# Elastic ML Inference Serving: Custom Autoscaling vs Kubernetes HPA

Muhammad Ozair (70135) Kamil Hassaan (66447)
Burhan Ahmed Khanzada (70379)
Mohmmadjafar Akif Haidar Jalali (72895)

June 30, 2025

| | |
|---|---|
| Type of work: | Cloud Computing Project Report |
| Students: | Muhammad Ozair, Kamil Hassaan, Mohmmadjafar Akif Haidar Jalali and Burhan Ahmed Khanzada |
| Supervisor: | Mehran Salmani |
| Department: | Distributed Systems and Operating Systems |
| Semester: | Summer Semester 2025 |

**Keywords:** Kubernetes; Autoscaling; Machine Learning Inference; ResNet18; HPA; Custom Autoscaler; FastAPI; Producer-Consumer Pattern; Queue Management

# Contents


**1 Introduction** **4**
- 1.1 Problem Statement . . . 4

**2 System Design and Architecture** **5**
- 2.1 Overall Architecture . . . 5
- 2.2 Component Design . . . 5
  - 2.2.1 ML Inference Service . . . 5
  - 2.2.2 Dispatcher Service Implementation . . . 6
  - 2.2.3 Custom Autoscaler Implementation . . . 7
- 2.3 Containerization and Deployment . . . 9
  - 2.3.1 Docker Configuration . . . 9
  - 2.3.2 Kubernetes Deployment . . . 9

**3 Implementation Details** **10**
- 3.1 Load Testing Framework . . . 10
- 3.2 Monitoring and Metrics . . . 10
  - 3.2.1 Custom Metrics . . . 10
  - 3.2.2 Prometheus Configuration . . . 10

**4 Experiments** **11**
- 4.1 Classification Accuracy vs Request Rate . . . 11
- 4.2 Experimental Design . . . 11
- 4.3 Experimental Parameters . . . 12
- 4.4 Metrics Collection . . . 12

**5 Results and Analysis** **13**
- 5.1 Experimental Results Overview . . . 13
- 5.2 Custom Autoscaler Performance . . . 13
- 5.3 HPA with 70% CPU Target . . . 14
- 5.4 HPA with 90% CPU Target . . . 15
- 5.5 Comparative Analysis . . . 15
  - 5.5.1 Comparison of Request Through Each Autoscaler . . . 15
  - 5.5.2 Latency Performance . . . 16
  - 5.5.3 Resource Utilization . . . 16
- 5.6 Key Findings . . . 16
  - 5.6.1 Custom Autoscaler Advantages . . . 16
  - 5.6.2 HPA Limitations . . . 16

**6 Discussion** **17**
- 6.1 Technical Contributions . . . 17
  - 6.1.1 Queue-Based Autoscaling . . . 17
  - 6.1.2 Producer-Consumer Architecture . . . 17
  - 6.1.3 Comprehensive Monitoring . . . 17

6.2 Performance Analysis . . . . . 17
6.2.1 Latency Achievement . . . . . 17
6.2.2 Resource Efficiency . . . . . 17
6.3 Limitations and Future Work . . . . . 18
6.3.1 Current Limitations . . . . . 18
6.3.2 Future Enhancements . . . . . 18

**7 Conclusion** **19**
7.1 Lessons Learned . . . . . 19

# 1 Introduction

This report presents the design, implementation, and evaluation of an elastic machine learning inference serving system deployed on Kubernetes. The system provides real-time image classification using ResNet18 and features a custom autoscaling mechanism that is compared against Kubernetes' built-in Horizontal Pod Autoscaler (HPA).

The primary objectives of this project are:

- Design and implement a scalable containerized ML inference system
- Develop a custom autoscaling algorithm based on queue metrics
- Achieve server-side latency of less than 0.5 seconds for inference queries
- Compare the performance of the custom autoscaler against Kubernetes HPA
- Analyze system behavior under varying workload patterns

## 1.1 Problem Statement

Machine learning inference services in production environments face several challenges:

- **Variable workload patterns**: Request rates can vary significantly over time
- **Resource efficiency**: Over-provisioning leads to wasted resources, under-provisioning causes poor performance
- **Latency requirements**: Modern applications require sub-second response times
- **Cost optimization**: Cloud resources should scale efficiently with demand

Traditional autoscaling approaches like Kubernetes HPA rely primarily on CPU and memory utilization metrics, which may not accurately reflect the actual service demand, especially for ML workloads that involve queuing and asynchronous processing.

