Adaptive Autoscaling in Kubernetes: A Study of HPA, VPA, and Hybrid Models

Team 5: Yilin Huai, Junhong Chen, Xiangyu Liu
{yilin.huai, jhong.chen, swift.liu}@mail.utoronto.ca

Abstract

With the growing adoption of cloud computing and containerization, optimizing resource allocation to handle fluctuating demands has become a critical challenge. Kubernetes, a popular container orchestration tool, is designed to automate the scaling, resource allocation and management of workloads in dynamic environments. This project evaluates the performance of Horizontal Pod Autoscaling (HPA), Vertical Pod Autoscaling (VPA), and Hybrid Pod Autoscaling in a simulated microservices environment, focusing on response time and throughput. By analyzing these metrics under continuous traffic spikes, we found that HPA and VPA outperform both the baseline and Hybrid approach, demonstrating their strengths in maintaining high performance and low response times. These findings offer useful insights into the trade-offs of Kubernetes autoscaling strategies, supporting future evaluation in cloud-native environments.

1 Background

Kubernetes has become a leading solution for managing containerized applications in cloud environments. This section introduces key concepts of Kubernetes and containerization, laying the groundwork for understanding its autoscaling features and how they enhance performance and resource management in microservices.

1.1 Monolithic vs. Microservices Architecture

The shift from monolithic applications to microservices has transformed application development and deployment. In a monolithic architecture, all components are tightly coupled and run as a single service, meaning that if one part of the application experiences increased demand, the entire system needs to be scaled [1]. This architecture often leads to high costs, increased complexity, and difficulties in scaling, especially as the codebase grows. Moreover, making updates or adding features becomes challenging, as changes require redeploying the entire application, which can lead to longer release cycles and increased risk of downtime.

In contrast, microservices architecture breaks down applications into smaller, independent services that communicate over well-defined APIs. Each microservice handles a specific function and can be scaled, deployed, or updated independently. For instance, an online shopping application can be divided into services like account management, product catalog, shopping cart, and order processing. This flexibility improves resource efficiency, reduces costs, and enables faster development cycles, as only the required services are adjusted. Microservices are particularly beneficial in handling demand spikes since individual services can be scaled without affecting the entire application. However, this approach introduces complexities with numerous components and inter-dependencies, which potentially causes dependency conflicts. Containerization is commonly used to address these challenges by providing isolated environments, ensuring consistent deployment and easier management of dependencies.

1.2 Containerisation

As shown in Figure 1, containerization is a virtualization technique that packages applications and their dependencies into isolated containers, allowing them to share the host operating system while running independently [2]. Unlike traditional virtualization, where each application requires a separate virtual machine (VM) with its own OS, containerization is more resource-efficient. Containers utilize fewer resources as they do not require full OS installations, making them particularly effective for smaller applications with lower hardware overhead.

The rise of containerization was significantly boosted by the introduction of Docker in 2013 [2], which simplified the process of building, deploying, and running containers. However, as applications and deployments have grown in complexity, managing containers at scale necessitates robust orchestration tools. This is where a container orchestration tool like Kubernetes comes into play.

1.3 Kubernetes

Kubernetes is a powerful open-source platform originally developed by Google to automate the

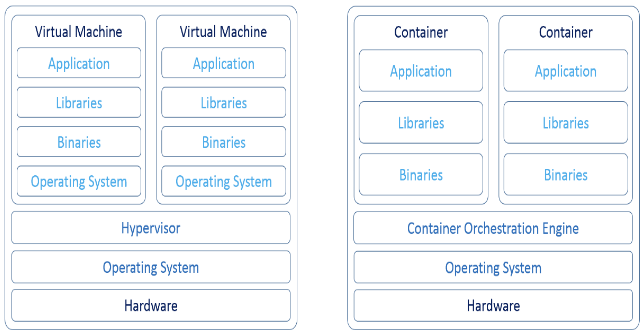

Figure 1: Virtual machine vs. containers [3]

deployment, scaling, and management of containerized applications [4]. As organizations increasingly adopt microservices and containerized environments, Kubernetes has become the de facto standard for orchestrating containerized workloads. The core architecture of Kubernetes revolves around several key components, as illustrated in Figure 2, which includes pods, nodes, services, and various controllers.

