SCALABLE IMAGE PROCESSING PLATFORM WITH KUBERNETES

Presented By : Yuxin Wang
Jingwen Huang
Lan Wang

TABLE OF CONTENT

INTRODUCTION

Traditional image processing systems suffer from:

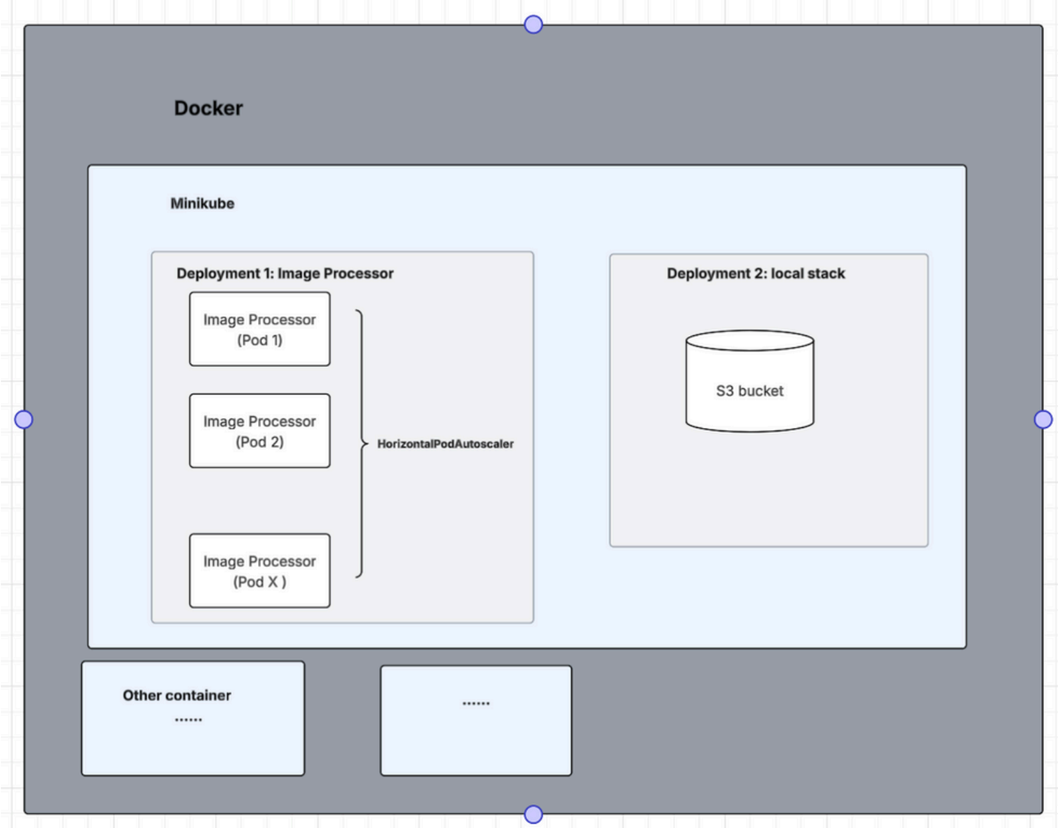
- ❌ High latency due to sequential task execution
- ❌ Poor scalability from monolithic designs
- ❌ Single points of failure, causing service downtime
- ❌ Inability to handle traffic spikes without autoscaling

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



- **Docker Environment**
  - The entire system runs within Docker environment
- **Minikube**
  - Minikube provides a local Kubernetes cluster for development. It runs inside Docker and manages the various deployments, services, and other Kubernetes resources.
- **Image Processor Deployment**
  - Core application deployment that handles the image processing workload:
  - **Multiple Pods**: The deployment creates multiple replica pods that handle incoming image processing requests.
  - **Horizontal Pod Autoscaler**: Automatically adjusts the number of pods based on CPU utilization, scaling up during high demand and down during low activity periods.
  - **Resource Management**: Each pod has defined resource limits and requests for CPU and memory.
- **LocalStack Deployment**
  - **S3 Bucket**: Stores the processed images, mimicking AWS S3 storage.
  - **LocalStack Service**: Exposes the S3 API on port 4566, allowing the Image Processor pods to interact with it using standard AWS SDK.