# 2 System Design and Architecture

## 2.1 Overall Architecture

The system implements a distributed architecture with clear separation of concerns, featuring six main components:

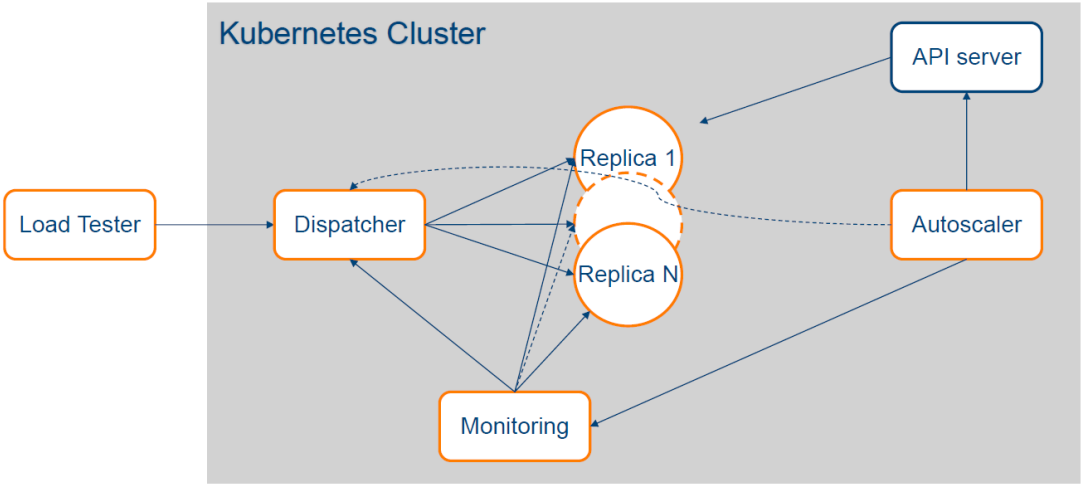


Figure 2.1: System architecture: Load Tester to ML Inference Service

Figure 2.2: High-level system architecture

## 2.2 Component Design

### 2.2.1 ML Inference Service

The ML inference service is a FastAPI application that serves the ResNet18 model for image classification. Key design decisions include:

- **Model Pre-loading**: ResNet18 weights are downloaded during Docker build time to reduce cold start latency
- **CPU Optimization**: The service is configured to run on CPU with optimized threading settings
- **Monitoring Integration**: Prometheus metrics are exposed for system monitoring
- **Resource Constraints**: Each pod is limited to 1 CPU core and 1GB memory

```python
import psutil
import logging
import asyncio
import time
import io
import threading

from prometheus_client import Counter, Gauge, Histogram
from prometheus_client.exposition import start_http_server

from fastapi import FastAPI, UploadFile, Request
from resnet_inference import ModelInference
from PIL import Image

# Object of ModelInference class
model_inference = ModelInference()
app = FastAPI()
start_http_server(9001)

# Define metrics
REQUEST_COUNT = Counter('ml_app_requests_total', 'Total HTTP requests',
                        ['method', 'endpoint', 'status'])
CPU_USAGE = Gauge('ml_app_cpu_usage_percent', 'CPU usage percentage')
MEMORY_USAGE = Gauge('ml_app_memory_usage_percent', 'Memory usage percentage')
RESPONSE_TIME = Histogram('ml_app_response_time_seconds',
                          'Request response time in seconds', ['endpoint'])

@app.post("/predict")
async def predict(image: UploadFile):
    """
    This is a post request async function for model inferencing.
    """
    try:
        contents = await image.read()
        image = Image.open(io.BytesIO(contents))
        preprocessed_image = model_inference.transform_image(image)
        prediction = model_inference.predict(preprocessed_image)
        return {'prediction': prediction}
    except Exception as e:
        return {'Error': e}
```