1.3.1 Components in Kubernetes

At the core of Kubernetes, the master node orchestrates the cluster's operations through several key components. The kube-apiserver acts as the central communication hub, exposing the Kubernetes API and coordinating requests across the cluster. The etcd database serves as a reliable, high-availability key-value store, where maintains the cluster's configuration and state. The kube-scheduler assigns newly created pods to appropriate nodes based on resource availability, affinity rules, and other constraints. The controllers, managed by the kube-controller-manager and cloud-controller-manager, are responsible for maintaining the cluster's desired state, handling node lifecycles, job execution, load balancing, and other critical tasks [5]. Together, these components ensure efficient resource allocation, high availability, and consistent cluster management.

Worker nodes, on the other hand, are responsible for running the application workloads. Each worker node includes a Kubelet, which acts as an agent to ensure that containers are running as expected by communicating with the master node and managing the lifecycle of containers. Within worker nodes, applications run inside pods, which are the smallest deployment units in Kubernetes. Pods serve as an abstraction layer for containers by grouping one or more containers that share resources and operate together. Additionally, the Container Runtime (commonly Docker) is responsible for pulling container images and running the containers. The Network Proxy (Kube-proxy) manages networking within the cluster, enabling seamless communication between pods, even if they are distributed across different nodes [5].

1.3.2 Horizontal Pod Autoscaling (HPA)

The Horizontal Pod Autoscaler in Kubernetes is a mechanism that adjusts the number of pod replicas in a deployment based on real-time metrics such as CPU or memory usage. This scaling mechanism is part of the Kubernetes control plane and works by periodically checking resource usage against predefined thresholds. The HPA calculates the desired number of replicas with the formula [6]:

$$desiredReplicas = currentReplicas \times \frac{currentMetricValue}{desiredMetricValue}$$

where:

- $currentReplicas$ is the current number of pod replicas,
- $currentMetricValue$ is the actual resource usage of the pods (e.g., CPU, memory, or custom metrics),
- $desiredMetricValue$ is the target resource usage defined in the HPA configuration.

For example, if the current number of replicas is 3, the current metric value is 75 millicores (mCPU), and the desired metric value is 50mCPU, the HPA calculates:

$$desiredReplicas = 3 \times \frac{75}{50} = 3 \times 1.5 = 4.5$$

Since the number of replicas must be a whole number, Kubernetes rounds up to 5 replicas.

This formula allows the HPA to dynamically adjust the number of replicas to maintain optimal resource utilization. The HPA also supports custom metrics, enabling scaling based on application-specific metrics such as request rate or error count, which offers greater flexibility and control over scaling behavior.

1.3.3 Vertical Pod Autoscaling (VPA)

The Vertical Pod Autoscaler adjusts the resource limits (CPU and memory) allocated to each pod based on historical usage, instead of changing the number of replicas [7]. The VPA includes three main components: the Recommender, the Updater, and admission webhooks. These components collaborate to assess a pod's past resource usage, create recommendations for resource allocation, and adjust the workload's configuration accordingly [8].

The VPA defines target resource limits based on specific percentiles:

- The 90th percentile is used for the target bound,
- The 50th and 95th percentiles are used for the lower and upper bounds respectively.