APPLICATION ARCHITECTURE

SPRING BOOT REST API

The application is built using Spring Boot and provides several RESTful endpoints:

- /api/images/health: Health check endpoint
- /api/images/upload/resize: Image resizing endpoint
- /api/images/upload/watermark: Image watermarking endpoint
- /api/images/upload/filter: Image filtering endpoint
- /api/images/{imageKey}: Image retrieval endpoint

STORAGE INTEGRATION

The system uses AWS S3 SDK to interact with LocalStack:

- Images are uploaded using the S3 client
- Processed images are stored with unique UUIDs
- Image retrieval is handled through S3 GetObject operations

APPLICATION ARCHITECTURE

POST TO WATERMARK

K8s / **watermark image**

Save | Sha

POST | {{url}} /api/images/upload/watermark | Send

Params | Authorization | Headers (9) | Body | Scripts | Tests | Settings | Coo

none | form-data | x-www-form-urlencoded | raw | binary | GraphQL

Key		Value	Description
file	File	humming2.jpeg	
text	Text	any watermark	
position	Text	center	
Key	Text	Value	Description

Body | Cookies | Headers (5) | Test Results | 200 OK | 83 ms | 459 B | Save Response

{ } JSON | Preview | Visualize


```
{
    "success": true,
    "message": "Image watermarked successfully",
    "imageUrl": "http://localhost:4566/images-bucket/34439992-bc93-4771-9232-c922cf8cb7bc-humming2.
        jpeg",
    "originalName": "humming2.jpeg",
    "timestamp": "2025-04-24T00:15:15.172521",
    "imageKey": "34439992-bc93-4771-9232-c922cf8cb7bc-humming2.jpeg"
}
```


GET PROCESSED IMAGE BY ID

K8s / **get image**

GET | {{url}} /api/images/34439992-bc93-4771-9232-c922cf8cb7bc-humming2.jpeg

Params | Authorization | Headers (7) | Body | Scripts | Tests | Settings

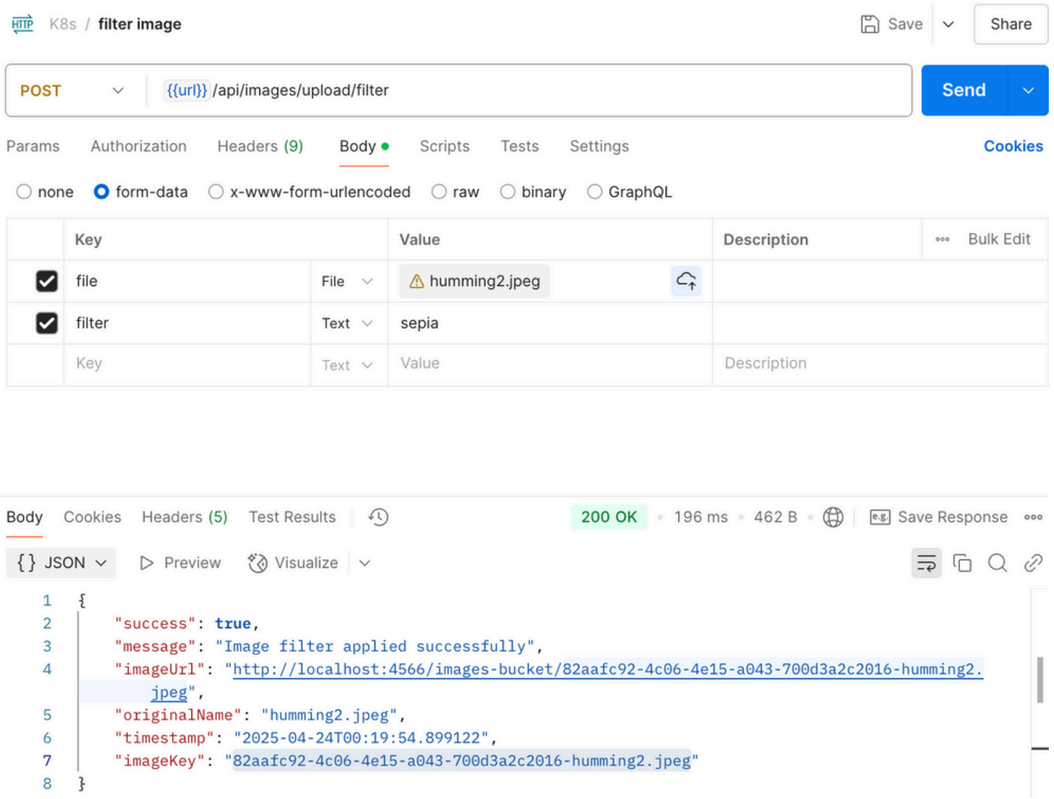
none | form-data | x-www-form-urlencoded | raw | binary | GraphQL