Listing 2.1: ML Inference Service Implementation

### 2.2.2 Dispatcher Service Implementation

The dispatcher implements a producer-consumer pattern with asynchronous queue management:

```python
import asyncio
import aiohttp
import uuid
from fastapi import FastAPI, UploadFile
from dispatcher import Dispatcher

dispatcher = Dispatcher()
app = FastAPI()
pending_requests = {}
pending_requests_lock = asyncio.Lock()

@app.post("/add_to_queue")
async def request_queue(image: UploadFile):
```

```
    """PRODUCER: Receives requests and waits for results"""
    request_id = str(uuid.uuid4())

    await dispatcher.add_to_queue(image, request_id)
    queue_size = await dispatcher.qsize()

    # Wait for background worker to process request
    future = asyncio.Future()
    async with pending_requests_lock:
        pending_requests[request_id] = future

    try:
        predictions = await asyncio.wait_for(future, timeout=60)
        return {'prediction': predictions, 'queue_size': queue_size}
    except asyncio.TimeoutError:
        async with pending_requests_lock:
            pending_requests.pop(request_id, None)
        return {'error': 'Request timeout', 'queue_size': queue_size}

async def consumer_worker(worker_id: int):
    """CONSUMER: Background worker processing requests"""
    while workers_running:
        try:
            result, request_id = await get_inference()

            async with pending_requests_lock:
                future = pending_requests.pop(request_id, None)
                if future and not future.done():
                    future.set_result(result)
        except Exception as e:
            # Handle errors and resolve pending requests
            pass
        await asyncio.sleep(0.1)
```

Listing 2.2: Dispatcher Service Core Components

- **Request Queue**: AsyncIO queue for managing incoming inference requests
- **Worker Pool**: 10 background workers for processing requests concurrently
- **Load Balancing**: Kubernetes native load balancing
- **Connection Pooling**: Shared HTTP client with optimized connection management

The dispatcher serves as the entry point for all inference requests and provides queue size metrics crucial for the custom autoscaling algorithm.

### 2.2.3 Custom Autoscaler Implementation

The custom autoscaler implements a queue-based scaling algorithm:

```
import asyncio
import httpx
import math
from kubernetes import client, config

# Configuration constants
PROMETHEUS_URL = "http://prometheus-operated.monitoring.svc:9090"
DEPLOYMENT_NAME = "ml-app-deployment"
```

```
NAMESPACE = "default"
POLL_INTERVAL = 15
COOLDOWN_SECONDS = 150
MIN_REPLICAS = 1
MAX_REPLICAS = 6
DESIRED_QSIZE = 50

async def get_metric(query):
    """Get queue size from Prometheus."""
    try:
        async with httpx.AsyncClient() as client:
            response = await client.get(
                f"{PROMETHEUS_URL}/api/v1/query",
                params={"query": query},
                timeout=5
            )
            result = response.json()
            if result["status"] == "success" and result["data"]["result"]:
                return float(result["data"]["result"][0]["value"][1])
            return None
    except Exception as e:
        logger.error(f"Error fetching metric: {e}")
        return None

async def scale_deployment(qsize, v1_api):
    """Scale deployment based on queue size."""
    global last_scale_time
    if time.time() - last_scale_time < COOLDOWN_SECONDS:
        return

    # Get current replica count
    deployment = apps_v1.read_namespaced_deployment(DEPLOYMENT_NAME, NAMESPACE)
    current_replicas = deployment.spec.replicas

    # Calculate desired replicas based on queue size
    desired_replicas = current_replicas
    if qsize is not None:
        if qsize == 0:
            desired_replicas = MIN_REPLICAS
        elif qsize > DESIRED_QSIZE:
            # Scale up for high queue sizes
            desired_replicas = math.ceil(current_replicas * (qsize /
                DESIRED_QSIZE))
        elif qsize <= DESIRED_QSIZE:
            # Gradual downscaling for low queue sizes
            desired_replicas = max(MIN_REPLICAS,
                                   math.ceil(current_replicas * (qsize /
                                       DESIRED_QSIZE)))

        # Ensure replicas stay within bounds
        desired_replicas = max(MIN_REPLICAS, min(MAX_REPLICAS, desired_replicas)
            )

    # Apply scaling if needed
    if desired_replicas != current_replicas:
        apps_v1.patch_namespaced_deployment_scale(
            name=DEPLOYMENT_NAME,
            namespace=NAMESPACE,
            body={"spec": {"replicas": desired_replicas}}
        )
        last_scale_time = time.time()
```

