CHAPTER I

INTRODUCTION

1.1 Research Background

Due to rapid technological advances, cloud computing has become an essential part of the computing landscape in the past ten years. It gives companies access to flexible and scalable computing resources and services (Malallah et al., 2023; *What Is Cloud Computing? | Microsoft Azure.*, n.d.). This is especially true with the rise of lightweight container solutions like Docker (Modak et al., 2018), which are often used for fast and consistent deployment of applications, such as web servers. However, real-world service workloads fluctuate over time and across different scenarios, leading to constantly changing resource demands.

To effectively manage large-scale container deployments, container orchestration tools are essential. Kubernetes, one of the most widely used orchestration tools today, provides not only basic container orchestration and management but also advanced features such as load balancing, auto-scaling, and auto-recovery (Gao et al., 2017; *Kubernetes Documentation.*, n.d.). These capabilities help companies maintain high service availability by dynamically adjusting to traffic patterns during business peaks and down.

As container-based cloud services become more prevalent, companies increasingly rely on automatic scaling mechanisms instead of manual ones. Kubernetes supports this through the Horizontal Pod Autoscaler (HPA), which automatically adjusts the number of pods based on real-time workload conditions (Šimon et al., 2023). In Kubernetes, a pod is the smallest deployable unit, typically

containing one or more containers, and it serves as the fundamental unit for deployment, scheduling, and scaling (Tonini et al., 2023).

Kubernetes offers three types of autoscaling mechanisms, including Horizontal Pod Autoscaler (HPA), Vertical Pod Autoscaler (VPA), and Cluster Autoscaler (CA). By default, HPA uses resource metrics, primarily CPU and memory usage to determine when to scale (*Kubernetes Documentation.*, n.d.). However, in certain scenarios, these metrics fail to reflect the actual operational load of a service.

A key example is observed in I/O-bound applications, such as web services using Nginx as a reverse proxy. In modern microservice architectures, these proxy services frequently depend on external downstream systems, such as third-party APIs or payment gateways, introducing unavoidable processing delays. During these waits, the proxy must hold concurrent connections open. Even when the pod's CPU utilization appears stable and low, the service may be overwhelmed due to system-level constraints, such as reaching the maximum number of file handles or active connections handled by worker processes. Once these thresholds are reached, Nginx may be unable to accept new connections, leading to timeouts or HTTP 5xx status codes. This behavior highlights a critical limitation: CPU metrics are insufficient indicators for service degradation caused by I/O-bound connection bottlenecks.

To address this problem, HPA can be extended to utilize custom application-level metrics such as Requests Per Second (RPS). Unlike CPU or memory metrics, which react to internal resource consumption, RPS provides a proactive signal of incoming workload intensity. When a sudden traffic surge

occurs, the RPS value can immediately spike, triggering the HPA to scale up before system-level constraints such as file handle exhaustion or worker overload take effect. This enables faster and more accurate scaling responses that better protect service performance.

Although HPA is widely used in Kubernetes environments, most research to date has focused on CPU/memory-based scaling (Ahmad et al., 2024; Nilsen, 2023) or other application-level metrics (Camilo & Campos, 2024; Zhou & Yong, 2024). Recent studies, specifically Zhou & Yong (2024), successfully highlighted this exact I/O bottleneck dilemma in Nginx reverse proxies, demonstrating that custom metrics like HTTP 5xx errors outperform default CPU metrics during connection exhaustion. However, their proposed solution remains inherently reactive, the system relies on user requests failing before initiating recovery. Very few studies have specifically explored the use of RPS as a proactive scaling metric that anticipates connection saturation before I/O bottlenecks cause errors.

To investigate this problem, this research will employ a two-tier system architecture designed to isolate and analyze the impact of metric selection on I/O-bound HPA behavior. The first tier consists of a frontend Nginx service configured as a reverse proxy, which will be the target of the HPA configuration. The second layer is an upstream backend service that introduces an artificial 500ms processing delay for every forwarded request, simulating the latency of external third-party integrations. This delay causes the frontend Nginx pods to hold concurrent connections for longer periods, eventually exhausting worker_connections limits as traffic increases. This setup deliberately induces a performance bottleneck at the connection level, rather than CPU or memory, creating a scenario in which resource

metrics are decoupled from actual service performance degradation. Additionally, the system is intentionally designed without complex business logic, allowing the results to reflect the pure impact of autoscaling metrics on I/O bottlenecks.

To systematically evaluate HPA behavior under each configuration, this study uses four targeted load testing scenarios: ramp-up load, sustained high-load, cyclic load, and gradual decreasing load. The ramp-up scenario evaluates autoscaling responsiveness during progressively increasing traffic, while the sustained high-load scenario examines service stability under continuous pressure. The cyclic load scenario evaluates the ability of the autoscaler to respond to repeated traffic fluctuations, and the gradual decreasing load scenario observes scale-down behavior under declining workload conditions. In addition to the main comparison, a post-comparison optimization phase is conducted for the RPS-based HPA configuration to improve autoscaling responsiveness under sudden and repeated high-load conditions.

