Authors: Jingwen Huang, Yuxin Wang, Lan Wang
Course: CS6650 - Building Scalable Distributed Systems

Scalable Image Processing Platform with Kubernetes

Abstract

This paper presents a scalable, fault-tolerant image processing platform built using Kubernetes and Docker. The system handles concurrent image uploads, processes them in parallel using Java-based microservices, and stores results in an S3-compatible local bucket powered by LocalStack. We demonstrate how container orchestration and autoscaling enable high throughput, resilience, and elasticity in a cloud-native image pipeline.

1. Introduction

Modern applications—such as e-commerce platforms, social media networks, and content delivery services—are increasingly required to process vast numbers of images in real time. These image processing tasks may include resizing, compression, format conversion, or the application of filters and visual effects. As user bases grow and traffic patterns become more unpredictable, traditional monolithic systems struggle to scale efficiently under variable loads, resulting in latency and resource underutilization.

To address these limitations, we built a Kubernetes-based distributed image processing platform that uses containerized microservices, dynamic autoscaling, and local S3-compatible storage for efficient and resilient images handling. The system supports real-time processing and adapts to load fluctuations without manual intervention.

2. Problem Statement

Modern systems often suffer from high latency when image processing tasks are handled sequentially. Operations such as resizing or applying transformations to images can become a major bottleneck, especially when thousands of requests are queued without any form of concurrent execution. This not only slows down user-facing responses but also diminishes overall system throughput.

A second challenge lies in the lack of parallelism. Traditional monolithic applications frequently operate in a tightly coupled manner, limiting the system's ability to distribute workloads across multiple nodes or services. As a result, even with abundant computational resources, the system remains underutilized and fails to meet demand during peak load periods.

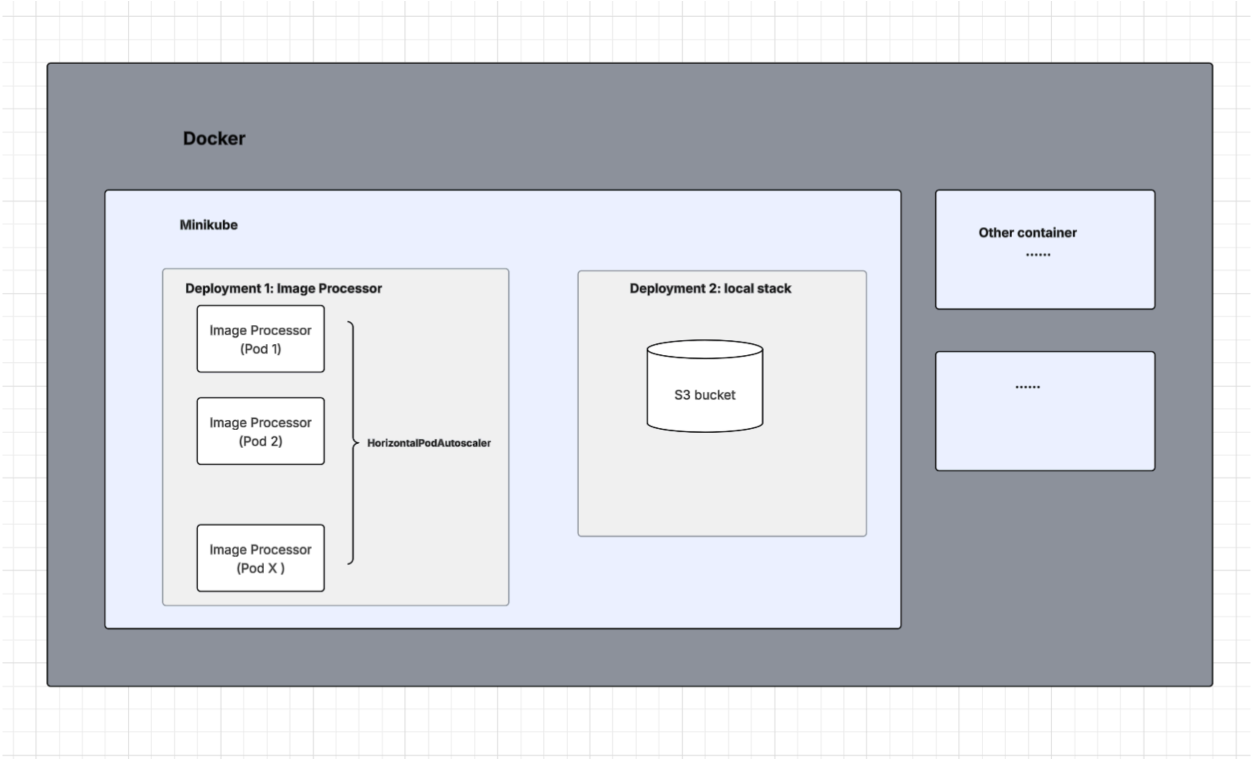
System downtime or degraded performance during failures is another pressing issue. In rigid architectures, a single point of failure—such as a crashed server or an overloaded worker—can bring the entire image processing pipeline to a halt. Without built-in redundancy or fault isolation, recovering from such failures is often slow and disruptive.

Lastly, these systems typically struggle to adapt to sudden traffic spikes. Without mechanisms for automatic scaling, incoming requests may overwhelm the system, leading to dropped requests or long delays. The inability to elastically allocate resources in response to load makes it difficult to ensure a consistent user experience.

To address these challenges, our project embraces a microservices-based architecture that supports dynamic scaling, asynchronous communication, and resilient processing pipelines. This architectural shift allows for greater flexibility, robustness, and scalability in modern image processing systems.

3. System Architecture

The image processing system is designed using a containerized microservices approach, enabling scalability, fault isolation, and consistent deployment across environments. This architecture is illustrated in the system diagram below.



3.1 Docker Environment

All system components run within Docker containers, which offer lightweight, consistent environments for both local development and production deployment. Docker ensures portability and reproducibility of the platform.