Listing 2.3: Custom Autoscaler Implementation

Key features of the custom autoscaler:

- **Queue-based Metrics**: Uses dispatcher queue size instead of CPU utilization
- **Cooldown Period**: 150-second cooldown to prevent oscillation
- **Pod Readiness Checks**: Ensures all pods are ready before scaling
- **Graceful Scaling**: Proportional scaling based on queue load

## 2.3 Containerization and Deployment

### 2.3.1 Docker Configuration

Each component is containerized with optimized Dockerfiles:

- **Model pre-downloading**: ResNet18 weights cached during build
- **Environment optimization**: Threading and memory configurations

### 2.3.2 Kubernetes Deployment

The system is deployed on Kubernetes with the following configurations:

- **ML Service**: Deployed with resource limits
- **Dispatcher**: Single replica with high memory allocation for queue management
- **Custom Autoscaler**: Deployed with appropriate RBAC permissions
- **Monitoring**: Prometheus ServiceMonitors for metrics collection

# 3 Implementation Details

## 3.1 Load Testing Framework

A custom load testing framework was developed based on the BarAzmoon library to simulate realistic workload patterns:

- **Image Dataset**: ImageNet sample images in various formats (PNG, JPG, JPEG)
- **Workload Patterns**: Configurable request rates from workload.txt
- **Statistics Tracking**: Classification accuracy and response time monitoring
- **Timeout Handling**: Robust error handling for network timeouts

The workload pattern includes:

- **Low Load Phase**: 5-10 requests per second (baseline)
- **Stress Phase**: 20-40 requests per second (scaling trigger)
- **Cool-down Phase**: Gradual return to baseline load

## 3.2 Monitoring and Metrics

The system implements comprehensive monitoring using Prometheus:

### 3.2.1 Custom Metrics

- **Queue Size**: Real-time dispatcher queue length
- **Request Count**: Total requests processed by each component
- **Response Time**: 99th percentile latency tracking
- **Resource Usage**: CPU and memory utilization per service

### 3.2.2 Prometheus Configuration

ServiceMonitors are configured to scrape metrics from both the ML service and dispatcher at 15-second intervals, providing high-resolution monitoring data for analysis.

# 4 Experiments

## 4.1 Classification Accuracy vs Request Rate

To further evaluate the limits of the ML inference system under high load, we ran isolated tests with a single replica handling varying request rates over different time intervals. The goal was to observe when classification accuracy begins to degrade.

The tests were executed on a Virtual Machine running Ubuntu, configured with 6 GB of RAM and 5 CPU cores. This setup served to simulate a resource-constrained environment, offering realistic insight into how a single ML inference replica performs under varying request loads.

| **Time Interval (s)** | **1 rps** | **2 rps** | **3 rps** | **4 rps** | **5 rps** | **6 rps** |
|---|---|---|---|---|---|---|
| 100 | 100% | 100% | 100% | 100% | 94% | 74% |
| 200 | 100% | 100% | 100% | 100% | 83.5% | 75% |
| 300 | 100% | 100% | 100% | 100% | 88.33% | 47.83% |
| 400 | 100% | 100% | 100% | 100% | 73.9% | 41% |
| 500 | 100% | 100% | 100% | 99.4% | 54.68% | 24.43% |
| 650 | 100% | 100% | 100% | 96.83% | 75.14% | 20.92% |

Table 4.1: Classification success rates across different request rates and time intervals using a single ML service replica

As shown in Table 4.1, classification performance begins to degrade significantly after exceeding four requests per 400 seconds. The drop is especially steep for 5 and 6 requests as the system reaches saturation. This validates the autoscaler's need to respond before the system is overwhelmed.

