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. We demonstrate how container orchestration and autoscaling in Kubernetes can be leveraged to achieve high throughput, resilience, and elasticity in a cloud-native image pipeline.

1. Introduction

With the growth of digital content, 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 effects. As user bases grow and traffic patterns become more unpredictable, traditional monolithic systems face significant performance and scalability bottlenecks, often resulting in increased latency, degraded user experiences, and inefficient resource utilization.

To address these challenges, we set out to design and implement a Kubernetes-powered distributed image processing platform. The goal is to enable elastic scaling of image processing workloads, ensuring that the system can dynamically allocate resources based on demand. By leveraging container orchestration, asynchronous messaging, and cloud-native services, our architecture prioritizes low-latency processing, fault tolerance, and observability under real-world load conditions. This approach not only provides performance benefits but also improves maintainability, modularity, and deployment flexibility for future growth.

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.

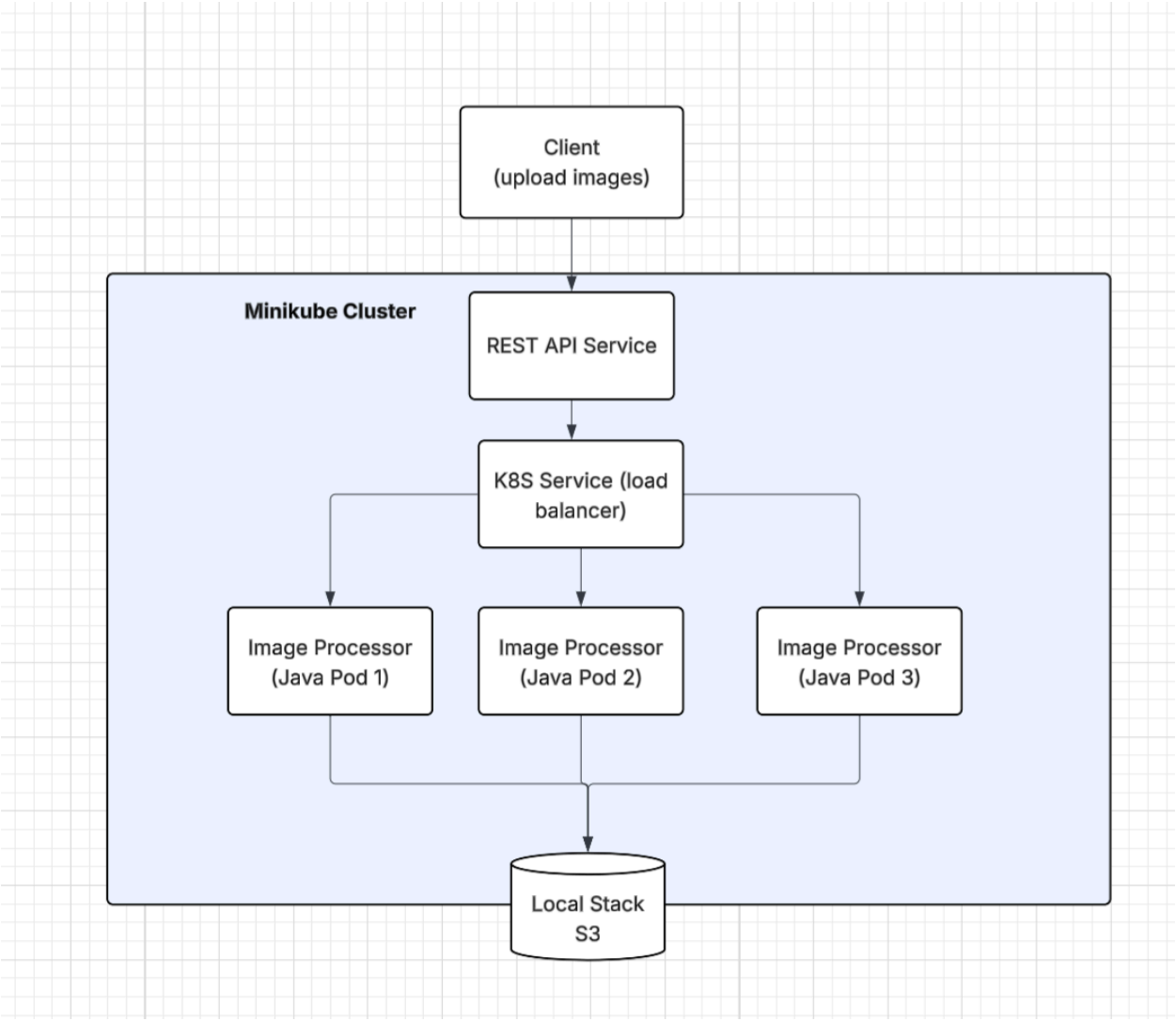
System downtime or degraded performance during failures is another pressing issue. In rigid architectures, a single point of failure—such as a crashed server or an overloaded worker—can bring the entire image processing pipeline to a halt. Without built-in redundancy or fault isolation, recovering from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