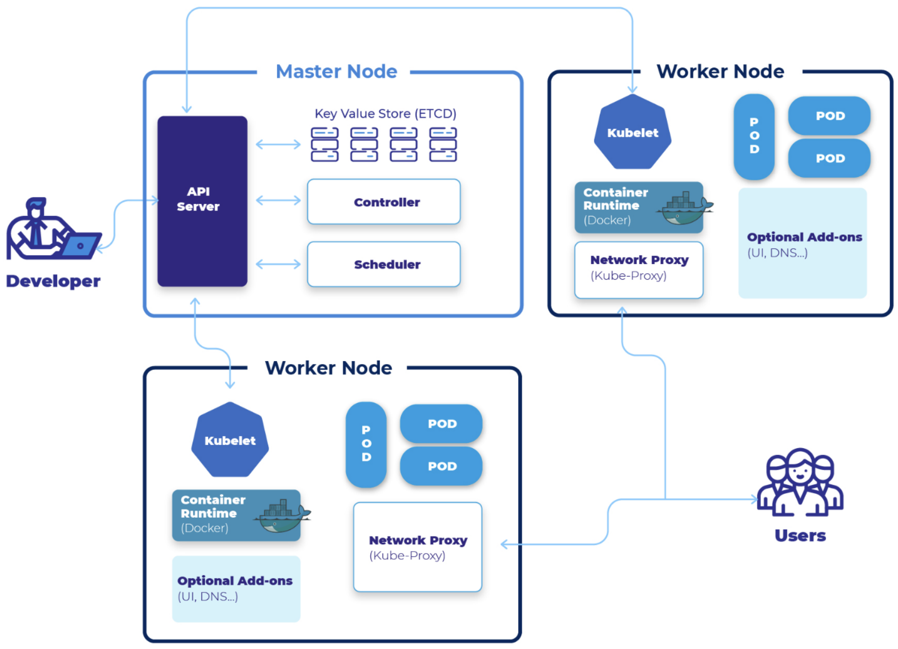
Master Node
Key Value Store (ETCD)
API Server
Controller
Scheduler
Developer
Worker Node
Kubelet
POD
POD
POD
Container Runtime (Docker)
Network Proxy (Kube-Proxy)
Optional Add-ons (UI, DNS...)
Worker Node
Kubelet
POD
POD
POD
Container Runtime (Docker)
Optional Add-ons (UI, DNS...)
Network Proxy (Kube-Proxy)
Users

Figure 2: Basic Architecture of a Kubernetes Cluster [4]

For resource recommendations, the confidence multiplier is applied to widen or narrow the resource bounds based on historical data. The confidence multiplier for the upper bound is calculated as [9]:

$$estimation = estimation \times \left(1 + \frac{1}{historyLengthInDays}\right)$$

and for the lower bound as:

$$estimation = estimation \times \left(1 + \frac{0.001}{(historyLengthInDays)^2}\right)$$

For example, assuming an initial CPU estimation of 500mCPU and a historical data length of 10 days:

The upper bound is calculated as:

$$estimation = 500 \times \left(1 + \frac{1}{10}\right) = 550\, mCPU$$

The lower bound is calculated as:

$$estimation = 500 \times \left(1 + \frac{0.001}{10^2}\right) = 500.05\, mCPU$$

So in this example, VPA defines the CPU allocation per pod to range between 500.05mCPU (lower bound) and 550mCPU (upper bound), ensuring that pods have sufficient resources while minimizing waste.

1.3.4 Hybrid Pod Autoscaling

The Hybrid Pod Autoscaling approach, also known as Multidimensional Autoscaling, combines both horizontal and vertical scaling. It first applies vertical scaling (VPA) to adjust the resource limits for each pod according to the current workload, and then horizontal scaling (HPA) to increase or decrease the number of pod replicas based on demand. This cascading approach optimizes both resource allocation per pod and the overall number of pods to handle load variations effectively.

In this model, VPA provides the resource baseline for each pod, which HPA then uses as the basis for scaling out or in. While this approach enhances flexibility by balancing workload and resource use, it may lead to underutilization during sudden traffic changes, as newly added pods might initially have lower resource allocation until VPA recalibrates them.

2 Research Methodology

Our approach involved systematically evaluating Kubernetes autoscaling strategies through a well-structured process. We started by creating an application capable of generating sufficient CPU and memory consumption to effectively trigger autoscaling. The Kubernetes cluster was then configured with appropriate resource allocations per pod to balance efficiency and avoid underprovisioning or overprovisioning.

We designed experiments to compare baseline performance with configurations for HPA, VPA, and a hybrid approach. A consistent workload was applied across all models to ensure fairness, and performance metrics such as response times and throughput were collected during the tests. The data was analyzed to draw conclusions about the effectiveness of each autoscaling method. Finally, limitations were examined to identify potential areas for improvement.

3 Implementation

This section provides an overview of the implementation process, detailing the configuration of the Kubernetes cluster, the development and deployment of a Spring Boot application for workload generation, the containerization and deployment pipeline, and the load testing methodology. The goal is to illustrate how the system was set up and tested to evaluate Kubernetes' autoscaling mechanisms effectively.