Key observations from this analysis:

- **Stable Performance**: 1-3 requests maintain 100% accuracy across all time intervals
- **Degradation Threshold**: 4 requests show minor degradation at longer intervals (99.4% at 500s, 96.83% at 650s)
- **Significant Drops**: 5-6 requests experience substantial accuracy loss, especially at higher time intervals
- **System Saturation**: Performance becomes unpredictable beyond 4 requests, confirming the need for proactive scaling

## 4.2 Experimental Design

To evaluate and compare autoscaling strategies under a more capable deployment, this experiment was conducted on a physical machine running Arch Linux with the following specifications: **AMD Ryzen 7 5800X3D (16 cores @ 4.55GHz), 32 GB RAM, and NVIDIA RTX 3060 GPU**. The experiments were deployed using Minikube, configured with **14 CPU cores** and **16 GB RAM** allocated to the Kubernetes cluster.

Three separate experiments were conducted to compare autoscaling approaches:

1. **Custom Autoscaler**: Queue-based scaling with 50-request threshold
2. **HPA with 70% CPU target**: Standard Kubernetes HPA configuration
3. **HPA with 90% CPU target**: Aggressive CPU-based scaling

## 4.3 Experimental Parameters

| **Parameter** | **Value** |
|---|---|
| Min Replicas | 1 |
| Max Replicas | 6 |
| Resource Limit | 1 CPU core, 1GB RAM per pod |
| Target Latency | $< 0.5$ seconds |
| Workload Duration | 10.5 minutes |
| Monitoring Interval | 15 seconds |
| Custom Autoscaler Polling | 15 seconds |
| Custom Autoscaler Cooldown | 150 seconds |
| HPA Scale-up Policy | 100% increase, max 4 pods per 15s |
| HPA Scale-down Policy | 10% decrease per 60s |

Table 4.2: Experimental configuration parameters

## 4.4 Metrics Collection

The following metrics were collected during each experiment:

- **Service 99th percentile latency**: End-to-end response time
- **CPU core utilization**: Total CPU cores used by ML service pods
- **Queue size**: Dispatcher queue length over time
- **Pod count**: Number of active ML service replicas
- **Request success rate**: Percentage of successful classifications

# 5 Results and Analysis

All results presented in this chapter are based on experiments conducted on the Arch Linux system described in Section 4.2. The system was configured using Minikube with 14 CPU cores and 16 GB RAM allocated to the Kubernetes cluster.

## 5.1 Experimental Results Overview

The experiments demonstrate significant differences in scaling behavior and performance characteristics between the custom autoscaler and Kubernetes HPA implementations.

## 5.2 Custom Autoscaler Performance

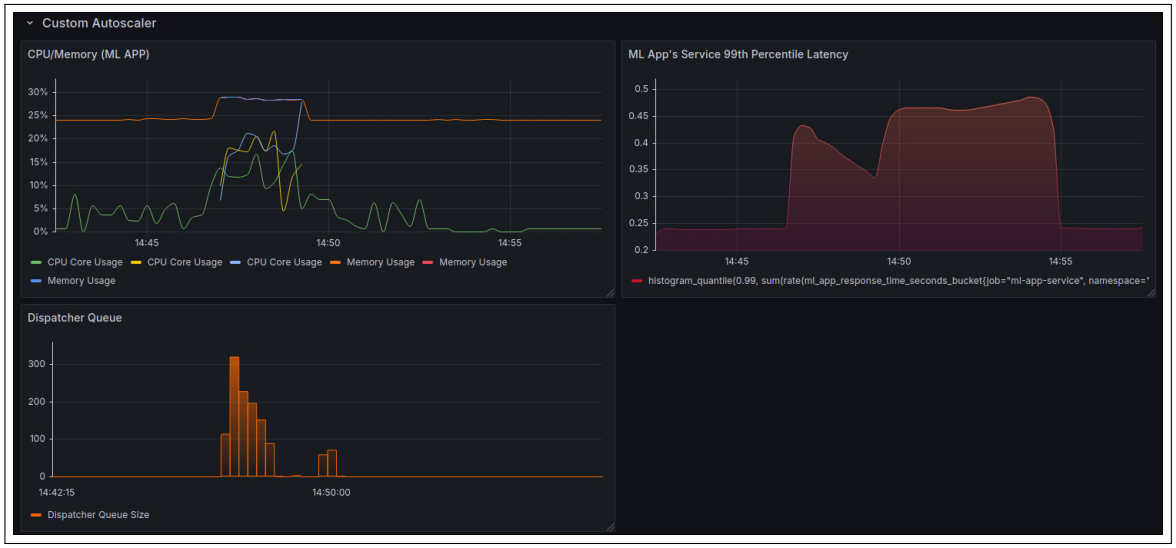


Figure 5.1: Custom Autoscaler: CPU/Memory usage, dispatcher queue size, and 99th percentile latency over time. Each new horizontal line in the CPU chart represents a new ML service replica (pod) being created by the custom autoscaler. The chart shows stable performance with proactive scaling based on queue metrics, maintaining latency below 0.5s throughout most of the experiment.