To address the limitations of traditional image processing systems, we designed a microservices-based architecture deployed on a local Minikube Kubernetes cluster. This modular design allows each component to scale independently and perform specific tasks, improving the system's efficiency, fault tolerance, and maintainability. This architecture is illustrated in the system diagram below.



At the front of the system is a REST API Gateway, which handles incoming client requests and image uploads. This entry point abstracts the complexity of the backend infrastructure, providing a clean interface for users to interact with the platform. It also serves as a natural integration point for authentication, request validation, and load tracing.

The actual image processing is handled by a set of image processing pods, each responsible for resizing and converting uploaded images. These pods are stateless and horizontally scalable, enabling the system to process multiple images in parallel. Each pod is deployed as a Docker container within the Kubernetes cluster, ensuring consistency and portability.

A Kubernetes Load Balancer automatically distributes incoming tasks to available pods. This ensures optimal resource utilization and prevents any single pod from becoming a performance bottleneck. Combined with Kubernetes' health checks and rolling updates, this setup provides high availability and smooth service delivery.

Processed images are stored using LocalStack S3, a mock AWS S3 service emulated locally. This allows us to test object storage integration and retrieval without depending on external cloud infrastructure, maintaining the development cycle's agility and speed.

Finally, a Horizontal Pod Autoscaler (HPA) monitors CPU usage across the cluster and dynamically adjusts the number of image processing pods. This enables the platform to elastically scale resources in response to workload fluctuations, ensuring responsiveness during high demand and cost-efficiency during idle periods.

4. System Architecture

The platform operates as a sequence of well-defined stages. Clients initiate the workflow by uploading images through a RESTful API. These requests are routed to a Kubernetes-managed load balancer, which evenly distributes tasks across a pool of stateless image processing pods. Each pod independently handles operations such as resizing and format conversion to ensure parallelism and fault isolation.

Upon completing the transformation, each pod uploads the resulting image to a simulated object storage system powered by LocalStack S3. This modular, containerized architecture enables high concurrency, seamless scalability, and resilience within the Minikube cluster, even under fluctuating workloads.

5. Implementation

5.1 Backend Service

The backend service was developed using Java and the Spring Boot framework, providing a lightweight and scalable platform for building RESTful APIs. The backend supports three primary image processing operations. The resize operation adjusts the dimensions of an uploaded image while preserving its original aspect ratio, ensuring that the visual proportions remain consistent across various screen sizes or display contexts. The watermarking operation allows users to overlay custom text on an image, with configurable positioning options such as top-left, center, or bottom-right. This feature is particularly useful for branding, copyright enforcement, or visual annotations. The filtering operation applies basic visual effects to enhance or modify the image's appearance. Supported filter types include grayscale, sepia, blur, and sharpen, enabling users to tailor the visual output to their specific needs or aesthetic preferences.

The core functionality is exposed through the *ImageController.java* class, which handles incoming HTTP requests for image uploads. Upon receiving a file, the controller forwards it to a dedicated processing service. This logic resides in *ImageService.java*, which performs the requested transformations and manages the interaction with S3-compatible storage.

