SCALABLE IMAGE PROCESSING PLATFORM WITH KUBERNETES

Presented By : Yuxin Wang
Jingwen Huang
Lan Wang

TABLE OF CONTENT

INTRODUCTION

Traditional image processing systems suffer from:

- ❌ High latency due to sequential task execution
- ❌ Poor scalability from monolithic designs
- ❌ Single points of failure, causing service downtime
- ❌ Inability to handle traffic spikes without autoscaling

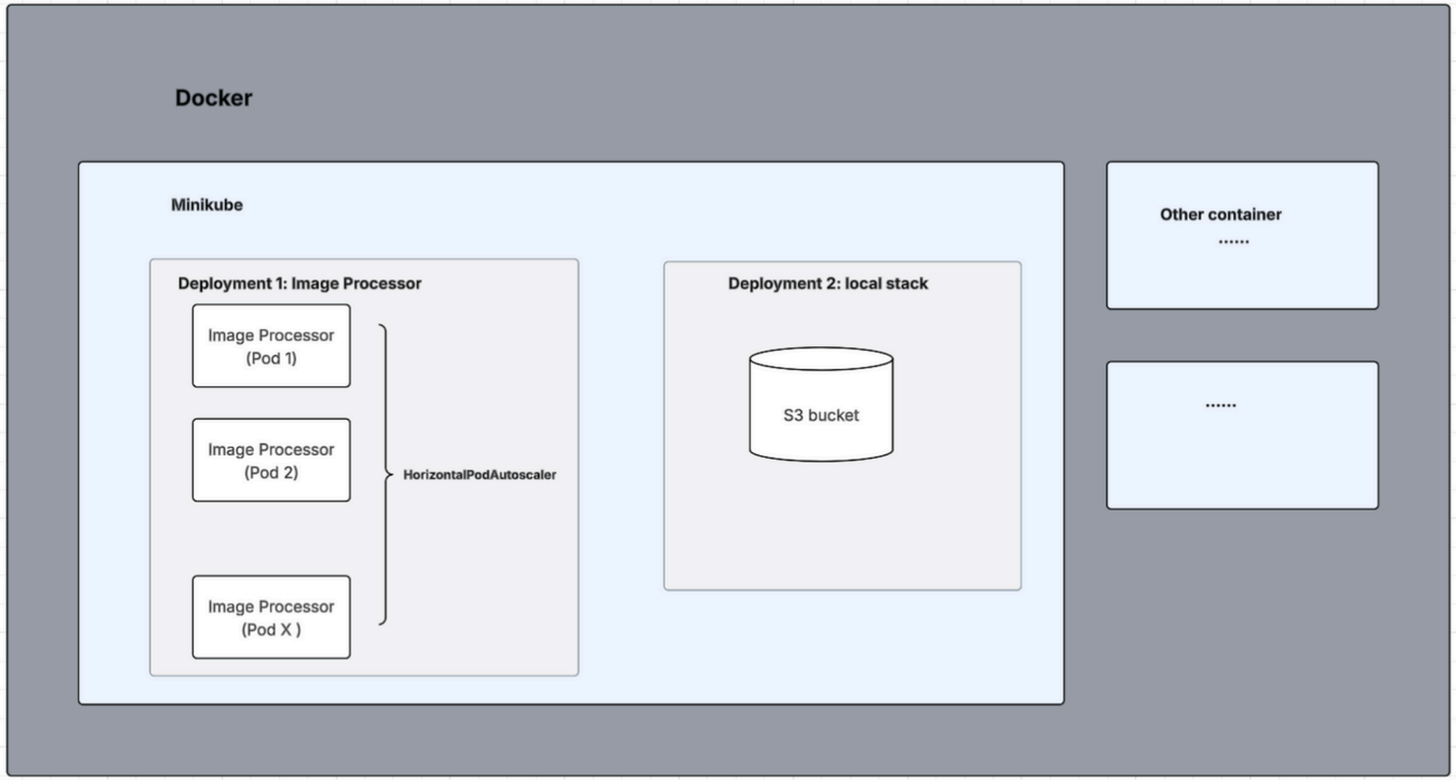
Goal: Build a scalable, resilient, and modular image processing platform

Key Features:

- REST API for image upload & transformations
- Java microservices (resize, watermark, filter)
- Kubernetes HPA for autoscaling
- S3-compatible storage via LocalStack
- Local deployment on Minikube