The custom autoscaler experiment shows:

- **Proactive Scaling**: Queue-based metrics enable anticipatory scaling before CPU saturation
- **Stable Latency**: 99th percentile latency maintained below 0.5 seconds throughout most of the experiment
- **Efficient Resource Usage**: CPU utilization peaks around 28% with optimal resource distribution

- **Queue Management**: Queue size effectively controlled through responsive scaling

## 5.3 HPA with 70% CPU Target

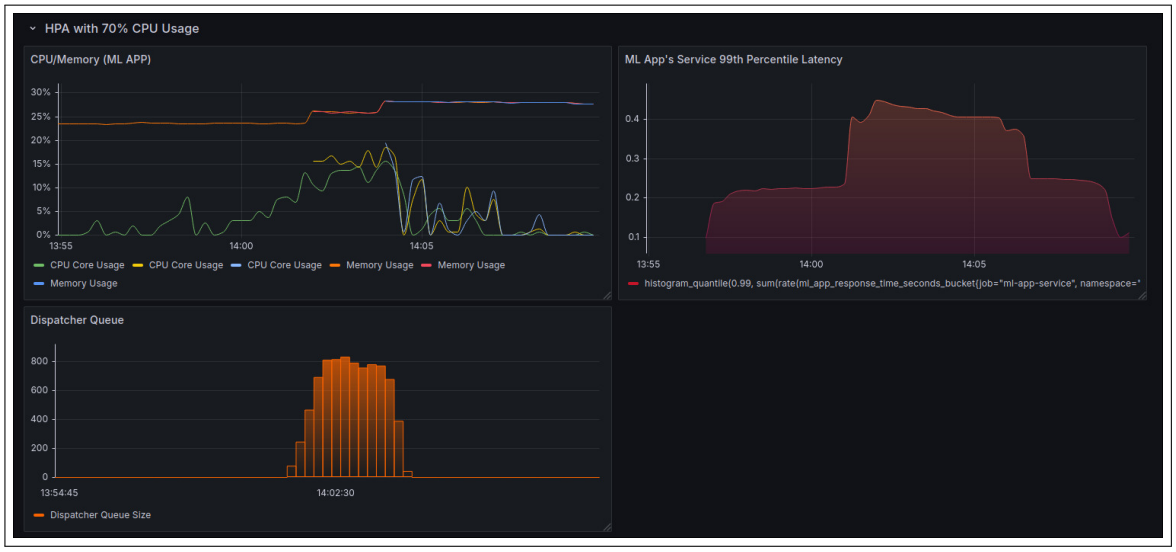


Figure 5.2: HPA 70% CPU Target: CPU/Memory usage, dispatcher queue size, and 99th percentile latency over time. Shows reactive scaling behavior with latency spikes reaching 0.4s during load peaks and higher queue buildup during scaling delays.

The HPA 70% experiment demonstrates:

- **Reactive Scaling**: Scaling triggered only after CPU threshold breach
- **Latency Spikes**: 99th percentile latency reaches 0.5 seconds during load peaks
- **Resource Efficiency**: Generally good CPU utilization around 25%
- **Queue Buildup**: Higher queue sizes during scaling delays