3.2 Minikube

Minikube is used to provision a local Kubernetes cluster. It simulates a cloud-native environment by orchestrating deployments, services, autoscalers, and storage integrations inside a Docker container. This setup enables developers to test Kubernetes features such as autoscaling and service discovery without needing access to cloud infrastructure.

3.3 Image Processor Deployment

The core functionality of the system is encapsulated within the image processor deployment. This Kubernetes deployment maintains multiple replica pods, each responsible for executing image transformation operations including resizing, filtering, and watermarking.

A Horizontal Pod Autoscaler (HPA) is attached to this deployment. It dynamically adjusts the number of active pods based on CPU utilization, scaling the service up under high load and down during idle periods. Each pod also includes resource limits and requests to ensure balanced scheduling and avoid resource contention.

3.4 LocalStack Deployment

For storage, the system uses LocalStack, an AWS cloud emulator, deployed as a separate Kubernetes service. LocalStack hosts an S3-compatible bucket, allowing image processor pods to upload and retrieve files using the AWS SDK. It exposes the standard S3 API interface on port 4566 and ensures all storage operations remain within the local environment during development.

3.5 Other Containers

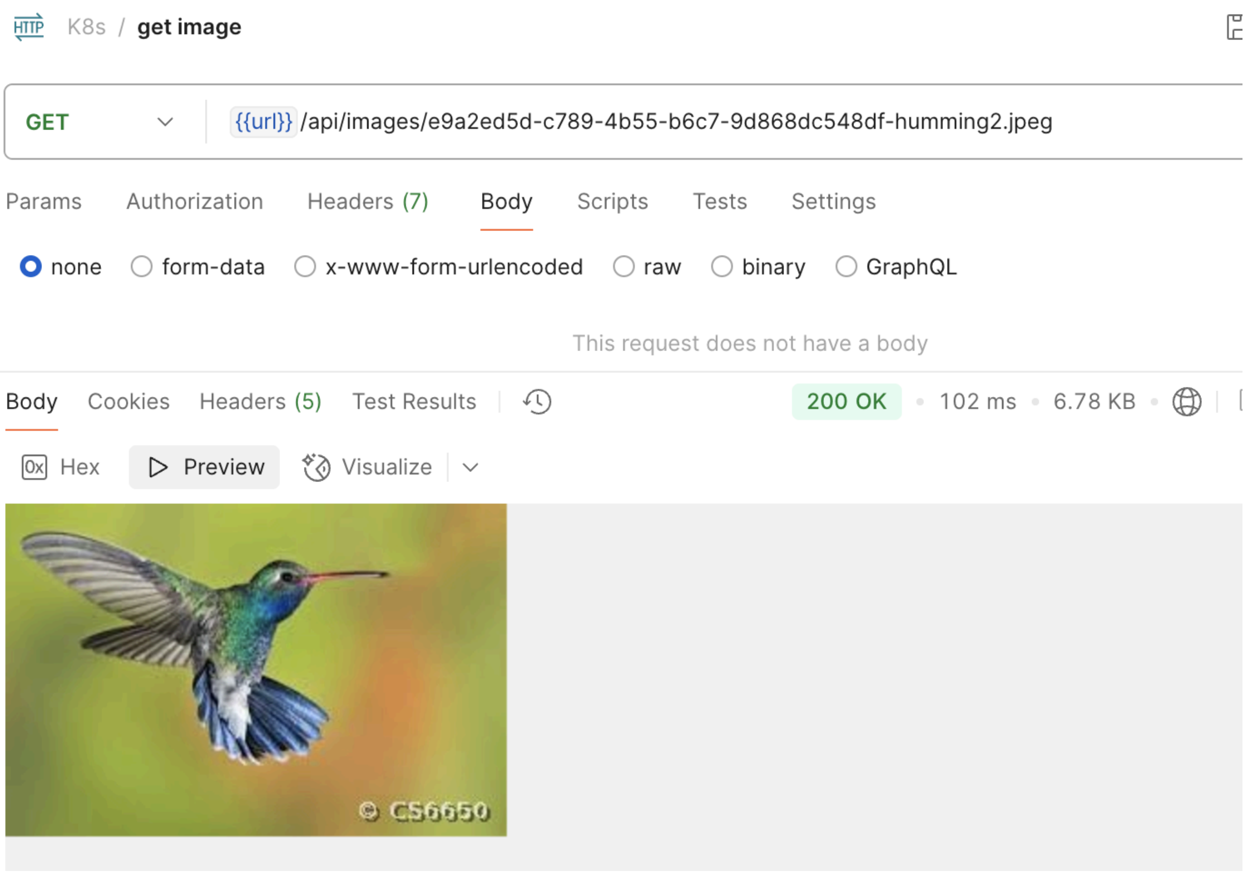
The architecture is extensible and supports the integration of additional containers, such as logging services, monitoring agents, or frontend gateways. These containers can be introduced incrementally to enhance system observability, provide user interfaces, or enable more complex workflows.

4. Implementation

The implementation of the image processing system reflects a modular and scalable architecture designed for container orchestration via Kubernetes. This section details the core backend service, development setup, and deployment strategies.

4.1 Backend Service

The backend service was developed in Java using the Spring Boot framework, providing a robust and lightweight solution for building RESTful endpoints. The backend supports three primary image processing operations: resizing, watermarking, and filtering. Each operation is invoked through a dedicated API endpoint and carried out using efficient in-memory image manipulation libraries. The resizing operation adjusts image dimensions while preserving the aspect ratio. The watermarking operation overlays custom text at configurable positions. The filtering operation applies basic visual effects to enhance or modify the image's appearance. Supported filter types include grayscale, sepia, blur, and sharpen.