SYSTEM ARCHITECTURE

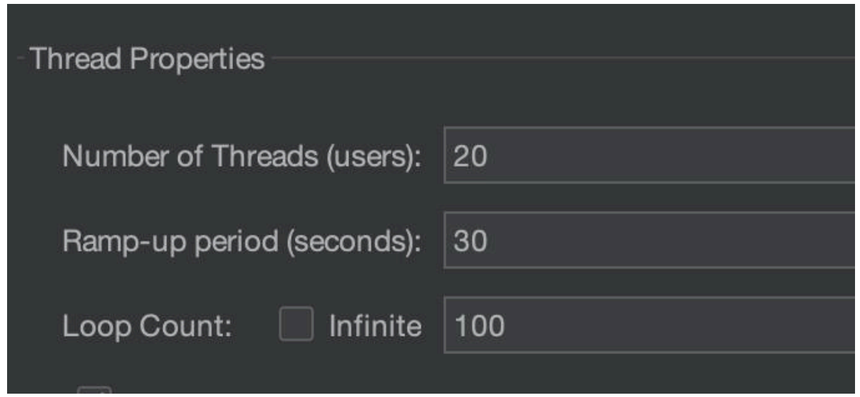
- **Containerized Microservices:** Each component runs in isolated Docker containers for portability and consistency.
- **Minikube Cluster:** Simulates cloud-native Kubernetes environment locally.
- **Image Processor Deployment:**
  - Stateless pods for resize, watermark, filter operations
  - Scales via Horizontal Pod Autoscaler (HPA) based on CPU load
- **LocalStack Deployment:**
  - Emulates AWS S3 for local image storage
  - Interacts with AWS SDK over port 4566
- **Extensible Design:**
  - Supports additional containers (e.g., monitoring, frontend, logging)