This request does not have a body

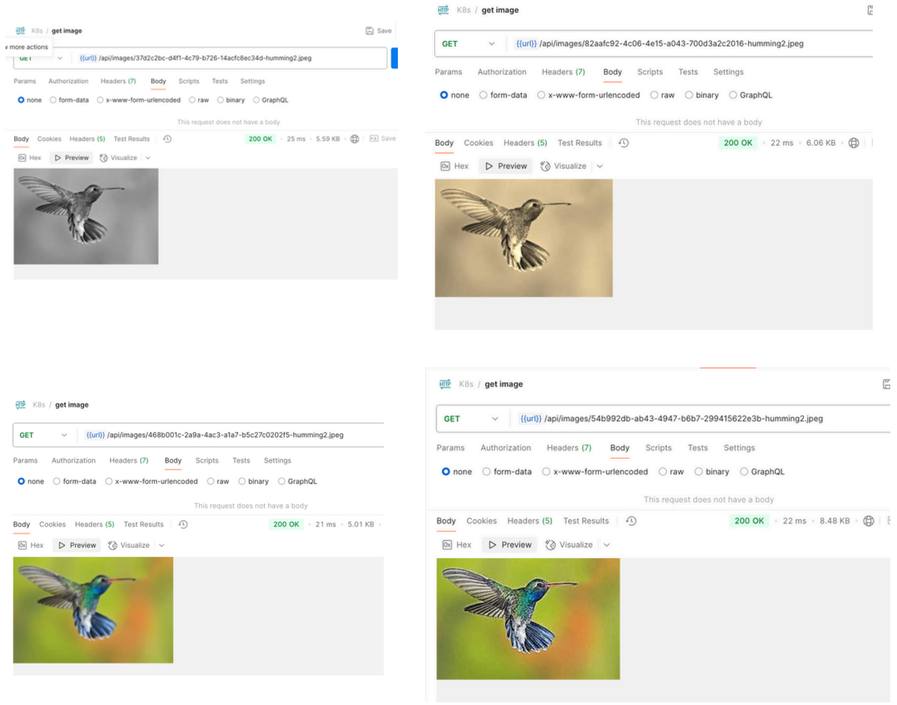
Body | Cookies | Headers (5) | Test Results | 200 OK | 22 ms | 6.71 KB

Hex | Preview | Visualize

APPLICATION ARCHITECTURE



POST TO ADD FILTER



DIFFERENT FILTER RESULTS

WORKFLOWS

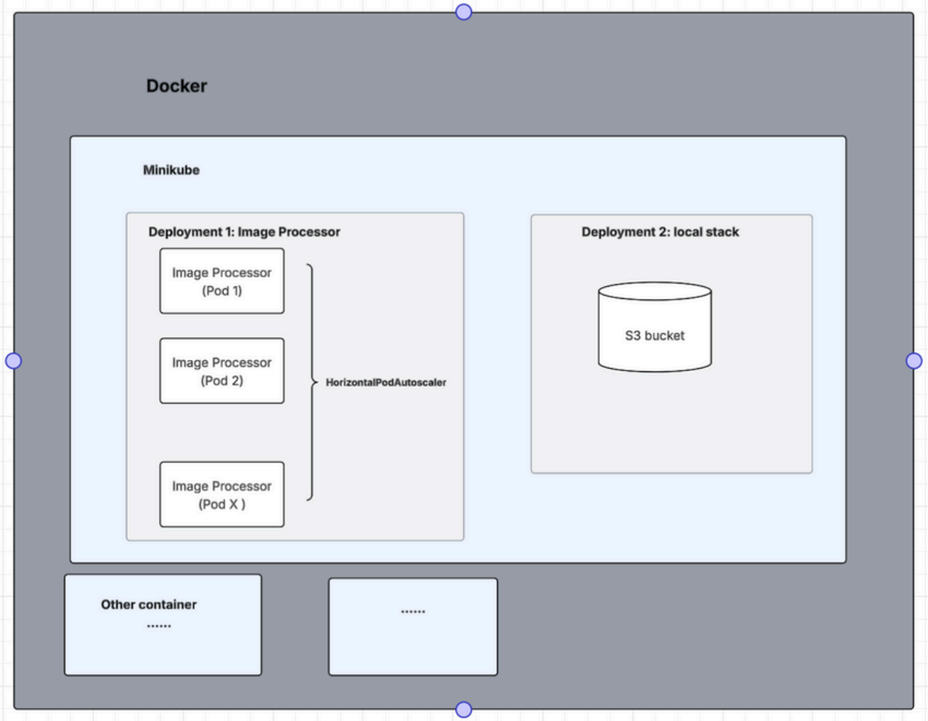


IMAGE PROCESSING WORKFLOW

- Client sends an image to one of the processing endpoints
- The request is routed to an available Image Processor pod
- The pod processes the image according to the requested operation
- The processed image is stored in the S3 bucket
- A response with the image URL and key is returned to the client