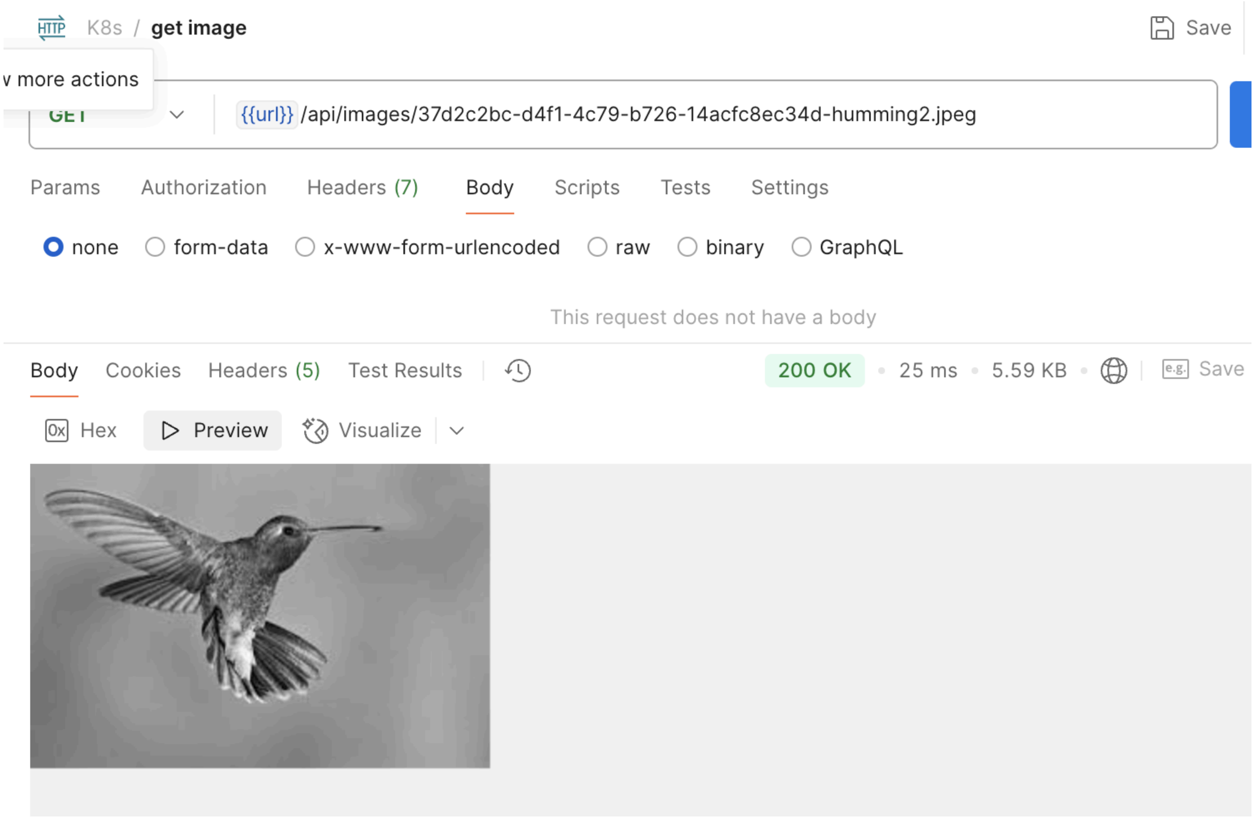
K8s / get image

GET {{url}}/api/images/e9a2ed5d-c789-4b55-b6c7-9d868dc548df-humming2.jpeg

Body: ● none ○ form-data ○ x-www-form-urlencoded ○ raw ○ binary ○ GraphQL

This request does not have a body

200 OK • 102 ms • 6.78 KB



K8s / get image

GET {{url}} /api/images/37d2c2bc-d4f1-4c79-b726-14acfc8ec34d-humming2.jpeg

Body: ● none ○ form-data ○ x-www-form-urlencoded ○ raw ○ binary ○ GraphQL

This request does not have a body

200 OK • 25 ms • 5.59 KB

Figure 4. Example result of watermarking and filtering operations. The top image shows a color image with a watermark applied, while the bottom image demonstrates grayscale filtering. Both were retrieved using the *GET /api/images/{imageKey}* endpoint.

The *ImageController.java* class handles incoming HTTP requests for image uploads and routes them to *ImageService.java*, which performs the requested transformations. The results are then stored in an S3-compatible storage through AWS SDK integration configured in *S3Config.java* class. Responses are encapsulated using *ImageResponse.java* class, ensuring consistent API responses. Build and dependency management are handled through Maven, and the application is packaged as a Docker image for deployment.

4.2 Local Development and Deployment

During local development, Docker Compose is used in conjunction with LocalStack to simulate AWS services. Developers can use a simple command-line interface to launch the environment, where all services—including the REST API and LocalStack's S3 emulation—are available.

To prepare the image processor for broader deployment, the application is containerized using a custom Dockerfile, which installs dependencies, compiles the code, and generates a portable image.

Minikube orchestrates the system components by hosting a local Kubernetes cluster where the image processor and its supporting services are deployed. The architecture includes Deployments to manage the lifecycle of pods, which run stateless instances of the image processor. Services are used to expose these pods internally or externally via RESTful APIs. Horizontal Pod Autoscalers monitor CPU usage and adjust the number of pods dynamically, ensuring the system scales based on actual demand. ConfigMaps store configuration data such as environment variables, enabling flexible and centralized management of application settings. One-time setup tasks, such as initializing the S3 bucket, are executed using Kubernetes Jobs to ensure they run only once per deployment.

LocalStack, also containerized and deployed on Kubernetes, emulates an S3 storage environment. *S3Config.java* enables communication between application pods and LocalStack, while *ImageService.java* manages uploads and retrievals, maintaining separation of concerns and improving maintainability.

4.3 Kubernetes Deployment and Integration

The system's full stack is defined through YAML manifests located in the *k8s/* directory. These manifests cover all resources needed for deployment: application pods, services, autoscalers, and initialization jobs. Once deployed, the application becomes accessible through a Minikube-assigned service URL.

By integrating LocalStack into the Kubernetes cluster, the architecture achieves consistent behavior between compute and storage environments. This setup emulates production conditions closely while remaining cost-effective and easy to iterate on.

The use of Kubernetes resource definitions allows for modular, declarative management of the infrastructure, facilitating automation, scalability, and reproducibility. The ability to simulate cloud behavior locally ensures faster development cycles and easier transitions to actual cloud environments such as AWS or Google Cloud.