3.1 Kubernetes Cluster Setup

To analyze the auto-scaling capabilities of Kubernetes, a Kubernetes cluster was deployed on Google Cloud Platform with the following specifications:

- **Kubernetes Version:** 1.30.5-gke.1699000
- **Node Configuration:** A single node was employed, characterized by:
  - **Machine Type:** e2-standard-4
  - **Virtual CPUs (vCPUs):** 4 (equivalent to 2 physical cores)
  - **Memory:** 16 GB
- **Operating System:** Container-optimized OS
- **Boot Disk:** 20 GB SSD

The decision to configure the Kubernetes cluster with only one node was made to simplify the evaluation of pod auto-scaling mechanisms by isolating the scaling behavior within a single-node environment.

3.2 Spring Boot Application Setup

To facilitate the evaluation of Kubernetes' auto-scaling capabilities, a Spring Boot application was developed and deployed as the primary workload for the cluster. The application was designed to simulate compute-intensive operations, enabling an accurate analysis of resource scaling under stress.

The application performs two key tasks to generate substantial resource demands. First, it executes complex matrix multiplication operations, which require significant memory and computational power. Second, it calculates prime numbers using iterative and computationally intensive algorithms, further stressing the CPU.

These tasks were selected to create a workload that places considerable pressure on both CPU and memory resources, ensuring that the cluster's auto-scaling mechanisms are actively triggered and evaluated effectively.

3.3 Containerization and Deployment to GKE

The first step in preparing the application for deployment involved containerizing the Spring Boot application using Docker. Containerization ensures that the application can run consistently across diverse environments by packaging the application code, dependencies, and runtime environment into a lightweight, portable container [10].

The process began with building a Docker image. A Dockerfile was created to define the build instructions for the Spring Boot application. The Dockerfile specified the application's base image, dependencies, and required configurations, ensuring an optimized and reproducible image build process. Using this Dockerfile, the image was built locally.

Next, the built Docker image was tagged and securely uploaded to Google Container Registry (GCR) for storage. Following this, a Kubernetes deployment manifest was created to define the configuration required for deploying the containerized application as a pod within the Kubernetes cluster. Finally, to make the application accessible, a Kubernetes Service was created and configured, exposing the application to external or internal traffic as needed.

3.4 Load Generation and Testing

3.4.1 Load Testing Configuration

The workload simulations in the test were conducted using Hey, a modern load testing tool written in Go. Hey was selected for its simplicity, efficiency, and ability to generate significant HTTP load, making it suitable for stress testing the deployed application [11].

For the load testing process, the following parameters were configured. Ten workers were used to run concurrently, simulating multiple users accessing the application simultaneously. Each worker issued a maximum of 100 queries per second (QPS), creating a sustained and significant load on the application to stress the cluster's resources effectively. The tests were conducted over three different durations—3 minutes, 5 minutes, and 8 minutes—to observe how the cluster responded to varying sustained loads. Additionally, the keep-alive feature was disabled, ensuring that new HTTP requests were not bound to existing connections. This allowed requests to be distributed to newly generated pods instead of continuing to route to old pods.

3.4.2 Metrics Collected

During the testing process, several performance metrics were monitored and recorded to evaluate the application and cluster performance. The response time metrics included:

- **Slowest Response:** The maximum time taken to receive a response from the server.

- **Fastest Response:** The minimum time taken to receive a response from the server.
- **Average Response Time:** The average time taken to receive a response from the server.
- **50th Percentile:** 50% of all responses were received within this time threshold.
- **95th Percentile:** 95% of all responses were received within this time threshold.
- **Responses/Second:** The overall throughput, indicating the number of responses successfully received per second.

4 Experimentation and Results Evaluation

In this section, we evaluate three Kubernetes autoscaling mechanisms: HPA, VPA, and a hybrid approach combining both methods. We first establish a baseline using a single pod configuration, then analyze each autoscaling strategy's performance under consistent load conditions. Our evaluation examines key metrics including response times, throughput, and latency percentiles across different test durations to understand the effectiveness of each approach.

4.1 Baseline

To evaluate the performance and efficiency of Kubernetes autoscaling mechanisms, we established a baseline configuration that reflects the behavior of a single pod without any autoscaling. This baseline serves as a reference point for comparing the results of implementing HPA, VPA, and a hybrid autoscaling approach. The CPU request and limit were both set to 400 mCPU, ensuring consistent CPU availability under baseline conditions. Similarly, the memory request and limit were configured to 256 MB, providing adequate memory resources for handling the workload without overprovisioning.