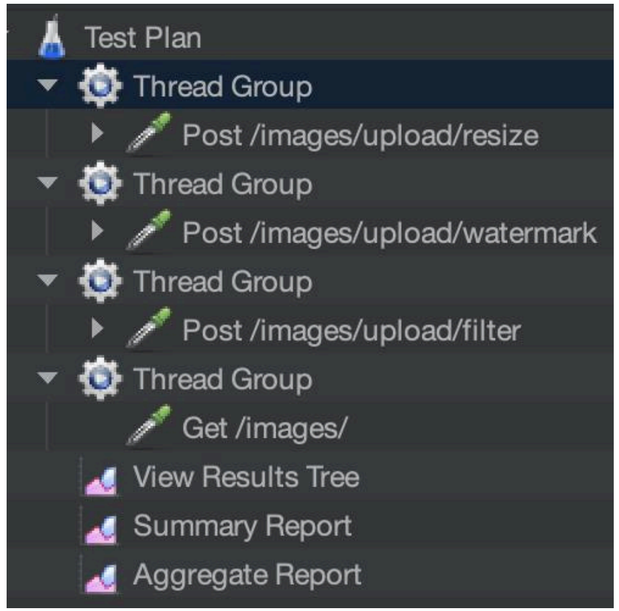
AUTOSCALING WORKFLOW

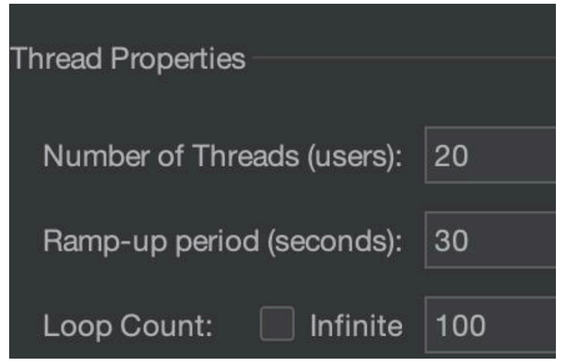
- As request volume increases, pod CPU utilization rises
- The Horizontal Pod Autoscaler monitors CPU metrics
- When utilization exceeds the target threshold (e.g., 70%), additional pods are created
- When demand decreases, excess pods are terminated