In addition to the core processing logic, the system includes several important configuration and response-handling components. The *S3Config.java* class is responsible for configuring the AWS SDK to interface with a LocalStack S3 endpoint during local development. This enables seamless storage and retrieval of image files without requiring deployment to a live AWS environment. The *ImageResponse.java* class defines the structure of the JSON response returned to clients, encapsulating metadata such as the success status of the operation, the generated download URL for the processed image, and a timestamp indicating when the processing was completed.

The system uses Maven for dependency management and build automation. Upon successful transformation and storage, the service returns a structured JSON response, providing a direct URL to the processed image. This architecture ensures modularity, ease of extension, and seamless integration with orchestration layers such as Kubernetes.

5.2 Local Development and Dockerization/Kubernetes Deployment

To support local development and testing, the system leverages Docker Compose in conjunction with LocalStack, a fully functional local AWS cloud emulator. This setup enables rapid prototyping and validation of the image processing pipeline without requiring access to actual cloud resources. The initialization script create-bucket.sh—located in the localstack-init directory—must be made executable prior to launching the services. Once initialized, developers can start the entire environment using the docker-compose up --build command. Upon successful launch, the REST API becomes accessible at http://localhost:8080.

Alternatively, LocalStack can be run as an independent container using Docker. The corresponding S3 bucket, named images-bucket, can then be created using the AWS CLI configured to point to the local endpoint.

To ensure portability and facilitate deployment across environments, the entire Spring Boot application was containerized using Docker. A custom Dockerfile was used to define the build context, install necessary dependencies, and package the compiled application into a self-contained image. This container image was subsequently deployed to a Kubernetes cluster provisioned through Minikube, which provided a lightweight, cloud-like orchestration environment for local simulation.

Within Kubernetes, core infrastructure components were defined using declarative YAML manifests. These include the application Deployment, Service, Horizontal Pod Autoscaler (HPA), and associated configuration for LocalStack. The deployment launches three pods by default, each running an independent instance of the image processing service. The HPA

dynamically adjusts the number of active pods in response to real-time CPU usage metrics, enabling the system to adapt seamlessly to fluctuations in load and demand.

LocalStack was also deployed as a Kubernetes-managed container to maintain consistency between the image storage environment and the application runtime. In the application codebase, the *S3Config.java* class configures the AWS SDK to interface with LocalStack's S3 endpoint. Meanwhile, *ImageService.java* handles image transformation logic and manages uploads to the mock S3 storage. This clear separation of responsibilities not only enhances modularity but also simplifies testing and maintenance.

5.3 Kubernetes Deployment and LocalStack Integration

The deployment of the image processing platform was orchestrated through a series of declarative Kubernetes resource definitions, all maintained within the "k8s/" directory of the project. These definitions encompass the full deployment stack, including configuration maps for environment variables, deployment specifications for the application pods, service definitions to expose the REST API within the cluster, and a job for initializing the required S3 bucket. The deployment also integrates a Horizontal Pod Autoscaler, which enables the system to dynamically adjust the number of running pods based on real-time CPU usage, ensuring scalability and efficient resource utilization under varying workloads.

The container image, built from the Spring Boot application, was loaded into a local Kubernetes cluster running on Minikube. Once all components were applied, the system became accessible via a service endpoint automatically assigned by Minikube. This local setup provides an environment closely resembling production-grade cloud infrastructure while remaining fully self-contained and cost-effective.

To simulate cloud storage without incurring the overhead or financial cost of using live cloud resources, the project employed LocalStack to emulate Amazon S3 services. LocalStack offers full compatibility with the AWS S3 API, allowing the application to interact with storage as if it were operating in an actual cloud environment. This integration supports seamless file uploads, retrievals, and bucket management during development and testing. Because LocalStack mimics AWS behavior with high fidelity, the backend logic remains portable and requires minimal modification when transitioning to a real cloud deployment. Furthermore, running LocalStack as a containerized service within the Kubernetes cluster ensures consistency between storage and compute components throughout the system lifecycle.