The single pod is responsible for processing all requests, providing a controlled environment to observe the system's behavior when scaling mechanisms are not applied. To simulate realistic load conditions, we configured our load testing environment with 10 concurrent users, each sending requests at a rate of 100 requests per second. The performance metrics are shown in Table 1 for three test durations: 3 minutes, 5 minutes, and 8 minutes.

Since our test starts with one pre-warmed pod deployed and it is already stable, the three test durations show consistent results across all performance indicators, including response time measurements (slowest, fastest, average), latency percentiles (50th and 95th), and throughput (responses per second). The system maintains a steady throughput of approximately 3.7 responses per second and about 3.4 seconds 95th percentile response time, with minimal variation across different test durations.

Table 1: Baseline Performance: 1 pod with 400mCPU, handling 100 requests/second and 10 users concurrently.

Test Duration	**3 min**	**5 min**	**8 min**
Slowest (s)	5.2790	4.8631	5.2382
Fastest (s)	1.5687	1.6764	1.5088
Average (s)	2.7506	2.6858	2.6628
50% Resp. (s)	2.6829	2.6619	2.6272
95% Resp. (s)	3.3674	3.4080	3.3596
Responses/sec	3.6256	3.7185	3.7538

4.2 HPA

For HPA, we configured the deployment with a minimum of 1 replica and a maximum of 5 replicas. The configuration maintained the same resource allocation per pod as our baseline (400mCPU and 256MB memory) but implemented CPU-based autoscaling with an 80% utilization threshold. This means scaling is triggered when a pod's CPU usage reaches 80% of 400mCPU, equivalent to 320mCPU. The HPA controller was configured to execute scaling actions immediately without any wait period to ensure responsive scaling behavior.

Under the same load testing conditions, the performance metrics of the HPA implementation are presented in Table 2. Throughout the test durations, the HPA mechanism demonstrated dynamic scaling behavior, stabilizing at 3 pods during the 3-minute test and 4 pods during both the 5-minute and 8-minute tests. While the system achieved higher throughput, processing between 6.46 and 8.62 responses per second due to multiple pods handling requests, there were some performance trade-offs. The 95th percentile response times increased to between 2.88 and 3.86 seconds, and notably, the slowest response times more than doubled compared to the baseline, ranging from 11.74 to 13.59 seconds.

This increase in the slowest response times can be attributed to the inherent delays associated with the application startup process when new pods are deployed. When the HPA scales up the deployment to handle increased load, the newly added pods require some time to initialize, which includes setting up the application environment and loading the necessary resources. During this initialization period, incoming requests routed to these new pods experience longer response times, contributing to the observed spike in the slowest response metrics.

Table 2: Horizontal Autoscaling Performance: Up to 5 pods with 400mCPU each, handling 100 requests/second and 10 users concurrently. The deployment scaled to 3 pods during the 3-minute test and 4 pods during both the 5-minute and 8-minute tests.

Test Duration	3 min	5 min	8 min
Slowest (s)	11.7756	13.5852	11.7427
Fastest (s)	0.2114	0.2136	0.1941
Average (s)	1.9540	1.6063	1.2825
50% Resp. (s)	1.6773	1.2414	1.0233
95% Resp. (s)	3.8676	3.5659	2.8861
Responses/sec	6.4586	7.5031	8.6183

4.3 VPA

For the VPA experimentation, we utilized VPA's ability to analyze and automatically adjust resource requirements based on historical usage patterns. Based on the resource utilization observed over the previous four hours, the VPA in the Kubernetes system recommended allocating 290mCPU per pod, which is lower than the baseline configuration of 400mCPU. To ensure a fair comparison with other autoscaling models, we maintained a fixed deployment of 3 pods throughout the experiment (in contrast to HPA, which scaled between 1 and 5 pods), and each pod was configured with the VPA-recommended 290mCPU.