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



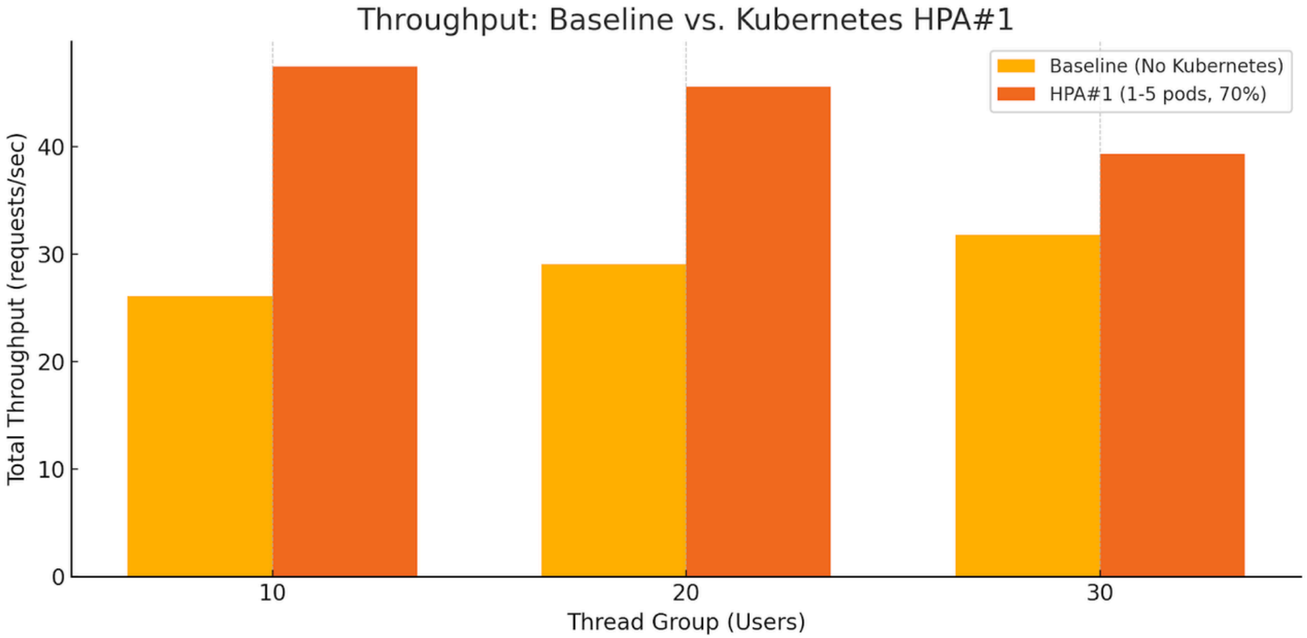


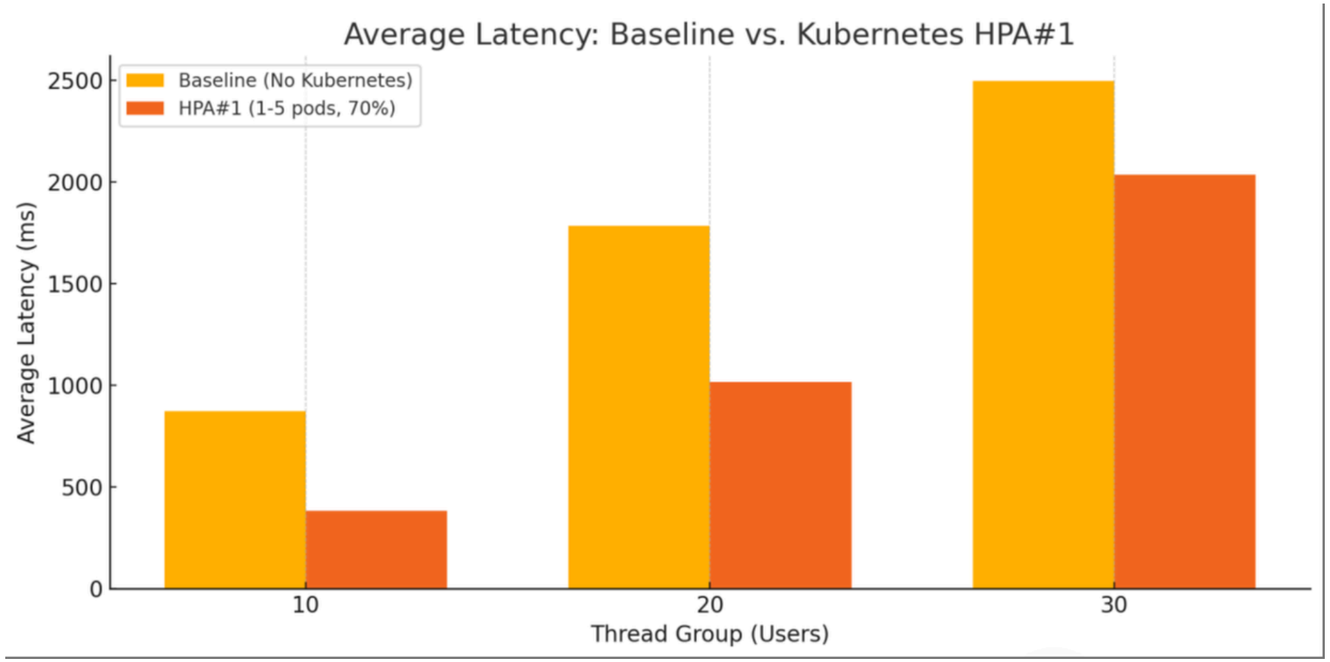
Label	# Samples	Average	Median	90% Line	95% Line	99% Line	Min	Max	Error %	Throughput	Received KB/sec	Sent KB/sec
Get /images/	1000	153	153	275	309	471	2	770	0.000%	21.52528	120.27	3.74
Post /images/upload/resize	1000	874	810	1461	1696	2323	48	2941	0.000%	8.70413	3.90	25.24
Post /images/upload/watermark	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

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.