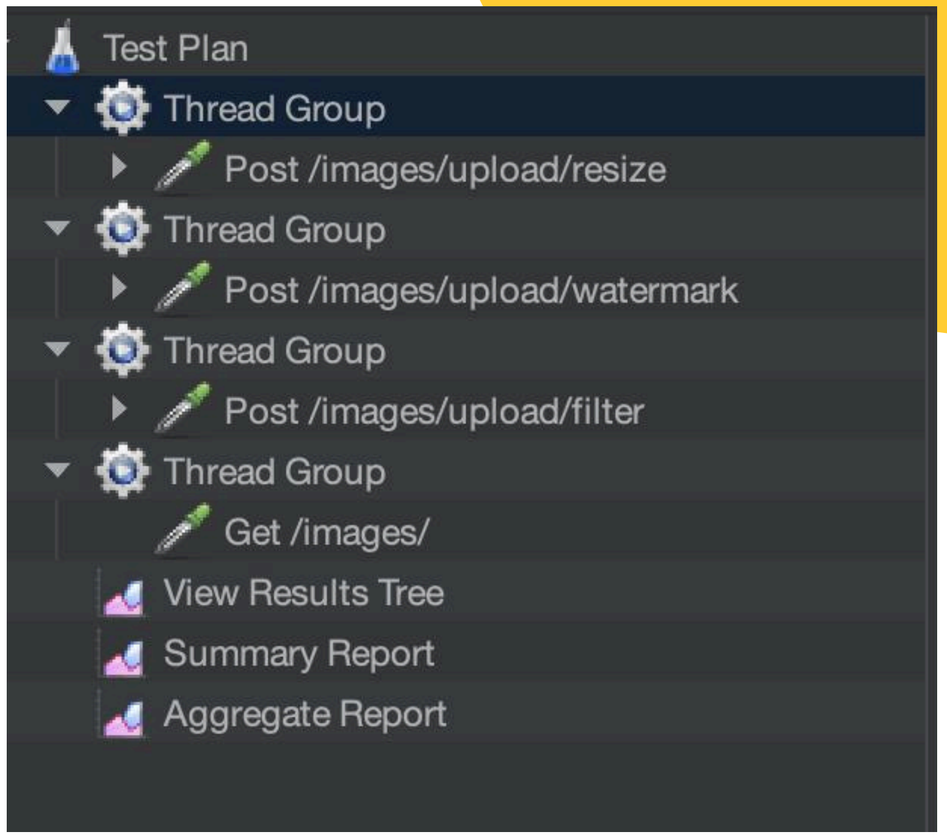


IMPLEMENTATION

- **Tech Stack:** Java (Spring Boot), Docker, Kubernetes, LocalStack (S3)
- **Core Endpoints:**
  - POST /resize, /watermark, /filter for image processing
  - GET /images/{imageKey} to retrieve results
- **Local Dev: Docker Compose + LocalStack for fast, offline testing**
- **K8s Deployment:**
  - **Managed with YAML (Deployments, Services, HPA, Jobs)**
  - **Autoscaling via CPU thresholds**
  - **ConfigMaps for environment setup**
- **Storage:** S3 API emulated with LocalStack; integrated via AWS SDK

RESULTS & ANALYSIS





Test Results- Raw data

Label	# Samples	Average	Median	90% Line	95% Line	99% Line	Min	Max	Error %	Throughput	Received KB/sec	Sent KB/sec
Get /images/	1000	153	153	275	309	471	2	770	0.000%	21.52528	120.27	3.74
Post /images/upload/resize	1000	874	810	1461	1696	2323	48	2941	0.000%	8.70413	3.90	25.24
Post /images/upload/watermark	1000	878	830	1431	1666	2180	58	3024	0.000%	8.69936	3.90	25.23
Post /images/upload/filter	1000	866	815	1438	1697	2200	58	2505	0.000%	8.71361	3.90	25.27
TOTAL	4000	693	678	1341	1597	2127	2	3024	0.000%	34.79744	60.30	77.19

RESULTS & ANALYSIS

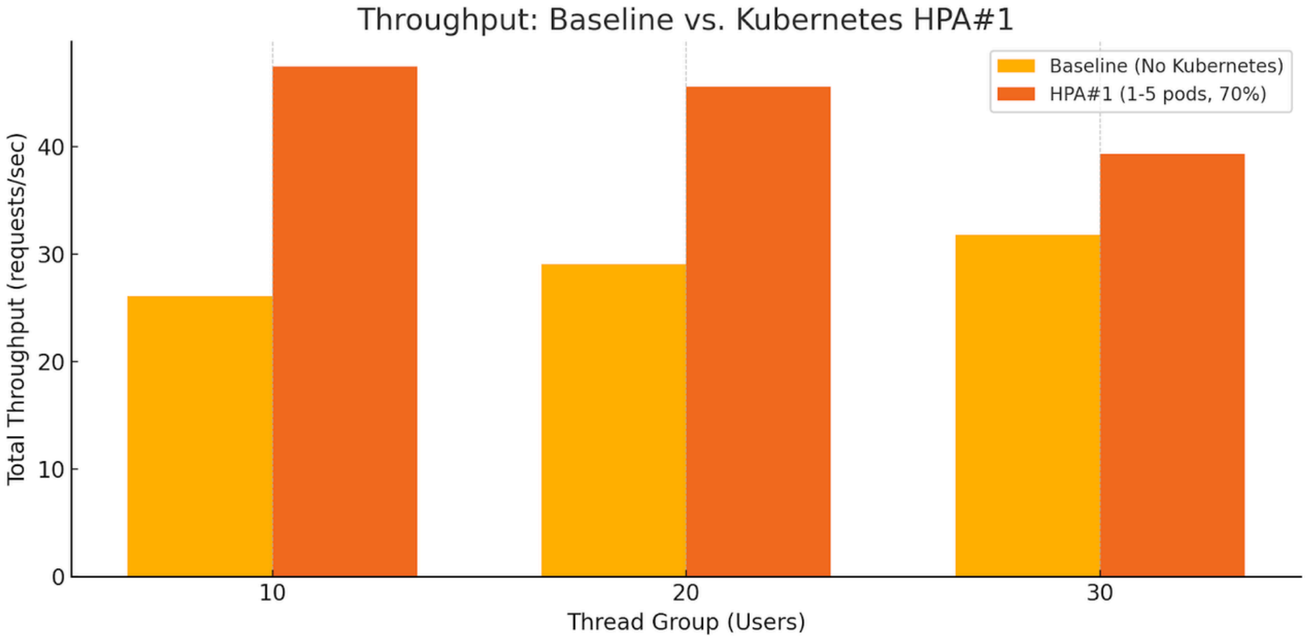
Load Testing Overview:

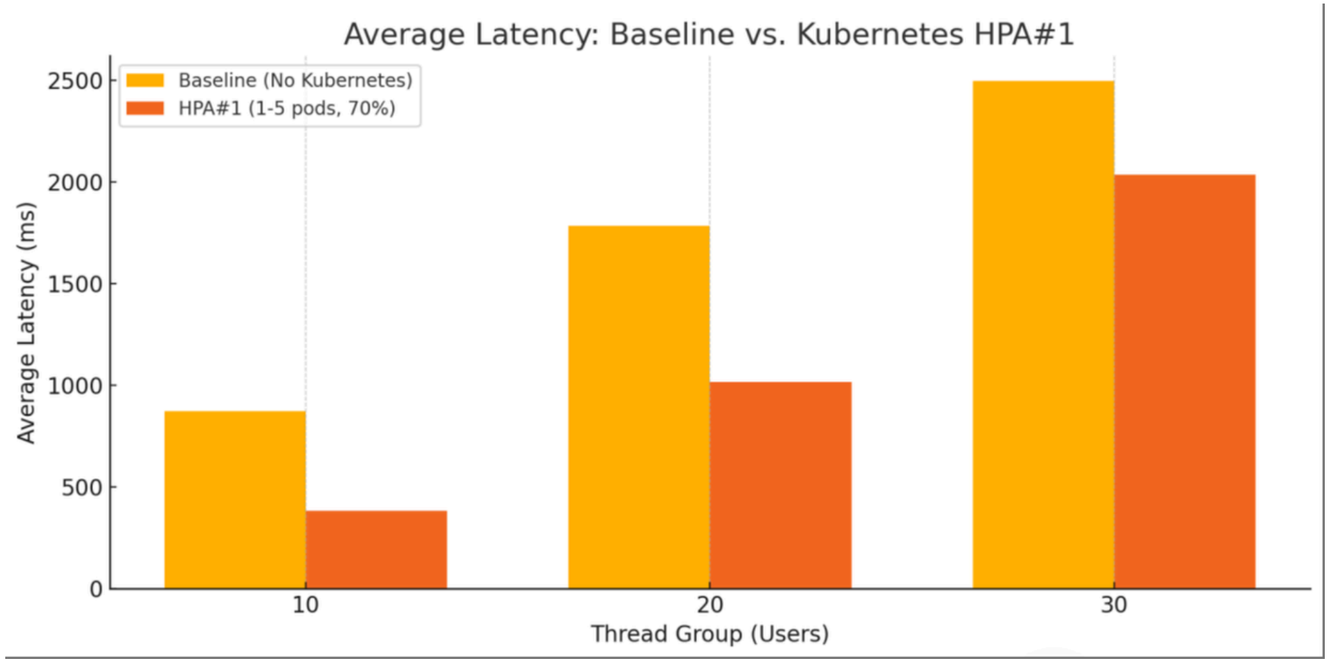
- We used JMeter to simulate different user loads.
- Each user sent 100 POST requests for resize, watermark, and filter functions. We tested with 10, 20, and 30 users. In regular tests, users ramped up over 30 seconds.
- Metrics:
  - Average latency
  - 90th/99th percentile latency
  - Throughput (requests/sec)
  - Failed request percentage

RESULTS & ANALYSIS

Baseline v.s. Kubernetes (HPA#1):

- Baseline server hit throughput limits early
- Kubernetes (HPA#1) scaled effectively with higher throughput
- Kubernetes better handled concurrency under pressure
- The system with Kubernetes has higher throughput and lower latency.