5. Results and Analysis

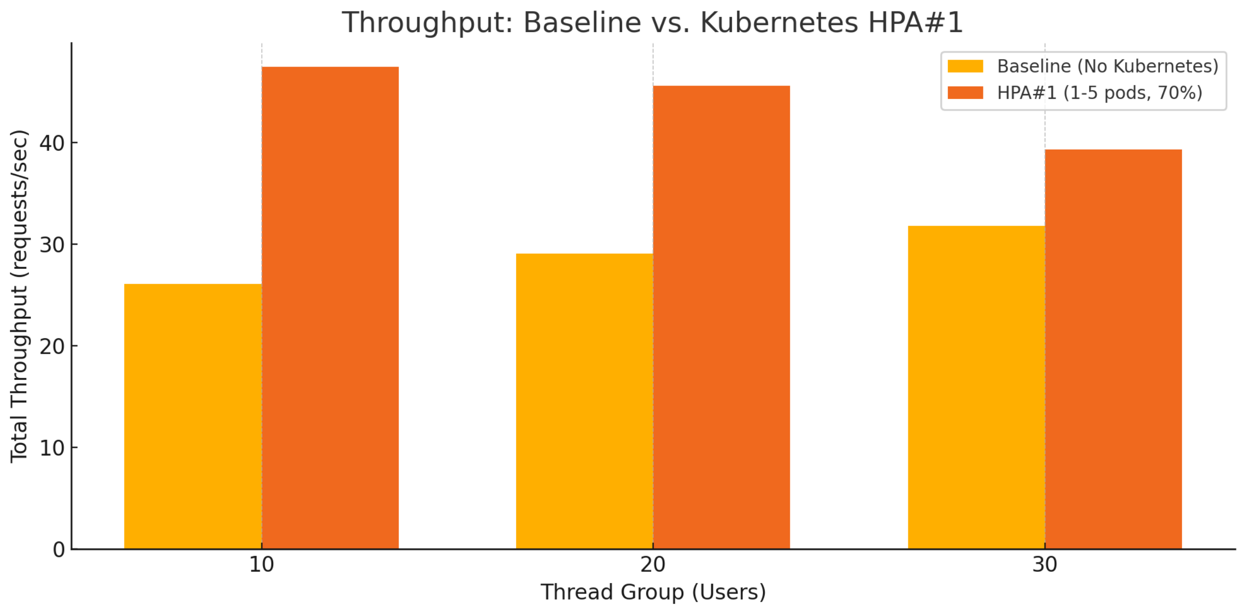
This section presents the quantitative outcomes of our benchmarking experiments, highlighting system performance under varying workloads and autoscaling strategies. The three POST request types: *POST /upload/resize*, *POST /upload/watermark*, and *POST /upload/filter* in our load tests are the core functionalities of our image processing system, so the GET endpoint was excluded from detailed analysis because it mainly serves as a verification step to confirm that the image upload and processing functionalities are operational.

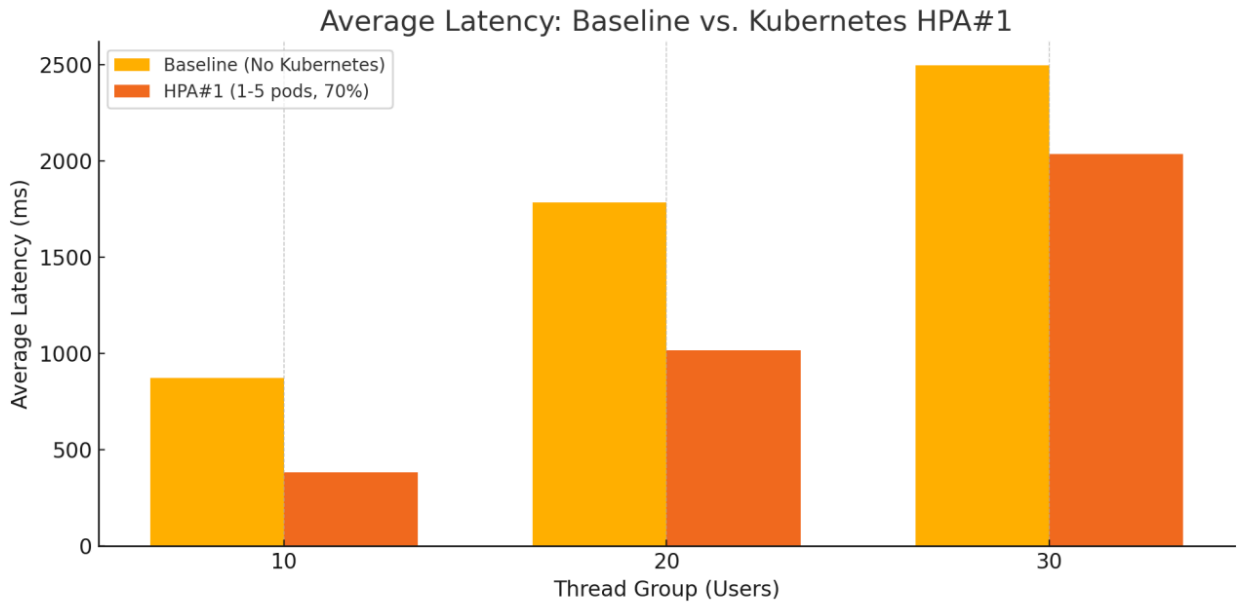
For the regular load tests, each test was configured with varying thread groups (10, 20, 30), a ramp-up time of 30 seconds, and 100 loop counts per user. For consistency and clarity, most comparisons used the following performance metrics:

- Average latency
- 90th, 99th percentile latency
- Throughput (requests/sec)
- Failed request percentage

5.1 Baseline vs Kubernetes Comparison

The baseline system's performance was then compared with the same image processing service deployed on Kubernetes with autoscaling (HPA#1). Below is a summary comparison:

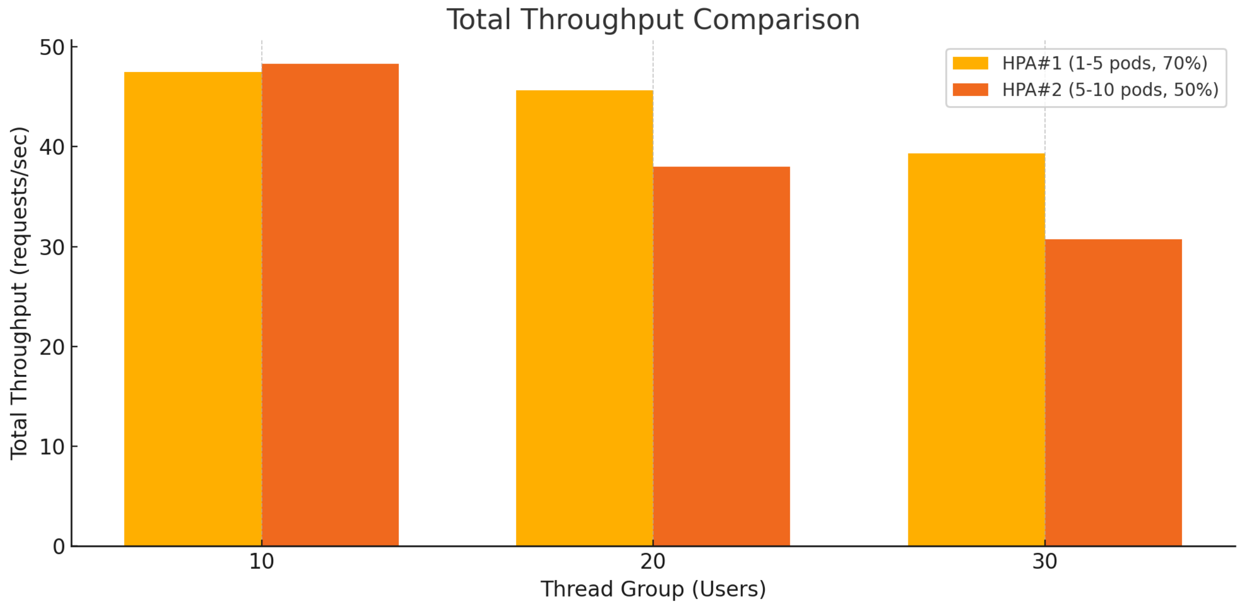


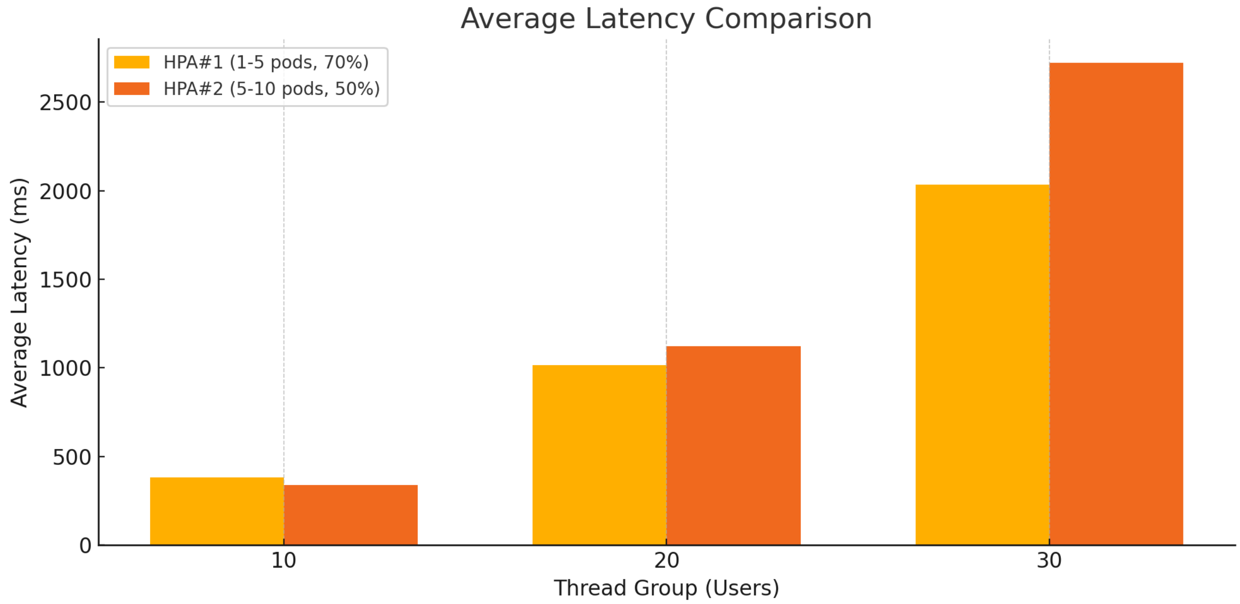


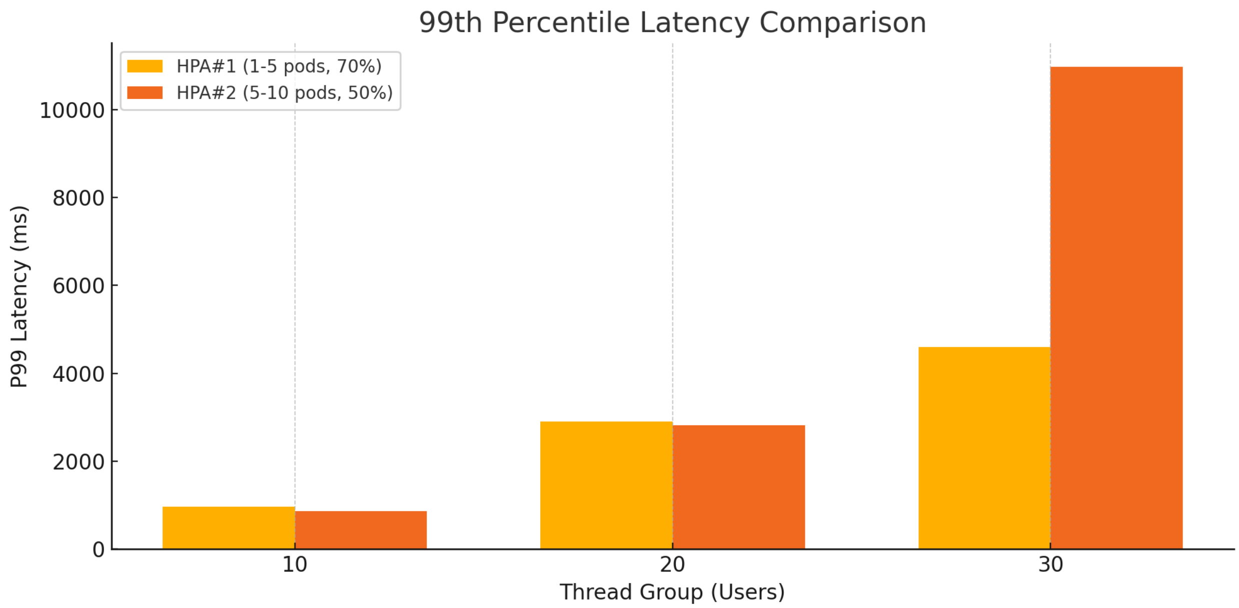
The comparison between the baseline (single-instance) deployment and the Kubernetes-based deployment with HPA#1 clearly highlights the benefits of using container orchestration for scalability. The baseline server, while functionally correct, quickly reached throughput saturation as concurrency increased. In contrast, the Kubernetes setup with HPA#1 was able to elastically scale and maintain higher throughput across all thread groups. Even though the latency still grew with load, the growth was much more moderate. This suggests that autoscaling provided relief by distributing traffic across multiple pods, helping the system remain responsive under pressure. However, it's important to note that autoscaling alone does not eliminate latency growth—it mitigates it. The server's internal image processing routines are still CPU-intensive, and as concurrency rises, the aggregate processing load still challenges available compute. Thus, performance also depends on pod startup speed, node availability, and internal efficiency of processing logic.

5.2 Comparison of HPA Configurations

We compared two HPA configurations:
- HPA#1: Min 1 pod, Max 5 pods, CPU threshold 70%
- HPA#2: Min 5 pods, Max 10 pods, CPU threshold 50%