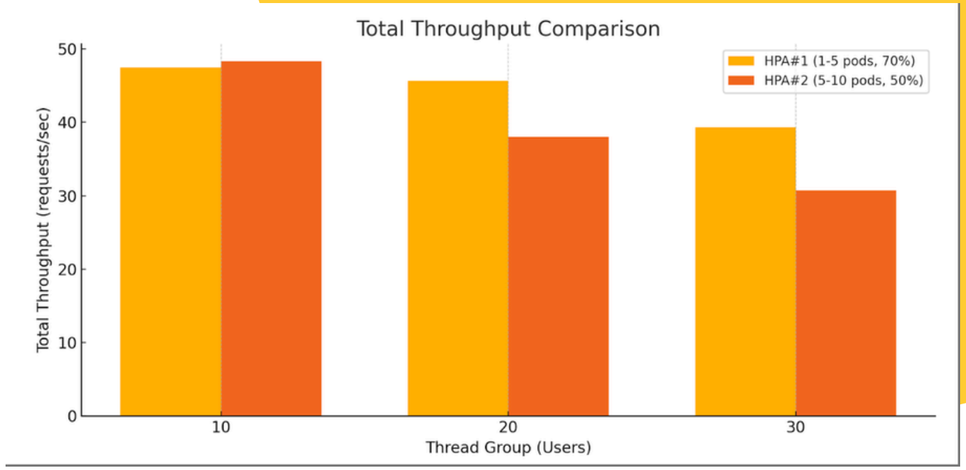


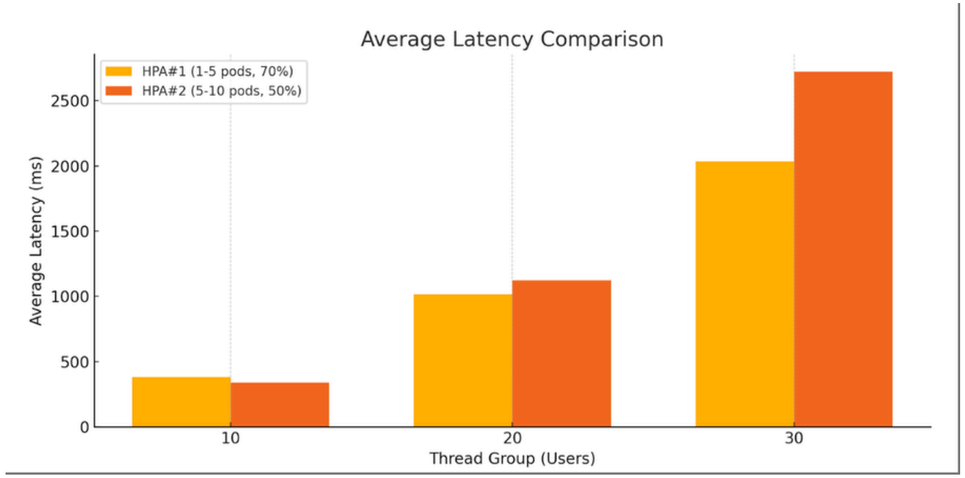
RESULTS & ANALYSIS

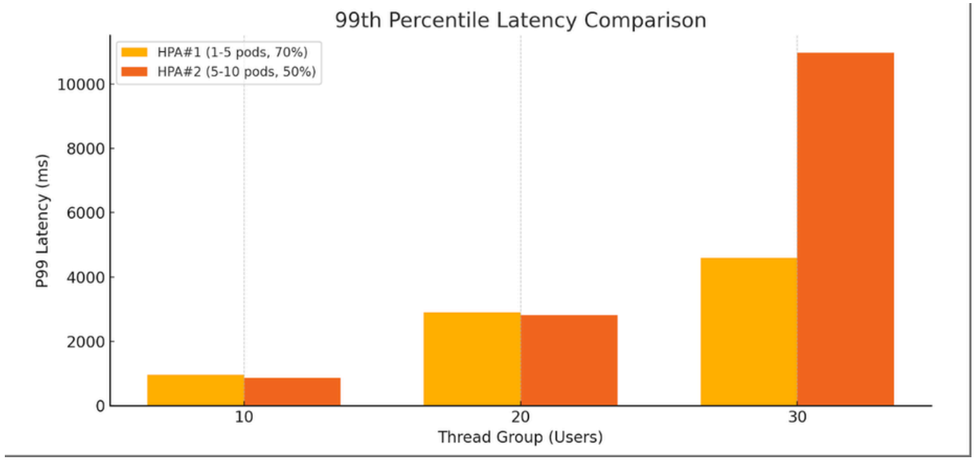
Comparing different rules of HPA:

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

- 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.