Under the same load testing conditions, the performance metrics of the VPA implementation are presented in Table 3. The system achieved higher throughput compared to the baseline, processing around 7 responses per second for the 5-minute and 8-minute runs, while using less CPU resources per pod. The 50th percentile response times also showed improvement, ranging from 1.25 to 1.40 seconds. While the 3-minute test showed an anomaly with an 11.89-second slowest response time, the 5-minute and 8-minute tests demonstrated slowest response times around 5 seconds, comparable to the baseline performance.

Table 3: Vertical Autoscaling Performance: 3 pods with 290mCPU each, handling 100 requests/second and 10 users concurrently.

Test Duration	3 min	5 min	8 min
Slowest (s)	11.8930	5.1907	4.9946
Fastest (s)	0.2972	0.2279	0.2232
Average (s)	1.7318	1.4605	1.4031
50% Resp. (s)	1.4003	1.2503	1.2494
95% Resp. (s)	3.8951	3.1519	3.0615
Responses/sec	5.7518	6.8444	7.1182

4.4 Hybrid Autoscaling

To perform the hybrid autoscaling approach, we combined both Horizontal and Vertical Pod Autoscaling mechanisms. The configuration maintained the HPA settings with a minimum of 1 replica and a maximum of 5 replicas while incorporating VPA's recommended CPU allocation of 290mCPU per pod, as determined from the previous VPA analysis.

Under the same load testing conditions with 10 concurrent workers each sending 100 requests per second, the performance metrics of the hybrid implementation are presented in Table 4. The system stabilized at 3 pods across all test durations, similar to the VPA implementation. Notably, all performance metrics showed consistent improvement as the test duration increased. The throughput increased from 4.03 to 5.31 responses per second, and the 50th percentile response times improved from 3.59 to 1.67 seconds. The 95th percentile response times also showed improvement, decreasing from 5.64 to 4.53 seconds across the test durations. However, the system exhibited notably higher slowest response times, ranging from 17.11 to 19.54 seconds, which were significantly greater than both the baseline and the other two scaling approaches. Additionally, its 50% and 95% response times were less favorable compared to the baseline and other scaling strategies. While the hybrid method achieved better throughput (responses per second) than the baseline, it lagged behind both HPA and VPA in this metric. Overall, the hybrid method's performance fell short of expectations across most evaluated metrics.

The degradation of the hybrid approach's slowest response times, as well as its 50% and 95% response times, is attributed to the same application startup delay discussed in Section 4.2. Additionally, the lower throughput can be attributed to its reduced CPU allocation of 290mCPU per pod, compared to the 400mCPU in the initial HPA setup. This resource constraint particularly impacts the system during scaling operations, as Spring Boot, being a heavyweight web framework, requires significant resources during application startup. While both HPA and hybrid approaches face pod initialization overhead during scaling, the hybrid approach is more severely impacted due to its lower CPU allocation. This effect is evidenced by the gradual improvement in throughput as test duration increases (from 4.03 to 5.31 responses/second), indicating that once pods overcome the initial startup phase, the system can achieve better steady-state performance.

Table 4: Hybrid Autoscaling Performance: Up to 5 pods with 290mCPU each, handling 100 requests/second and 10 users concurrently. The deployment consistently scaled to 3 pods across the 3-minute, 5-minute, and 8-minute tests.

Test Duration	**3 min**	**5 min**	**8 min**
Slowest (s)	19.5407	17.3314	17.1102
Fastest (s)	0.2529	0.2476	0.2287
Average (s)	3.6088	2.6006	2.0623
50% Resp. (s)	3.5939	2.2348	1.6682
95% Resp. (s)	5.6402	4.7917	4.5341
Responses/sec	4.0281	4.9108	5.3120

4.5 Overall Evaluation

From the throughput analysis shown in Figure 3, all three autoscaling approaches demonstrated higher responses per second compared to the baseline configuration. HPA showed the most significant improvement in throughput, increasing from 6.46 to 8.62 responses per second as the test duration extended. This superior performance can be attributed to its ability to scale up to 4 pods in the longer duration tests. VPA also showed substantial improvement, achieving between 5.75 and 7.11 responses per second, despite using less CPU per pod than the baseline configuration.

However, the hybrid approach, while still outperforming the baseline, showed the lowest throughput improvement among the three autoscaling strategies. It processed between 4.03 and 5.31 responses per second, approximately 40% lower than HPA's peak performance. Although the hybrid approach improved steadily with longer test durations, it consistently remained below both individual HPA and VPA implementations in terms of throughput capacity.