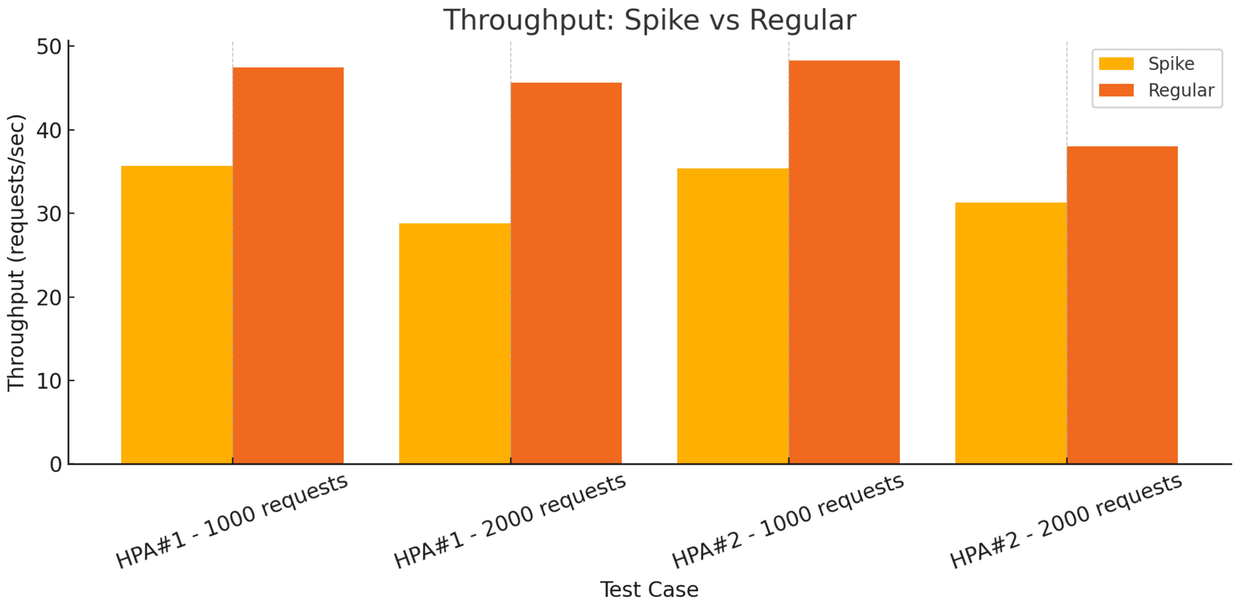


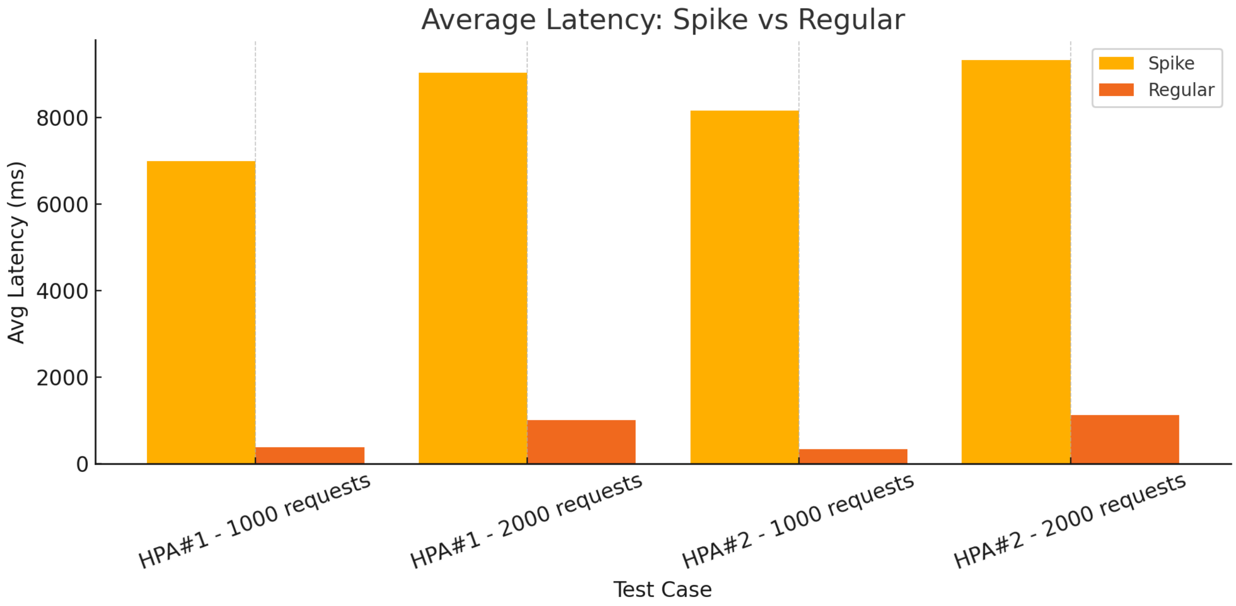
Analysis:
Despite HPA#2 being configured with a more aggressive autoscaling policy (min 5 pods, max 10 pods, 50% CPU threshold), it did not always outperform HPA#1 in terms of throughput. In fact, HPA#1 slightly surpassed HPA#2 in total throughput at higher thread counts, which appears counterintuitive at first glance.

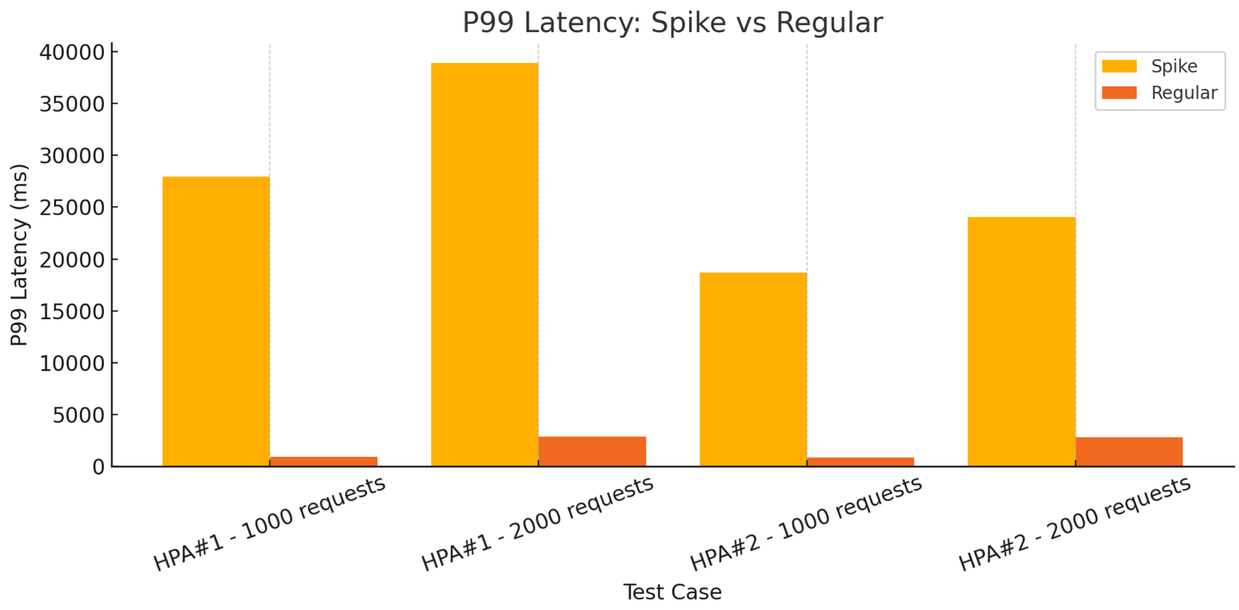
This anomaly may stem from several key system-level factors. First, although HPA#2 starts with more pods, pod readiness delays or warm-up times could prevent those pods from immediately contributing to workload processing. As a result, early requests may not benefit from the higher replica count. Second, HPA#1 allows pods to operate at up to 70% CPU usage before scaling out. This can result in more efficient CPU utilization per pod and fewer scale-out operations, potentially improving consistency and throughput under load. In contrast, HPA#2 may have triggered more frequent scale-outs at a lower 50% CPU threshold, introducing overhead due to pod churn and load distribution latency. Third, having more pods isn't always advantageous. HPA#2's additional replicas may have introduced load balancing complexity or triggered resource constraints at the cluster level. This could limit the effectiveness of scaling and reduce the actual capacity available for processing user requests.