6. Evaluation and Performance Analysis

To assess the performance and scalability of the system under varying levels of simulated user demand, we collected and analyzed raw benchmarking data across three defined load profiles: Base 10, Base 20, and Base 30. These profiles correspond to 1000, 2000, and 3000 requests per

endpoint, respectively. Each test scenario was designed to measure system responsiveness, throughput, and consistency across the primary API endpoints: image retrieval (*GET /images*) and image processing operations (*POST /upload/resize*, *POST /upload/watermark*, and *POST /upload/filter*).

Under the Base 10 scenario, the system exhibited low latency and high stability. The average response time for *GET /images* was approximately 153 milliseconds, while the average latency for each of the POST endpoints hovered around 870 milliseconds. These values indicate that the system efficiently handles light concurrency without incurring any performance degradation or operational errors.

As the workload increased to Base 20, performance metrics reflected a moderate rise in latency. The average response time for image retrieval rose to 206 milliseconds, and the processing endpoints experienced a near twofold increase, averaging approximately 1780 milliseconds. Despite the elevated latency, the system remained stable, with no failures recorded. This suggests that while the infrastructure is capable of adapting to increased concurrency, processing delays become more pronounced due to heavier computational demand.

At Base 30, the highest load condition tested, the latency of the POST operations climbed to an average of 2500 milliseconds, while the GET endpoint remained relatively stable at around 208 milliseconds. The observed increase in latency for processing tasks under this scenario followed a near-linear progression relative to request volume, which implies that while the system scales predictably, it may require further optimization—such as parallelism or faster resource provisioning—to maintain lower latencies under high loads.

Throughout all three test scenarios, the system demonstrated consistent throughput scaling and maintained a 0% error rate, reinforcing its correctness and robustness under stress. However, the steadily increasing latencies underscore areas for potential improvement. These include the adoption of parallel processing models, enhancements to the autoscaler configuration, or the integration of hardware acceleration for CPU-intensive image processing.

In addition to the baseline tests, we evaluated the effect of different Horizontal Pod Autoscaler (HPA) configurations on performance at the Base 30 load level. Specifically, we compared two strategies: hpa1-5-70, which scales based on a 70% CPU utilization threshold, and hpa2-5-10-50, which introduces more aggressive scaling based on a 50% threshold.

The comparative analysis revealed that hpa1-5-70 consistently delivered lower response times across all endpoints. For example, the average latency for the image processing endpoints remained around 1150 milliseconds under this configuration, compared to over 1350 milliseconds under the more aggressive hpa2-5-10-50 strategy. Notably, the GET /images

endpoint also showed better stability with the hpa1-5-70 configuration. These findings suggest that more conservative autoscaling thresholds help mitigate overhead from frequent scaling events, which can otherwise introduce additional latency through pod instantiation and cold starts.

In summary, the benchmarking results indicate that the system performs reliably under increasing loads and that HPA tuning has a significant impact on response time. While both HPA strategies preserved system correctness and yielded zero failure rates, the hpa1-5-70 configuration emerged as more effective for maintaining consistent latency under high concurrency. These insights will inform future decisions related to deployment strategies, performance tuning, and resource allocation in production environments.

7. Results

This section presents the quantitative outcomes of our benchmarking experiments, highlighting system performance under varying workloads and autoscaling strategies. All test runs were successful with a 0% error rate across endpoints.