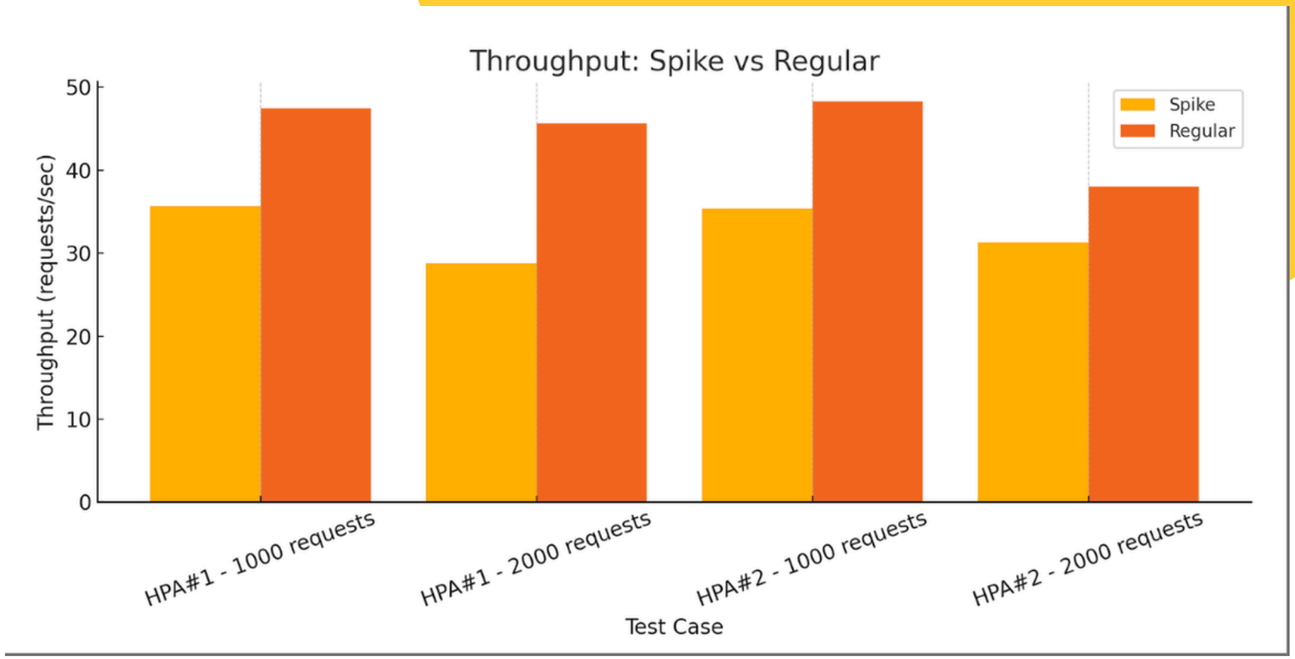


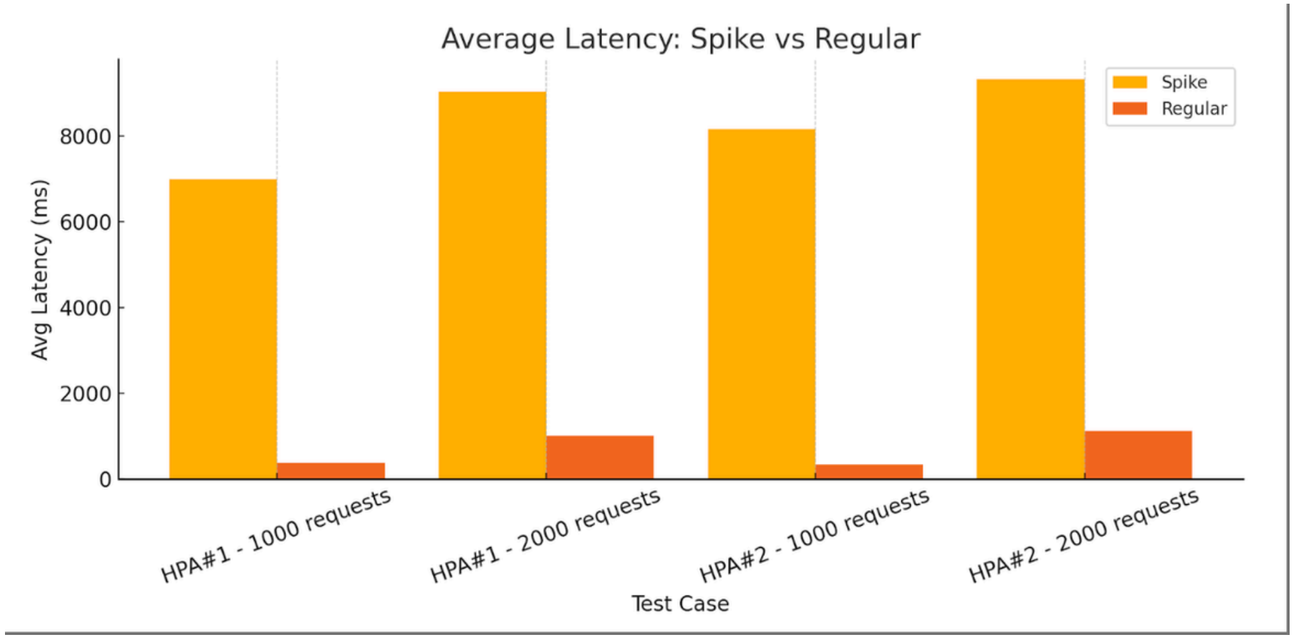
RESULTS & ANALYSIS

Spike Testing :

- Sudden traffic bursts,
  0 ramp-up time

- Throughput:
  The regular tests performed better.