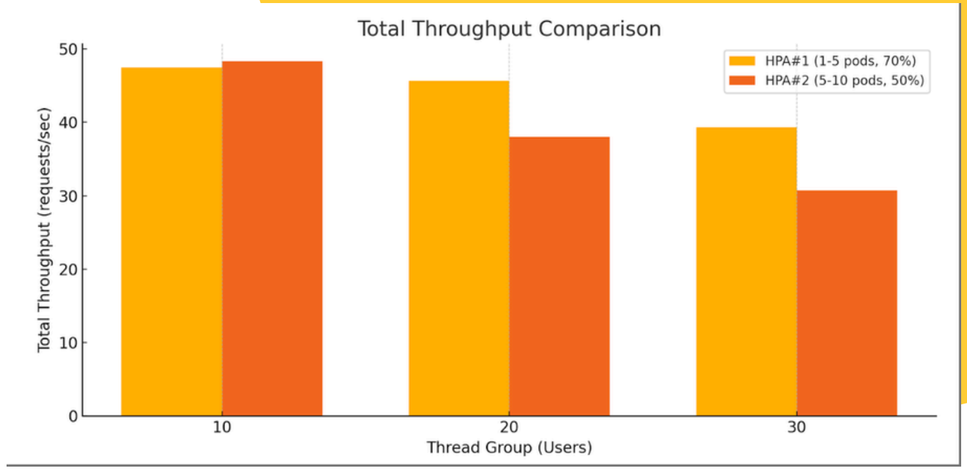


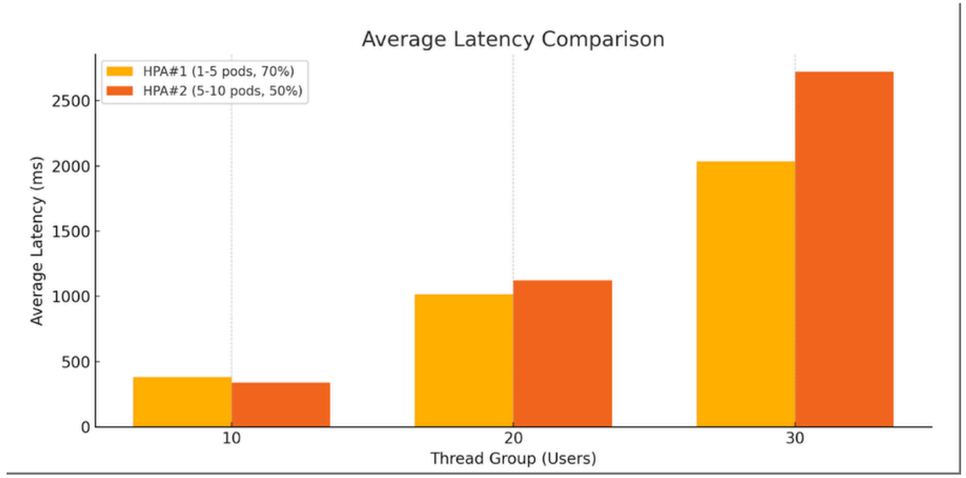
RESULTS & ANALYSIS

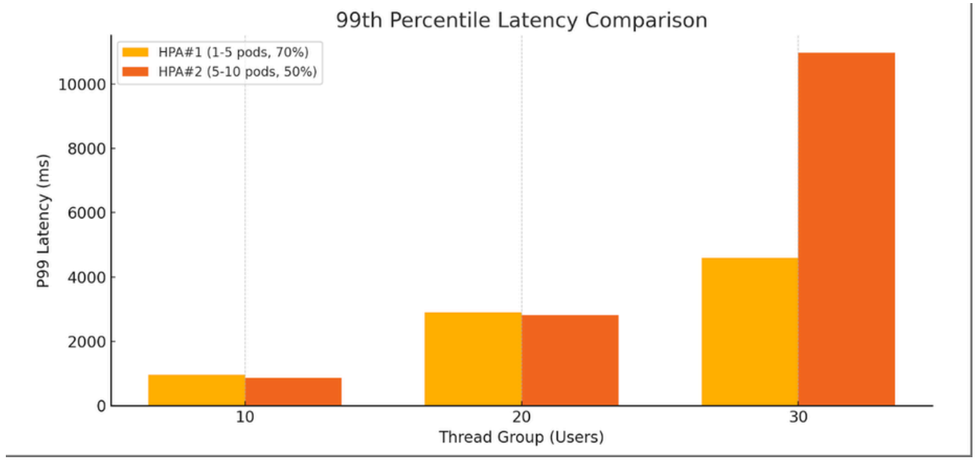
Comparing different rules of HPA:

- HPA#1: 1-5 replicas, 70% CPU threshold
- HPA#2:5-10 replicas, 50% threshold (more aggresive)

- HPA#1 had slightly better throughput at high load
- HPA#2's frequent scaling caused pod churn and delays
- Having more pods doesn't guarantee better performance if they're not fully initialized.