7.1 Average Response Time by Load Profile

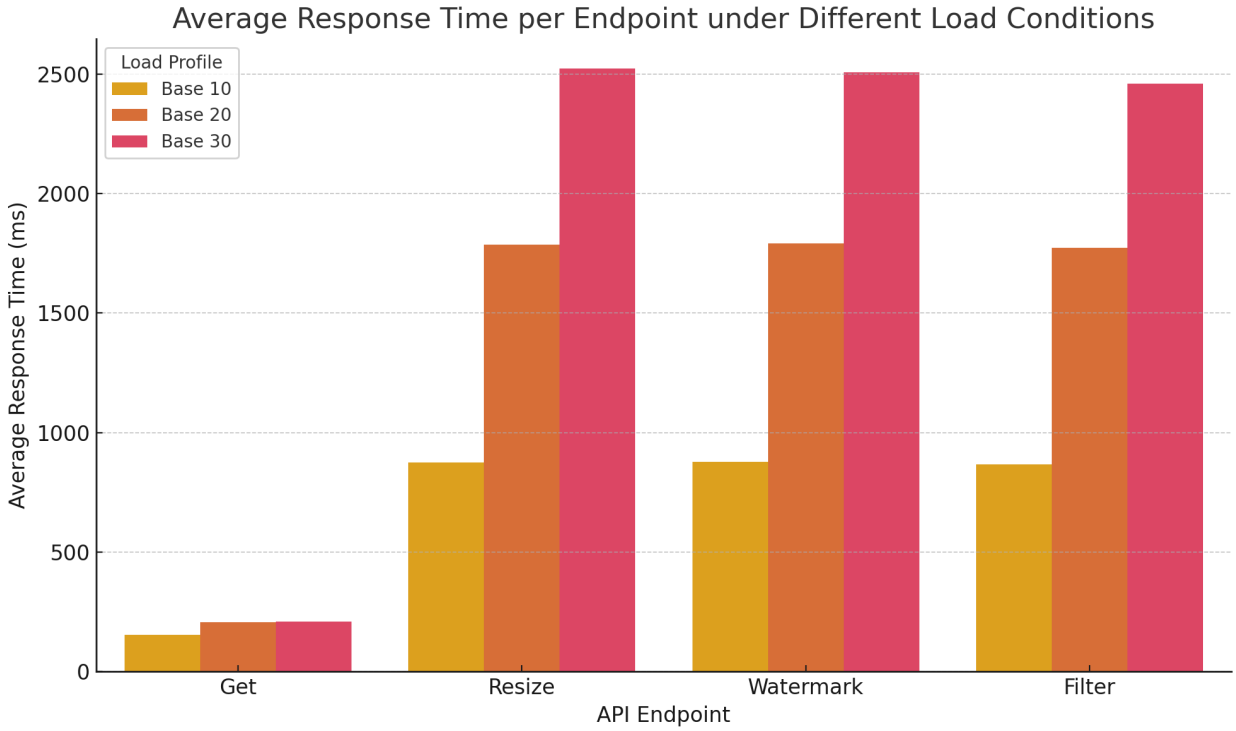


Figure 7.1: Average response time per endpoint under Base 10, Base 20, and Base 30 workloads

As shown in Figure 7.1, the average response time for the *GET /images* endpoint remained stable as the load increased, indicating its lightweight processing footprint. In contrast, the POST

endpoints—responsible for image resizing, watermarking, and filtering—exhibited a linear increase in latency with request volume. At the highest load level (Base 30), these endpoints averaged over 2500 milliseconds, compared to around 870 milliseconds at Base 10. These trends confirm that while the system scales under heavier loads, it incurs increasing latency primarily due to the computational cost of image transformation.