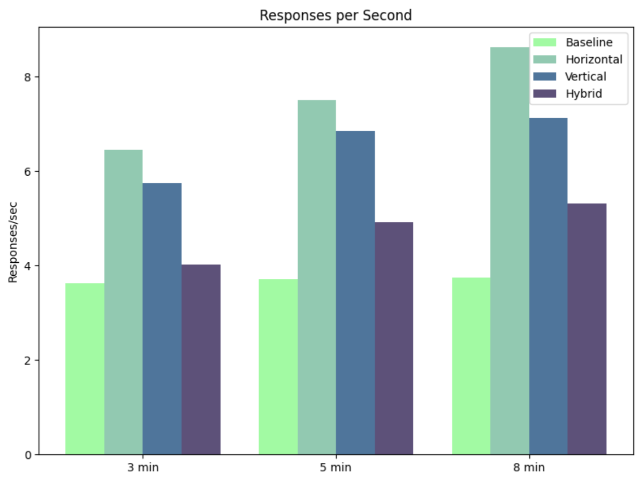

Figure 3: Comparison of responses per second across different autoscaling strategies and test durations.

In terms of latency performance, the 50th percentile response times in Figure 4 reveal distinct patterns across different autoscaling approaches. The baseline configuration maintained consistent performance with 50th percentile response times around 2.65 seconds across all test durations. Both HPA and VPA demonstrated notable improvements over the baseline. HPA's 50th percentile response times improved from 1.67 to 1.02 seconds across test durations, while VPA achieved similar improvements with response times between 1.40 and 1.25 seconds. The hybrid approach showed more varied performance, starting with higher response times at 3.59 seconds in the 3-minute test but improving significantly to 1.67 seconds in the 8-minute test.

The 95th percentile response times in Figure 4, which indicate worst-case performance, showed the baseline maintaining stable performance at approximately 3.4 seconds. HPA and VPA again showed comparable performance to each other, with times ranging from 3.87 to 2.89 seconds for HPA and 3.90 to 3.06 seconds for VPA. However, the hybrid approach exhibited the highest 95th percentile response times among all approaches, ranging from 5.64 to 4.53 seconds.

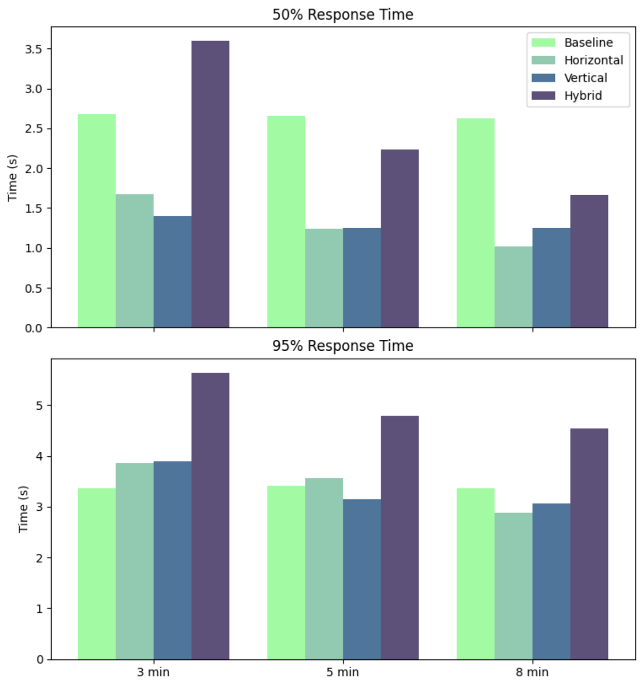

Figure 4: Comparison of 50th and 95th percentile response times across different autoscaling strategies and test durations.

5 Conclusion

In our continuous traffic spikes pattern, both HPA and VPA demonstrated excellent performance across all evaluated metrics, including throughput, 50% response time, and 95% response time. These results highlight their effectiveness in dynamically adapting to high-demand scenarios. In contrast, the hybrid autoscaling approach showed mixed results, outperforming the baseline only in terms of throughput but underperforming in 50% and 95% response times. This indicates that while the hybrid method can improve overall throughput, it struggles with latency under sustained load. Overall, Kubernetes proved to be an effective autoscaling system, showcasing its ability to handle high-demand situations efficiently and maintain application performance during continuous traffic spikes.