In addition to comparing CPU-based and RPS-based HPA configurations, this study also includes a post-comparison optimization phase for the RPS-based HPA configuration. This phase is designed to evaluate whether tuning the autoscaling pipeline, including image pull policy, Prometheus scrape interval, and HPA scale-up behavior, can improve autoscaling responsiveness under high-intensity traffic conditions. The optimization is applied only to the RPS-based HPA configuration because RPS is the main custom metric investigated in this study.

1.2 Research Identification

Based on the background described above, this research faces technical challenges such as:

1. Configuration and Tuning of HPA Metrics

   This study involves configuring the HPA for two fundamentally different scaling metrics: standard CPU usage and custom metrics, specifically RPS. Implementing RPS-based scaling introduces added complexity, requiring the integration of Prometheus and a Prometheus Adapter to expose application-specific metrics.

2. Design and Execution of Load Testing Scenarios

   Designing and implementing diverse load testing scenarios (including ramp-up and sustain high-load) using k6 introduces technical difficulties. These include scripting, managing test durations, ensuring reproducibility, and incorporating warm-up phases to ensure stable and valid measurements.

3. Metric Collection and Data Aggregation

   Accurate, synchronized, and reliable metric collection is crucial. Key metrics such as the number of active pods, average response time (latency), and error rates must be collected from Prometheus during dynamic workloads to enable meaningful analysis.

1.3 Research Questions

Based on the problems mentioned above, the research questions that can be formulated are as follows:

1. How do CPU-based and RPS-based HPA configurations compare in terms of CPU usage, pod count, average response time (latency), and HTTP 5xx error rate when subjected to different traffic patterns such as ramp-up, sustained high load, cyclic load, and gradual decrease load?

2. Is RPS-based HPA more effective in maintaining service performance, particularly during connection bottlenecks in web services like Nginx
3. How does the optimization of the RPS-based HPA configuration, including image pull policy, Prometheus scrape interval, and scale-up behavior, affect error rate, latency, CPU usage, and pod replica count under sustained high-load and cyclic load scenarios?

1.4 Problem Limitations

To maintain focus and feasibility, this study is limited to the following aspects:

1. The study will only compare HPA configurations based on CPU utilization and RPS as the scaling metric. Other autoscaling strategies such as VPA, KEDA, or event-driven mechanisms will not be considered.
2. The experiments were conducted using a single-node Kubernetes cluster through Minikube deployed on a Google Cloud Platform e2-standard-2 virtual machine. This controlled environment was selected to isolate the effect of autoscaling metric selection and to maintain consistency across repeated experiments. However, all application pods, monitoring components, and Kubernetes control processes operated within the same node. Therefore, the experiment did not represent several characteristics of production-grade multi-node Kubernetes environments, including inter-node communication, cross-node pod scheduling, external load balancer behavior, node-level resource contention, and node failure conditions. Consequently, the quantitative results of this study, including scaling time, pod replica count, latency, and error rate, should be interpreted within the single-node experimental context and should

not be directly generalized to production-scale Kubernetes deployments without further validation.

3. The evaluation focuses solely on core performance metrics such as CPU usage, number of pods, average response time, and error rate. Other factors such as cost, energy efficiency, or long-term system stability are beyond the scope of this study.
4. The observation window for each test scenario is limited to short durations (10 minutes per traffic pattern). The study does not include long-term performance behavior such as diurnal traffic fluctuations, weekly load cycles, or post-autoscaling recovery behavior.
5. The optimization phase is limited to the RPS-based HPA configuration and is evaluated only under sustained high-load and cyclic load scenarios. The optimization does not modify the application architecture, backend delay, or scaling metric. It only adjusts image pull policy, Prometheus scrape interval, and HPA scale-up behavior.

1.5 Research Objective

Based on the research problems presented, the objectives of this study are as follows:

1. To measure and compare the performance of CPU-based and RPS-based HPA configurations, in terms CPU usage, of pod count, average response time (latency), and HTTP 5xx error rate, under varying traffic loads such as ramp-up sustained high load, cyclic load, and gradual decrease load.

2. To evaluate the effectiveness of RPS-based autoscaling in maintaining service performance, especially in connection-limited conditions, compared to default CPU-based scaling.
3. To evaluate the impact of optimizing the RPS-based HPA configuration, including image pull policy, Prometheus scrape interval, and scale-up behavior, on error rate, latency, CPU usage, and pod replica count under sustained high-load and cyclic load scenarios.

1.6 Research Significances

This study is expected to provide several benefits:

1.6.1 Theoretical Benefits

This study will contribute to a better understanding of HPA behavior by systematically evaluating its performance using two distinct metric configurations under various simulated workload patterns. It addresses a notable gap in research comparing CPU/memory-based and RPS-based autoscaling in Kubernetes.

1.6.2 Practical Benefits

By identifying which HPA setup is more effective for specific workload types, this research provides practical insights for engineers and system administrators looking to optimize resource utilization and application responsiveness in containerized environments.