella, N. Moha and H.-y. Paik (eds), *Service-Oriented Computing*, Springer International Publishing, Cham, pp. 821–829.

Beltre, A., Saha, P. and Govindaraju, M. (2019). Kubesphere: An approach to multi-tenant fair scheduling for kubernetes clusters, *2019 IEEE Cloud Summit*, pp. 14–20.
URL: *https://ieeexplore.ieee.org/document/9045748*

Burroughs, S., Dickel, H., van Zijl, M., Podolskiy, V., Gerndt, M., Malik, R. and Patros, P. (2021). Towards autoscaling with guarantees on kubernetes clusters, *2021 IEEE*

International Conference on Autonomic Computing and Self-Organizing Systems Companion (ACSOS-C), pp. 295–296.

Chang, C.-C., Yang, S.-R., Yeh, E.-H., Lin, P. and Jeng, J.-Y. (2017). A kubernetes-based monitoring platform for dynamic cloud resource provisioning, *GLOBECOM 2017 - 2017 IEEE Global Communications Conference*, pp. 1–6.

Deshpande, N. (2021). *Autoscaling cloud-native applications using custom controller of kubernetes*, Master's thesis, Dublin, National College of Ireland. Submitted.
URL: *https://norma.ncirl.ie/5089/*

Garces, L., Martinez-Fernandez, S., Graciano Neto, V. V. and Nakagawa, E. Y. (2020). Architectural solutions for self-adaptive systems, *Computer* **53**(12): 47–59.

Jahan, S., Riley, I., Walter, C., Gamble, R. F., Pasco, M., McKinley, P. K. and Cheng, B. H. (2020). Mape-k/mape-sac: An interaction framework for adaptive systems with security assurance cases, *Future Generation Computer Systems* **109**: 197–209.
URL: *https://www.sciencedirect.com/science/article/pii/S0167739X19320527*

Joseph, C. T. and Chandrasekaran, K. (2020). Intma: Dynamic interaction-aware resource allocation for containerized microservices in cloud environments, *Journal of Systems Architecture* **111**: 101785.
URL: *https://www.sciencedirect.com/science/article/pii/S1383762120300758*

Joshi, N. S., Raghuwanshi, R., Agarwal, Y. M., Annappa, B. and Sachin, D. (2024). Arima-pid: container auto scaling based on predictive analysis and control theory, *Multimedia Tools and Applications* **83**(9): 26369–26386.
URL: *https://doi.org/10.1007/s11042-023-16587-0*

Joyce, J. E. and Sebastian, S. (2023). Enhancing kubernetes auto-scaling: Leveraging metrics for improved workload performance, *2023 Global Conference on Information Technologies and Communications (GCITC)*, pp. 1–7.

Mavridis, I. and Karatza, H. (2017). Performance and overhead study of containers running on top of virtual machines, *2017 IEEE 19th Conference on Business Informatics (CBI)*, Vol. 02, pp. 32–38.
URL: *https://ieeexplore.ieee.org/document/8012937*

Medel, V., Tolosana-Calasanz, R., Ángel Bañares, J., Arronategui, U. and Rana, O. F. (2018). Characterising resource management performance in kubernetes, *Computers Electrical Engineering* **68**: 286–297.
URL: *https://www.sciencedirect.com/science/article/pii/S0045790617315240*

Mulubagilu Nagaraj, A. (2020). *Optimizing kubernetes performance by handling resource contention with custom scheduler*, Master's thesis, Dublin, National College of Ireland. Submitted.
URL: *https://norma.ncirl.ie/4543/*

Pahl, C., Brogi, A., Soldani, J. and Jamshidi, P. (2019). Cloud container technologies: A state-of-the-art review, *IEEE Transactions on Cloud Computing* **7**(3): 677–692.
URL: *https://ieeexplore.ieee.org/document/7922500*

Senjab, K., Abbas, S., Ahmed, N. and ur Rehman Khan, A. (2023). A survey of kubernetes scheduling algorithms, *Journal of Cloud Computing* **12**(1): 87.
URL: *https://doi.org/10.1186/s13677-023-00471-1*

Shelar, P. L. (2019). *Dynamic resources allocation using priority aware scheduling in kubernetes*, Master's thesis, Dublin, National College of Ireland. Submitted.
URL: *https://norma.ncirl.ie/4137/*

Tran, M.-N., Vu, D.-D. and Kim, Y. (2022). A survey of autoscaling in kubernetes, *2022 Thirteenth International Conference on Ubiquitous and Future Networks (ICUFN)*, pp. 263–265.
URL: *https://ieeexplore.ieee.org/document/9829572*

Wei-guo, Z., Xi-lin, M. and Jin-zhong, Z. (2018). Research on kubernetes' resource scheduling scheme, *Proceedings of the 8th International Conference on Communication and Network Security*, ICCNS '18, Association for Computing Machinery, New York, NY, USA, p. 144–148.
URL: *https://doi.org/10.1145/3290480.3290507*

Xing, S., Qian, S., Cheng, B., Cao, J., Xue, G., Yu, J., Zhu, Y. and Li, M. (2019). A qos-oriented scheduling and autoscaling framework for deep learning, *2019 International Joint Conference on Neural Networks (IJCNN)*, pp. 1–8.
URL: *https://ieeexplore.ieee.org/document/8852319*

Zhang, F., Tang, X., Li, X., Khan, S. U. and Li, Z. (2019). Quantifying cloud elasticity with container-based autoscaling, *Future Generation Computer Systems* **98**: 672–681.
URL: *https://www.sciencedirect.com/science/article/pii/S0167739X18307842*