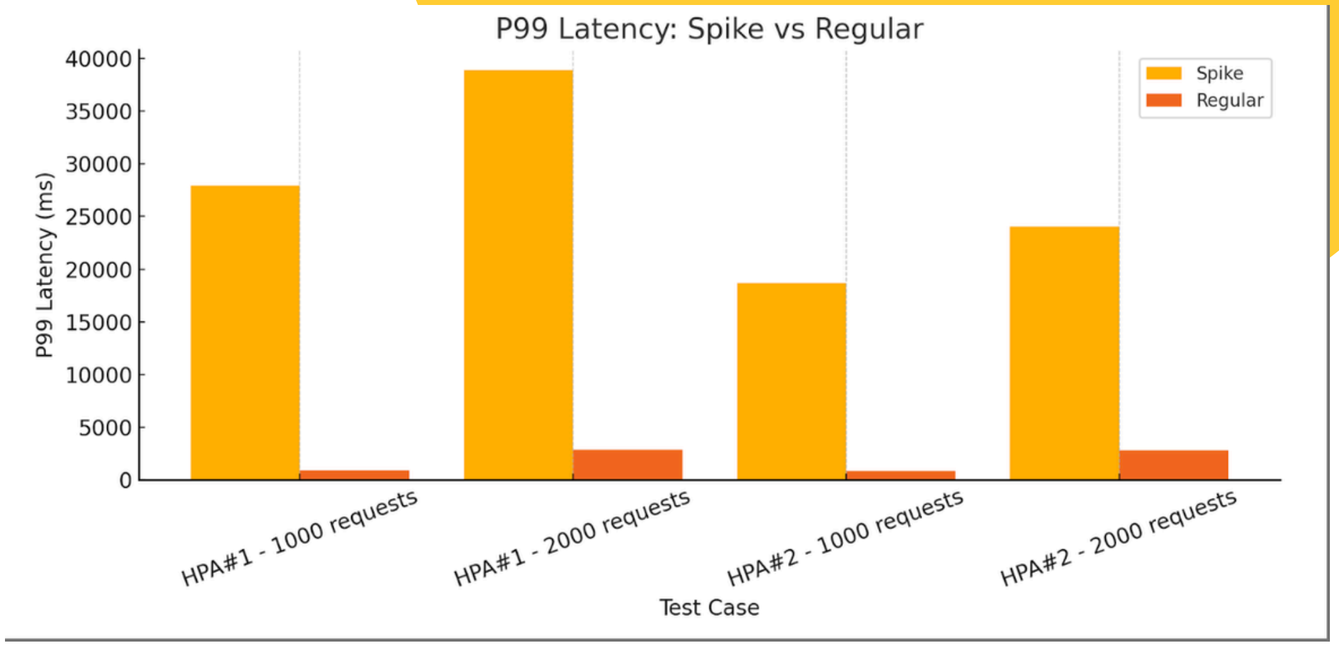


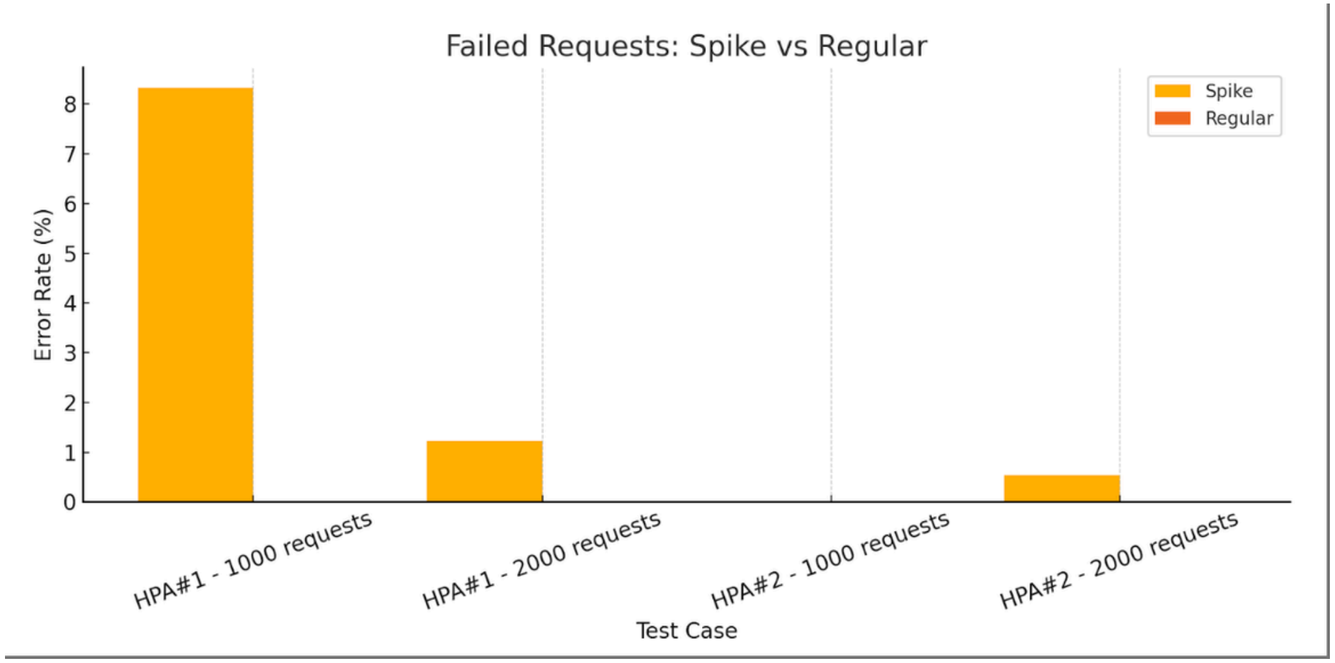


RESULTS & ANALYSIS

Spike Testing :

- HPA#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
- CPU power limited
- 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 Elastic Kubernetes Service (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**
- **Advanced monitoring with Prometheus/Grafana**
- **Additional image processing features**

THANK YOU