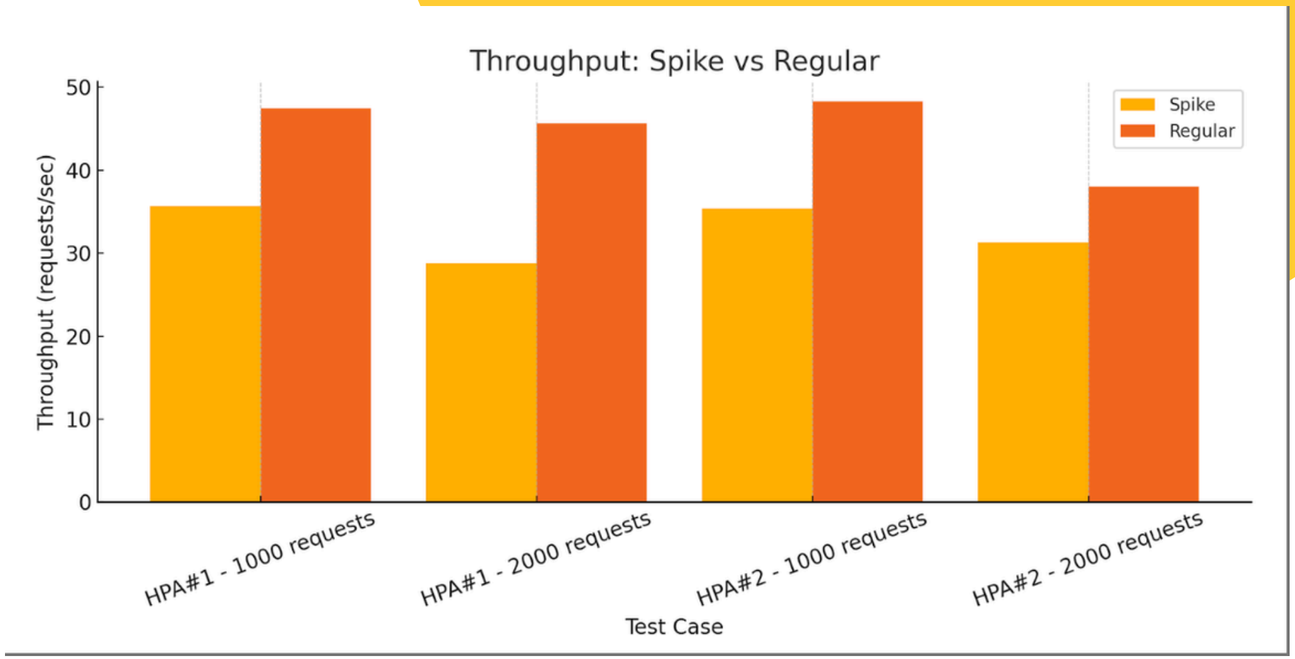


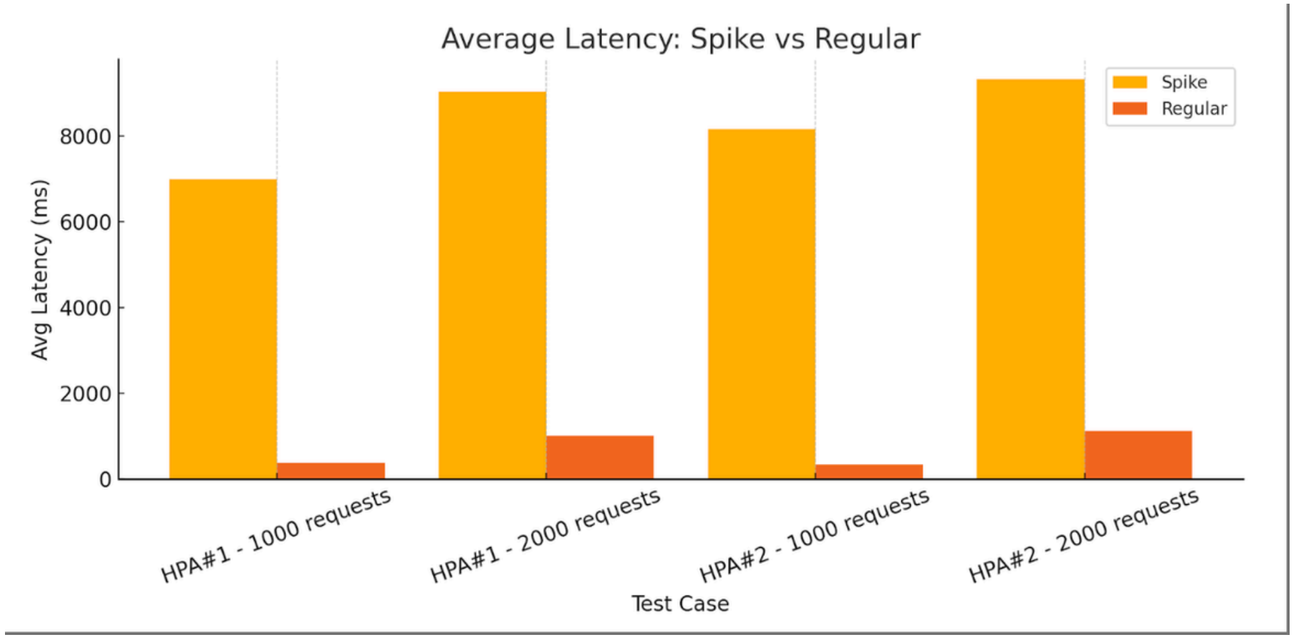
RESULTS & ANALYSIS

Spike Testing :

- Sudden traffic bursts,
  0 ramp-up time

- Throughput:
  The regular tests performed better.