6 Future Work

This research can be extended in several directions to provide more comprehensive insights into Kubernetes autoscaling mechanisms.

Real-world Application Traffic Patterns: While our study used consistent load patterns for controlled evaluation, future work should focus on testing with real-world traffic patterns. This would provide more practical insights into how autoscaling mechanisms perform under variable and unpredictable workloads that better reflect production environments.

Extended Historical Data Collection: Our current VPA implementation based its recommendations on only 4 hours of historical data. Future work should extend this data collection period to days or weeks, allowing VPA to capture more comprehensive resource usage patterns and make more accurate recommendations based on long-term application behavior.

Extended Test Duration: Our performance evaluation was limited to relatively short test runs (3, 5, and 8 minutes). Future work should conduct longer test durations to minimize the impact of initial pod startup times, especially for hybrid autoscaling. Longer runs would provide better insights into the steady-state performance and efficiency of different autoscaling approaches once the initial overhead is amortized.

Custom Autoscaling Metrics: Future implementations could explore custom metrics for triggering autoscaling decisions. These could include request queue length, application error rates, database connection pool utilization, transaction processing times, and memory heap usage patterns. These metrics could provide more sophisticated and application-specific scaling triggers compared to the basic CPU-based scaling used in our study.

Alternative Application Frameworks: Our study used Spring Boot, which is relatively heavyweight and impacts startup times during scaling operations. Testing with lighter-weight frameworks, such as Golang's Gin, could provide insights into how application stack choices affect autoscaling behavior and overall system performance.

Cluster Autoscaling: Due to budget limitations, our current work focused on pod-level scaling within a single node. Future research could examine Cluster Autoscaling, which automatically adjusts the number of worker nodes within a cluster based on resource demands. This would provide valuable insights into the interaction between pod-level and node-level scaling, resource optimization across multiple nodes, and cost-performance trade-offs in cloud environments.

References

[1] *Microservices-AWS*, Amazon Web Services, Available: https://aws.amazon.com/microservices/#:~:text=Microservices%20are%20an%20architectural%20and,small%2C%20self%2Dcontained%20teams

[2] *Kubernetes vs Docker: Understanding the Difference*, Josh Campbell, Atlassian, Available: https://www.atlassian.com/microservices/microservices-architecture/kubernetes-vs-docker

[3] Alibaba Cloud, *Difference Between Container and Virtual Machine*, Available at: https://www.alibabacloud.com/en/knowledge/difference-between-container-and-virtual-machine?_p_lc=1.

[4] *Kubernetes Architecture Diagram: A Complete Overview*, Nikola Pantic, Available: https://www.clickittech.com/devops/kubernetes-architecture-diagram/

[5] Kubernetes Documentation, *Kubernetes Architecture*, Available at: https://kubernetes.io/docs/concepts/architecture/.

[6] *Horizontal Pod Autoscale - Kubernetes Documentation*, Kubernetes, Available: https://kubernetes.io/docs/tasks/run-application/horizontal-pod-autoscale/

[7] David Balla, Csaba Simon, and Markosz Maliosz, *Adaptive Scaling of Kubernetes Pods*, High Speed Networks Laboratory, Budapest University of Technology and Economics, NOMS 2020 - IEEE/IFIP Network Operations and Management Symposium, 2020.

[8] *Vertical Pod Autoscaler*, StormForge, Available: https://stormforge.io/kubernetes-autoscaling/vertical-pod-autoscaler/, Accessed: 2024-11-08.

[9] Povilas Versockas, *Vertical Pod Autoscaling: The Definitive Guide*, Povilas, Available: https://povilasv.me/vertical-pod-autoscaling-the-definitive-guide/

[10] Docker, *Docker Build Concepts: Overview*, Available: https://docs.docker.com/build/concepts/overview/.

[11] Ahmet Soormally, *rakyll/hey Load Testing — enhanced with HDR, Medium*, Apr. 16, 2020. Available: https://medium.com/@asoorm/rakyll-hey-load-testing-extended-with-hdr-1500aa262411.