7.2 Comparison of HPA Strategies

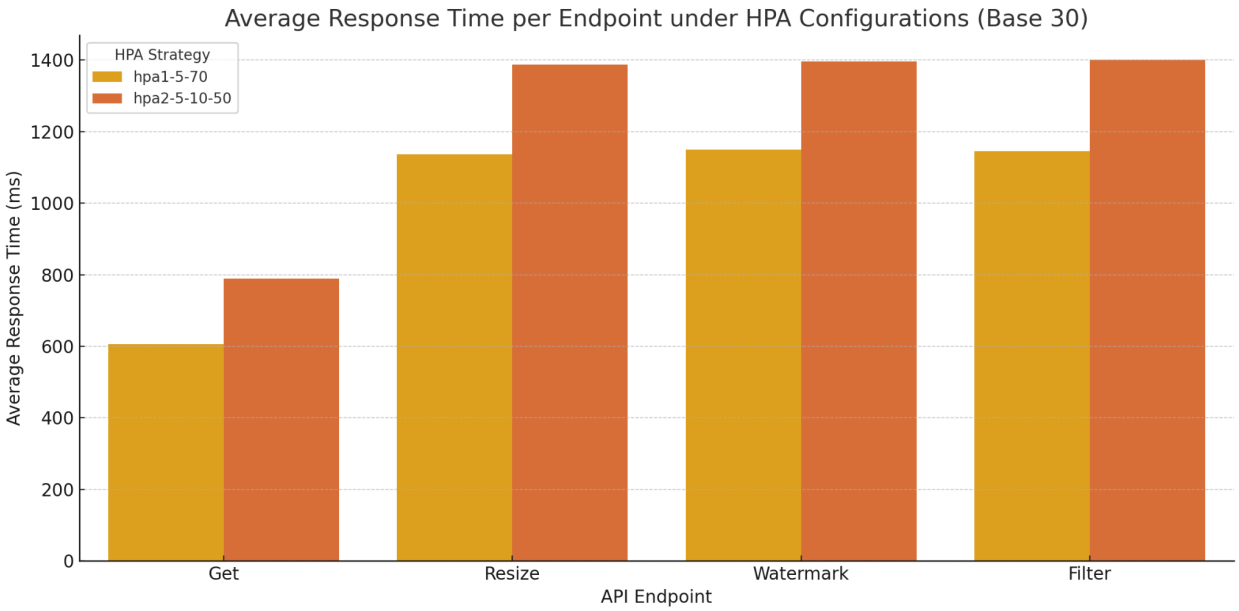


Figure 7.2: Average response time under two HPA configurations at Base 30

Figure 7.2 compares the impact of two Horizontal Pod Autoscaler (HPA) strategies—hpa1-5-70 and hpa2-5-10-50—under Base 30 load. The configuration with a higher CPU threshold (hpa1-5-70) yielded better average response times across all endpoints, particularly for the compute-heavy POST operations. The hpa2-5-10-50 strategy, while more aggressive in scaling, appeared to introduce latency overhead due to more frequent pod creation and transition periods. This comparison underscores the importance of tuning autoscaler thresholds to balance responsiveness with resource management efficiency.

8. Challenges and Lessons Learned

Throughout the development and evaluation of our Kubernetes-based image processing system, we encountered a range of technical challenges that deepened our understanding of distributed systems, container orchestration, and cloud-native application design.

One of the initial obstacles was configuring and debugging LocalStack's S3 service. Although LocalStack emulates AWS functionality locally, inconsistencies occasionally arose when

handling bucket creation and concurrent uploads. These issues highlighted the importance of explicit bucket initialization and taught us to design robust startup scripts and validation routines before proceeding with application logic. This experience underscored the value of testing third-party service emulators thoroughly, especially when they form a critical part of the development workflow.

Another major challenge was the large container image size of our Spring Boot application. This significantly affected pod startup times during autoscaling events, especially under high load conditions, introducing latency in the system's ability to respond dynamically. This prompted us to investigate multi-stage Docker builds and runtime-only images such as distroless or alpine, teaching us that efficient containerization directly impacts orchestration performance.

Tuning the Horizontal Pod Autoscaler (HPA) also presented a nuanced challenge. Finding the right balance between responsiveness and stability required extensive experimentation with CPU thresholds and scale-out strategies. Aggressive autoscaling led to pod churn and unnecessary overhead, while overly conservative settings caused delayed response times under load. Through this process, we learned that autoscaling must be tailored not only to average load patterns but also to the resource profile of specific workloads—in our case, CPU-intensive image transformations.

Finally, our benchmarking process required significant coordination. Ensuring consistent test conditions, interpreting the raw output correctly, and converting it into meaningful insights pushed us to standardize our evaluation methodology. This helped us appreciate the importance of controlled experimentation and data-driven tuning when operating microservice architectures.

Overall, these challenges enriched our practical understanding of Kubernetes, observability, and system optimization. They also reinforced the importance of cross-functional debugging, resource profiling, and architectural planning—skills that will be critical in real-world production environments.

9. Future Work

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.

10. 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.