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.

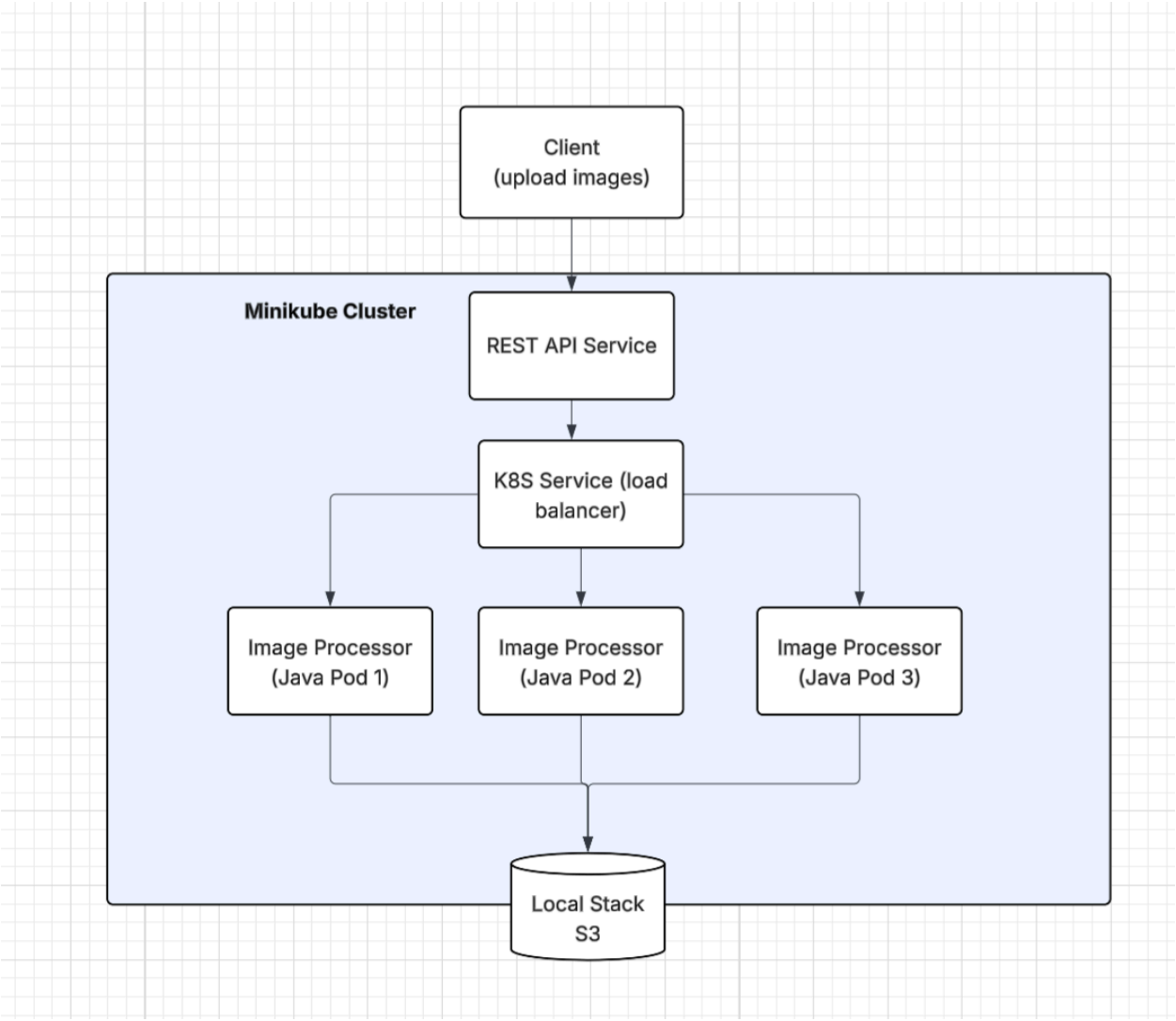
System downtime or degraded performance during failures is another pressing issue. In rigid architectures, a single point of failure—such as a crashed server or an overloaded worker—can bring the entire image processing pipeline to a halt. Without built-in redundancy or fault isolation, recovering from 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 core functionality is exposed through the `ImageController.java` class, which handles incoming HTTP requests for image uploads. When a client submits an image, the controller forwards the file to a dedicated service for processing. Upon successful transformation and storage, the service responds with a URL pointing to the processed image, making the result easily accessible for clients or downstream systems.

5.2 Dockerization & K8s Deployment

To ensure portability and ease of deployment, the entire application was containerized using Docker. A custom `Dockerfile` was written to define the build environment, install necessary dependencies, and package the Spring Boot application into a runnable container image. This image was then deployed to a Minikube-based Kubernetes cluster, which allowed us to simulate a cloud-native environment locally.

Within Kubernetes, we defined and applied resource configurations such as Deployments, Services, and a Horizontal Pod Autoscaler (HPA). The deployment consists of three pods, each running an instance of the Java image processor. These pods are automatically scaled based on CPU utilization to handle variable workloads efficiently.

The supporting infrastructure includes LocalStack, which was initialized using docker-compose as a standalone container alongside the Kubernetes cluster. Within the backend codebase, the class `S3Config.java` is responsible for configuring the AWS SDK to interact with the local S3-compatible endpoint provided by LocalStack. The `ImageService.java` class orchestrates the image transformation logic and manages the upload process to the mock S3 storage, ensuring a clean separation of concerns within the application.

5.3 LocalStack for S3

To avoid the overhead and cost of deploying to an actual AWS environment during development, we used LocalStack to emulate AWS S3 functionality. This allowed our system to interact with a fully compatible S3 API, enabling seamless testing of file uploads, retrievals, and bucket management. Since LocalStack mimics real AWS behavior, the same backend logic can be deployed to a production environment with minimal changes. This approach helped accelerate development cycles while preserving cloud compatibility.

6. Testing and Evaluation

We used **Apache JMeter** to simulate concurrent clients and test system behaviour under increasing load.

Metrics Evaluated:

- Throughput: Images processed per second
- Latency: Time from request to completion
- Recovery: Response to pod failure
- Scalability: Resource utilization with autoscaling

7. **Results**

Graphs and detailed metrics can be found in Appendix A.

8. **Challenges & Lessons Learned**

- Debugging LocalStack S3 took more effort than expected.
- The container image size affected pod startup time.
- Autoscaler tuning was crucial for a balance between latency and resource cost.

9. **Future Work**

- Extend support for filters, watermarking, and facial recognition.
- Deploy on AWS EKS or GKE for production-grade testing.
- Integrate Prometheus & Grafana for monitoring dashboards.
- Implement CI/CD with GitHub Actions.

10. Conclusion

Our Kubernetes-based image processor demonstrates how microservices and container orchestration enable modern systems to meet high-throughput demands with resilience and elasticity. With minimal cost, we achieved scalability and laid a foundation for further enhancements like cloud deployment and advanced processing pipelines.

References