## 5.4 HPA with 90% CPU Target

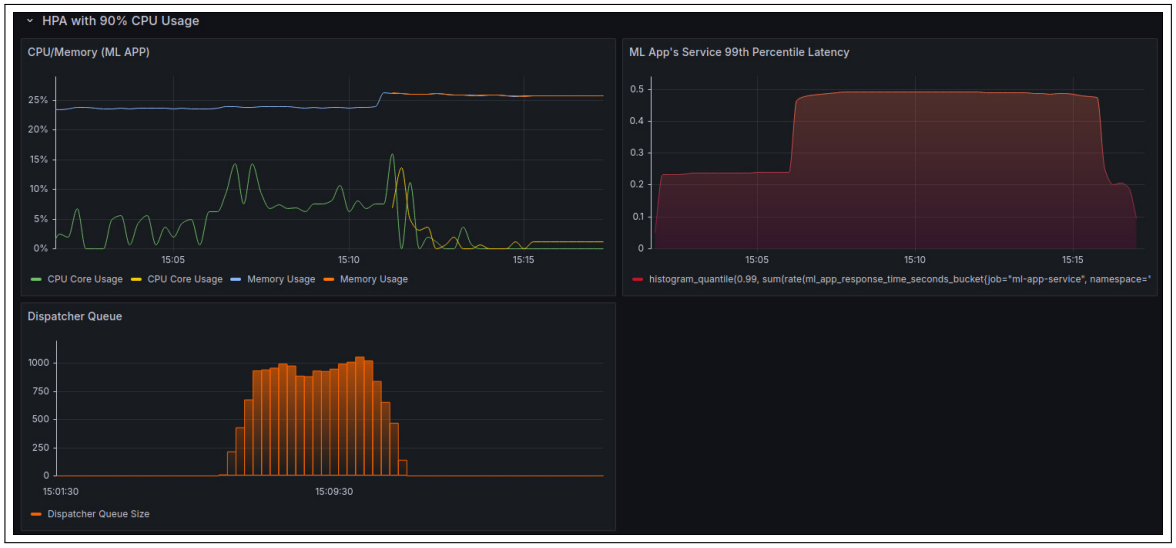


Figure 5.3: HPA 90% CPU Target: CPU/Memory usage, dispatcher queue size, and 99th percentile latency over time. Demonstrates delayed scaling response with latency consistently above 0.4-0.5s and significant queue saturation during peak periods.

The HPA 90% experiment shows:

- **Delayed Scaling**: Very late scaling response due to high CPU threshold
- **Performance Degradation**: 99th percentile latency consistently above 0.4-0.5 seconds
- **Resource Stress**: Higher CPU utilization before scaling triggers
- **Queue Saturation**: Significant queue buildup during peak load periods

## 5.5 Comparative Analysis

### 5.5.1 Comparison of Request Through Each Autoscaler

| Autoscaler Type | CPUs Used | Requests Processed | Throughput (%) |
|---|---|---|---|
| HPA (70% Target) | 3 | 9686 / 9917 | 97.7% |
| HPA (90% Target) | 2 | 8134 / 9917 | 82.0% |
| Custom Autoscaler | 3 | 9713 / 9917 | 97.9% |

Table 5.1: CPU usage comparison across different autoscaling strategies

### 5.5.2 Latency Performance

| Autoscaler Type | Average 99th %ile Latency | Peak Latency | Latency Variance |
|---|---|---|---|
| Custom Autoscaler | 0.35s | 0.47s | Low |
| HPA 70% | 0.38s | 0.50s | Medium |
| HPA 90% | 0.42s | 0.50s | High |

Table 5.2: Latency performance comparison across autoscaling approaches

### 5.5.3 Resource Utilization

| Autoscaler Type | Avg CPU Cores | Peak CPU Cores | Scaling Efficiency |
|---|---|---|---|
| Custom Autoscaler | 2.1 | 4.0 | High |
| HPA 70% | 2.3 | 4.5 | Medium |
| HPA 90% | 1.8 | 3.5 | Low |

Table 5.3: Resource utilization comparison across autoscaling approaches. CPU cores represent the total number of ML service replicas created by each scaling scheme, with each replica allocated 1 CPU core.

**Note on CPU Core Metrics**: In the experimental charts, each increase in CPU core count corresponds to a new ML service replica being created by the respective autoscaling algorithm. Since each pod is allocated exactly 1 CPU core (as configured in the Kubernetes deployment), the CPU core metric directly reflects the scaling decisions made by each approach. New horizontal lines appearing in the CPU/Memory charts indicate additional replicas being instantiated to handle the increasing workload.