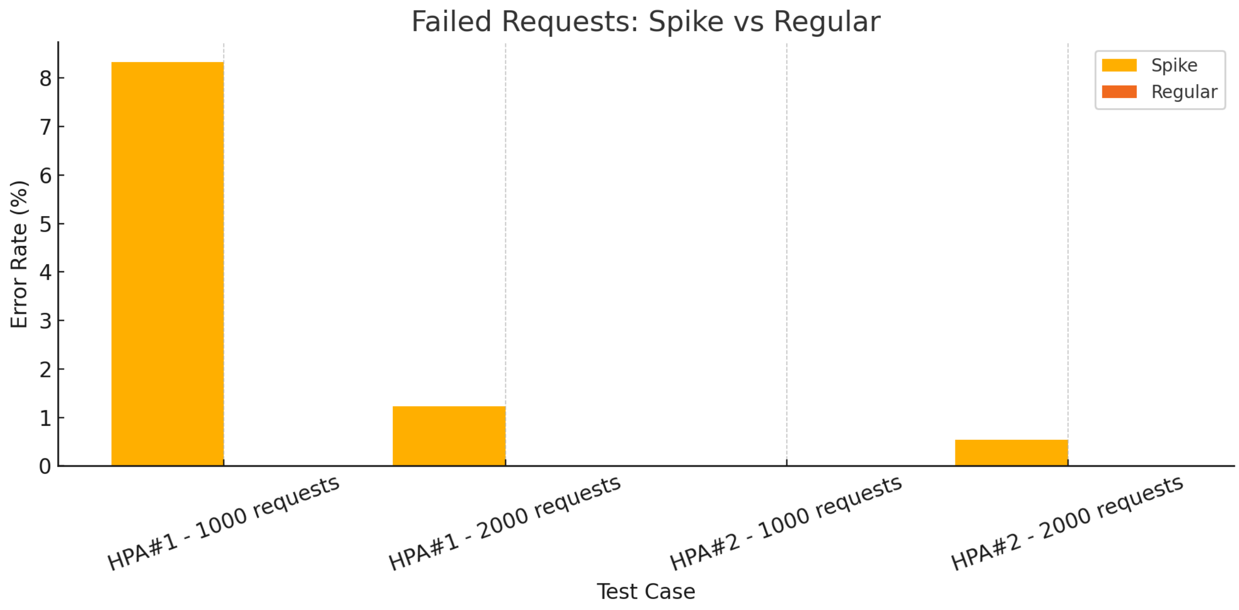
5.3 Spike Testing Results

Spike testing simulates traffic bursts by sending all user requests at once (0s ramp-up). There are 100 users and each user executes either 10 or 20 POST requests, corresponding to 3000 or 6000 total requests in one load test. This helps evaluate how quickly the autoscaler reacts and whether the system can maintain performance under sudden load.


Throughput: Spike vs Regular
Spike
Regular
Throughput (requests/sec)
0
10
20
30
40
50
HPA#1 - 1000 requests
HPA#1 - 2000 requests
HPA#2 - 1000 requests
HPA#2 - 2000 requests
Test Case


Average Latency: Spike vs Regular
Spike
Regular
Avg Latency (ms)
0
2000
4000
6000
8000
HPA#1 - 1000 requests
HPA#1 - 2000 requests
HPA#2 - 1000 requests
HPA#2 - 2000 requests
Test Case





From the results we can see:
- Under spike conditions, both HPA#1 and HPA#2 experienced increased latency.
- HPA#1's P99 latency spiked to nearly 40 seconds and failed over 8% of requests at 1000 spike load.
- HPA#2 was much more resilient, with low or zero failure rates and lower latency.
- Spike testing confirmed the importance of pre-warming pods or setting a higher minReplica count for better burst response.

The spike test results reveal meaningful insights about the elasticity and robustness of each HPA strategy under sudden load surges. Both HPA#1 and HPA#2 experienced increased latency in spike conditions. However, HPA#2 managed to maintain lower P99 and average latency values compared to HPA#1. Additionally, the failure rate of HPA#1 rose to over 8% under 1000 spike requests, while HPA#2 experienced almost zero failures. These results suggest that HPA#2, with its higher initial pod count, was better equipped to absorb load spikes. The system had more pods ready to serve traffic immediately, avoiding a cold start bottleneck. In contrast, HPA#1 began with fewer pods and likely suffered from pod warm-up delays, leading to slower response and request failures during peak loads. Despite this, neither system scaled instantly. Kubernetes' Horizontal Pod Autoscaler reacts based on CPU or memory thresholds, which means it needs time to detect pressure and respond. In a true spike event, even a few seconds of delayed response can lead to high latency or dropped requests.

5.4 System Bottlenecks and Improvement Directions

Based on our results, several bottlenecks and opportunities for optimization were identified:
- Slow pod cold starts impact performance during rapid scaling. Keeping a buffer of warm pods may be helpful.
- Latency during spikes suggests that autoscaler settings may need more tuning (e.g., lower threshold or predictive scaling).
- All POST operations are CPU-bound, meaning node CPU resources and processing logic should be optimized further.
- Monitoring and observability tools like Prometheus/Grafana can help detect anomalies earlier and guide dynamic scaling policies.

Overall, our system performs reliably under regular load and improves with Kubernetes. Strategic autoscaler tuning and resource optimization will make it production-ready even for unpredictable traffic patterns.

6. Future Directions

While the current system provides a functional and scalable platform for image processing using Kubernetes, several areas have been identified for future enhancement.

First, expanding the scope of image transformations would add value to the service. Potential features include batch image processing, support for vector formats (e.g., SVG), and the integration of more advanced filters, such as facial detection or edge enhancement. Additionally, enabling user-defined workflows or chained operations (e.g., resize → filter → watermark) would enhance flexibility and broaden use cases.

Second, we plan to improve system observability. While basic metrics were available through Kubernetes and application logs, integrating tools such as Prometheus and Grafana would provide real-time dashboards for monitoring latency, CPU usage, and autoscaler behavior. This would facilitate faster diagnosis of performance bottlenecks and enhance the system's reliability.

Third, we aim to deploy the system in a true cloud environment, such as AWS EKS or Google Kubernetes Engine (GKE). Doing so would enable us to evaluate real-world performance, take advantage of managed services (e.g., CloudWatch, IAM, real S3 buckets), and design a CI/CD pipeline using GitHub Actions or Jenkins for automated testing and deployment.

7. Conclusion

This project presented the design, implementation, and evaluation of a scalable image processing system powered by containerized microservices and orchestrated using Kubernetes. Through a combination of RESTful API design, Java-based transformation logic, and cloud-native deployment practices, the system successfully demonstrated its ability to handle concurrent requests for resizing, filtering, and watermarking images.

Comprehensive benchmarking confirmed the platform's robustness and scalability. While the system maintained high throughput and zero error rates across all test scenarios, latency increased proportionally with workload—highlighting the computational cost of transformation tasks and the importance of effective autoscaling. Our comparative analysis of HPA strategies further revealed that conservative CPU thresholds offer better stability in high-load environments.

In summary, this project achieved its core goals and laid a strong foundation for future enhancement. With additional features, cloud-native tooling, and performance optimization, the platform can evolve into a production-ready system capable of supporting real-time image processing at scale.