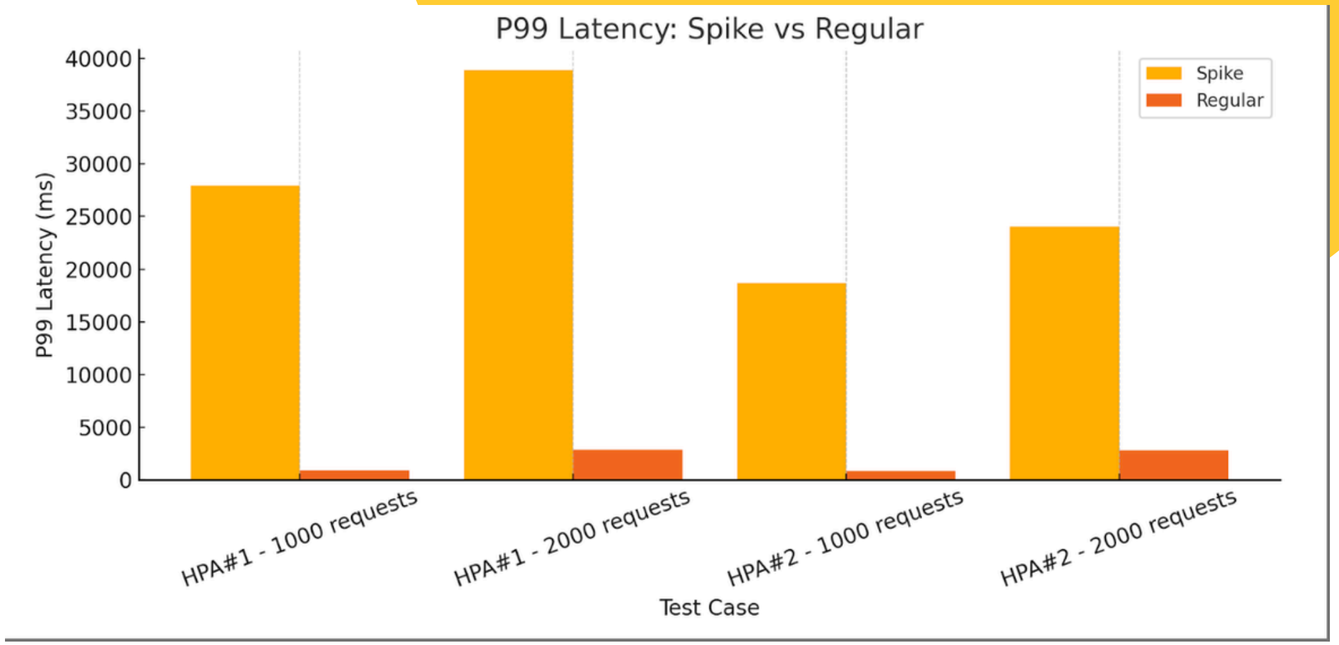


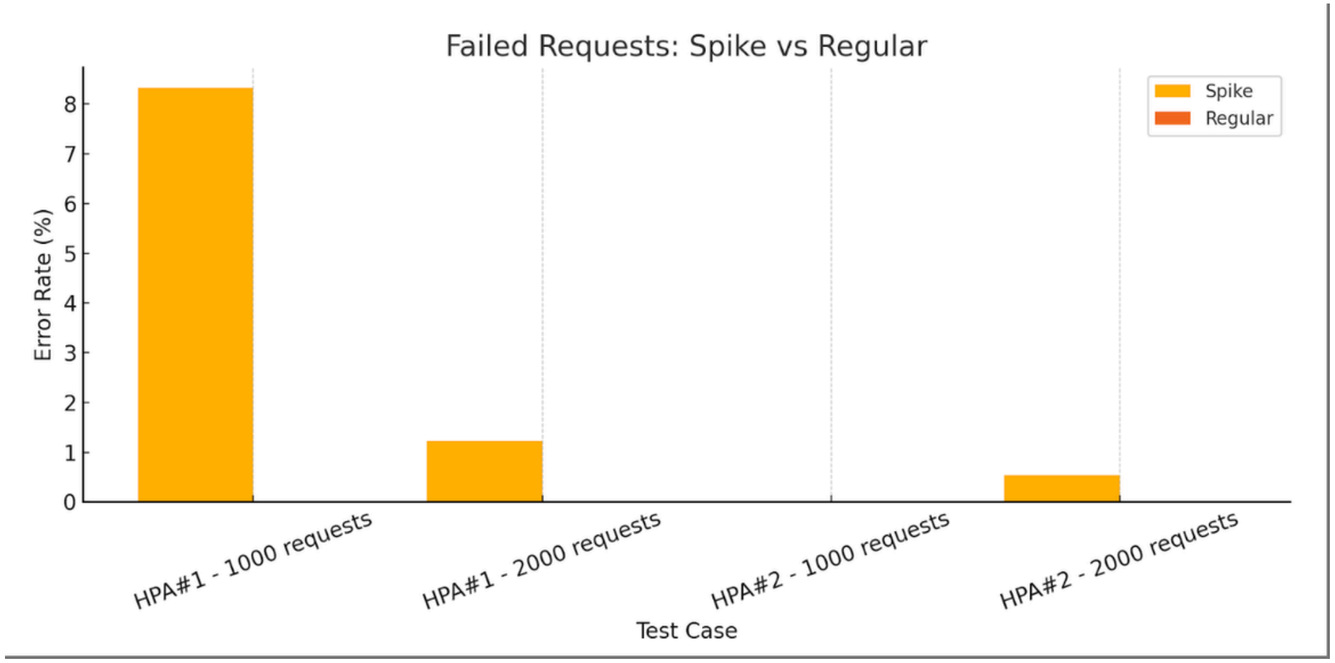


RESULTS & ANALYSIS

Spike Testing :

- HPA#2 handled spikes better with lower latency, near 0% errors
- HPA#1 failed ~8% of requests due to pod startup lag
- Pre-warming pods maybe helpful





RESULTS & ANALYSIS

Key Bottlenecks & Future Improvement:

- Cold starts slow response—keep warm pods
- POST ops are CPU-bound—optimize processing logic
- Larger-size S3 bucket to store image
- Autoscaler tuning essentials for stability

FUTURE DIRECTIONS

- **Support batch jobs and chained processing**:
  - Extend functionality by allowing batch image uploads and multi-step transformations in a single workflow (e.g., resize → filter → watermark).
- **Deploy on AWS EKS or GKE**
  - Moving from Minikube to a managed Kubernetes platform like **AWS EKS** or **Google Kubernetes Engine (GKE)** would enable real-world scalability testing. It also offers access to production-grade services (IAM, CloudWatch, real S3) and better reflects enterprise deployment environments.
- **Implement CI/CD with GitHub Actions or Jenkins**

CONCLUSION

- Built a modular, autoscalable, fault-tolerant image processing system
- Validated its performance with simulated workloads
- Learned valuable lessons in container orchestration, autoscaling, and benchmarking
- Ready to evolve into a production-grade pipeline

THANK YOU