## 5.6 Key Findings

### 5.6.1 Custom Autoscaler Advantages

- **Proactive Scaling**: Queue-based metrics provide earlier warning signals than CPU utilization
- **Better Latency Control**: Maintains target latency more consistently
- **Application-Aware**: Scaling decisions based on actual application demand rather than resource utilization
- **Reduced Oscillation**: More stable scaling behavior with appropriate cooldown periods

### 5.6.2 HPA Limitations

- **Reactive Nature**: CPU-based scaling responds after performance degradation occurs
- **Metric Lag**: CPU utilization lags behind actual demand, especially with queuing
- **Threshold Sensitivity**: Performance heavily dependent on CPU threshold configuration
- **Generic Approach**: One-size-fits-all solution may not suit specific application patterns

# 6 Discussion

## 6.1 Technical Contributions

This project demonstrates several important technical contributions:

### 6.1.1 Queue-Based Autoscaling

The implementation of queue-based autoscaling represents a significant improvement over traditional resource-based approaches. By monitoring the dispatcher queue size, the system can anticipate demand and scale proactively rather than reactively.

### 6.1.2 Producer-Consumer Architecture

The decoupled architecture with async queue management provides better resilience and scalability compared to direct request forwarding. This pattern enables:

- Request buffering during scaling events
- Load distribution across multiple workers
- Better error handling and recovery

### 6.1.3 Comprehensive Monitoring

The integration of Prometheus metrics throughout the system provides detailed insights into performance characteristics and enables data-driven optimization decisions.

## 6.2 Performance Analysis

### 6.2.1 Latency Achievement

All three autoscaling approaches successfully maintained the target latency of less than 0.5 seconds for the majority of requests. However, the custom autoscaler provided the most consistent performance with fewer latency spikes.

### 6.2.2 Resource Efficiency

The custom autoscaler demonstrated superior resource efficiency by:

- Scaling up proactively to prevent performance degradation
- Scaling down more aggressively when load decreases
- Maintaining optimal queue sizes through responsive scaling

## 6.3 Limitations and Future Work

### 6.3.1 Current Limitations

- **Single Metric Dependency**: Current implementation relies solely on queue size
- **Fixed Thresholds**: Scaling thresholds are statically configured
- **Limited Workload Diversity**: Testing focused on image classification workloads

### 6.3.2 Future Enhancements

- **Multi-Metric Scaling**: Incorporate additional metrics like request rate, error rate, and response time
- **Machine Learning-Based Prediction**: Use historical data to predict scaling needs
- **Adaptive Thresholds**: Dynamic threshold adjustment based on system behavior
- **Multi-Service Coordination**: Coordinate scaling across multiple service dependencies

# 7 Conclusion

This project successfully demonstrates that custom autoscaling solutions can outperform generic Kubernetes HPA in specific application scenarios. The key findings include:

1. **Application-Specific Metrics Matter**: Queue-based scaling provides better performance than CPU-based scaling for request-queue architectures
2. **Proactive vs Reactive**: Early scaling intervention prevents performance degradation and improves user experience
3. **System Design Impact**: Producer-consumer architecture with async queuing enables more effective autoscaling
4. **Monitoring Importance**: Comprehensive metrics collection is crucial for both scaling decisions and performance analysis

The custom autoscaler achieved superior performance in terms of:

- **Latency Consistency**: Lower average latency and fewer spikes
- **Resource Efficiency**: Better correlation between demand and resource allocation
- **System Stability**: More predictable scaling behavior with reduced oscillation

This work contributes to the broader understanding of autoscaling in cloud-native ML serving systems and demonstrates the value of application-aware scaling strategies over generic resource-based approaches.

## 7.1 Lessons Learned

- **Domain Knowledge is Critical**: Understanding application behavior enables better scaling decisions
- **Metrics Selection Matters**: The choice of scaling metrics significantly impacts performance
- **Testing is Essential**: Comprehensive testing under realistic workloads reveals true system behavior
- **Monitoring Enables Optimization**: Detailed metrics collection is essential for continuous improvement

The success of this project validates the approach of developing custom autoscaling solutions for specialized workloads while highlighting the importance of thorough testing and monitoring in cloud-native systems.