Explanation of OCR Implementation

1. Summary

This project implements a complete DevOps pipeline for an OCR (Optical Character Recognition) microservices system. The solution demonstrates enterprise-level practices including containerization, Kubernetes orchestration, GitOps deployment, and comprehensive monitoring. The system processes image uploads through a FastAPI gateway that forwards requests to a KServe-based OCR model service, all running on a production-grade Kubernetes infrastructure.

1.1 Cloud Platform Selection

Choice: Google Cloud Platform (GCP) Ubuntu 25 VM Rationale

- Cost-effective: Single VM approach reduces complexity and costs during development
- Resource adequacy: 16GB RAM and 8 CPUs provide sufficient resources for Minikube cluster
- Scalability: Easy to upgrade resources
- Free tier: AWS Haven't that much resources for free tier

1.2 Kubernetes Distribution

Choice: Minikube with Docker driver

- Local development: Perfect for development and testing environments
- Resource optimization: Efficient use of VM resources while leaving headroom for OS
- Docker integration: Seamless container runtime integration
- Feature completeness: Supports all required Kubernetes features (ingress, metrics-server)

1.3 External Access Strategy: Port forwarding with firewall rules Ports Configured

- 8080: OCR Model Service (KServe metrics)
- 8001: OCR Gateway Service (FastAPI API)
- 8443: ArgoCD UI (GitOps management)
- 3000: Grafana Dashboard (monitoring)
- 9090: Prometheus UI (metrics collection)

Security Considerations:

- Firewall rules restrict access to specific ports only
- Internal communication uses Kubernetes Service DNS
- No direct pod access from external networks

2. Application Architecture & Implementation

2.1 Microservices Design Pattern

The system follows a microservices architecture with clear separation of concerns:

OCR Model Service (KServe):

- Purpose: Core OCR processing using Tesseract engine
- Framework: KServe for ML model serving
- Port: 8080
- Responsibilities:
    - Image processing and text extraction
    - Metrics exposure for monitoring
    - Health check endpoints
    - Base64 image decoding

OCR Gateway Service (FastAPI):

- Purpose: API gateway and request routing
- Framework: FastAPI for high-performance APIs
- Port: 8001
- Responsibilities:
    - File upload handling
    - Request validation and processing
    - Load balancing to model service
    - API documentation (Swagger/OpenAPI)

2.2 Inter-Service Communication

Communication Pattern: Synchronous HTTP/REST Service Discovery: Kubernetes DNS (ocr-model:8080) Data Flow:

- Client uploads image to Gateway (POST /gateway/ocr)
- Gateway validates and encodes image to base64
- Gateway forwards request to Model service (HTTP POST)
- Model service processes image with Tesseract
- Model service returns extracted text
- Gateway returns response to client

3. Containerization

3.1 Docker Image Optimization

Base Image Selection: python:3.12-slim Rationale:

- Size efficiency: Slim variant reduces image size (~150MB vs 1GB+ full Python)
- Security: Fewer packages mean smaller attack surface
- Performance: Faster pull and deployment times
- Compatibility: Supports all required Python packages

Multi-stage Build Benefits (though simplified for assignment):

- Separate build and runtime environments
- Smaller production images
- Improved security by excluding build tools

Health Checks:

- Check container status

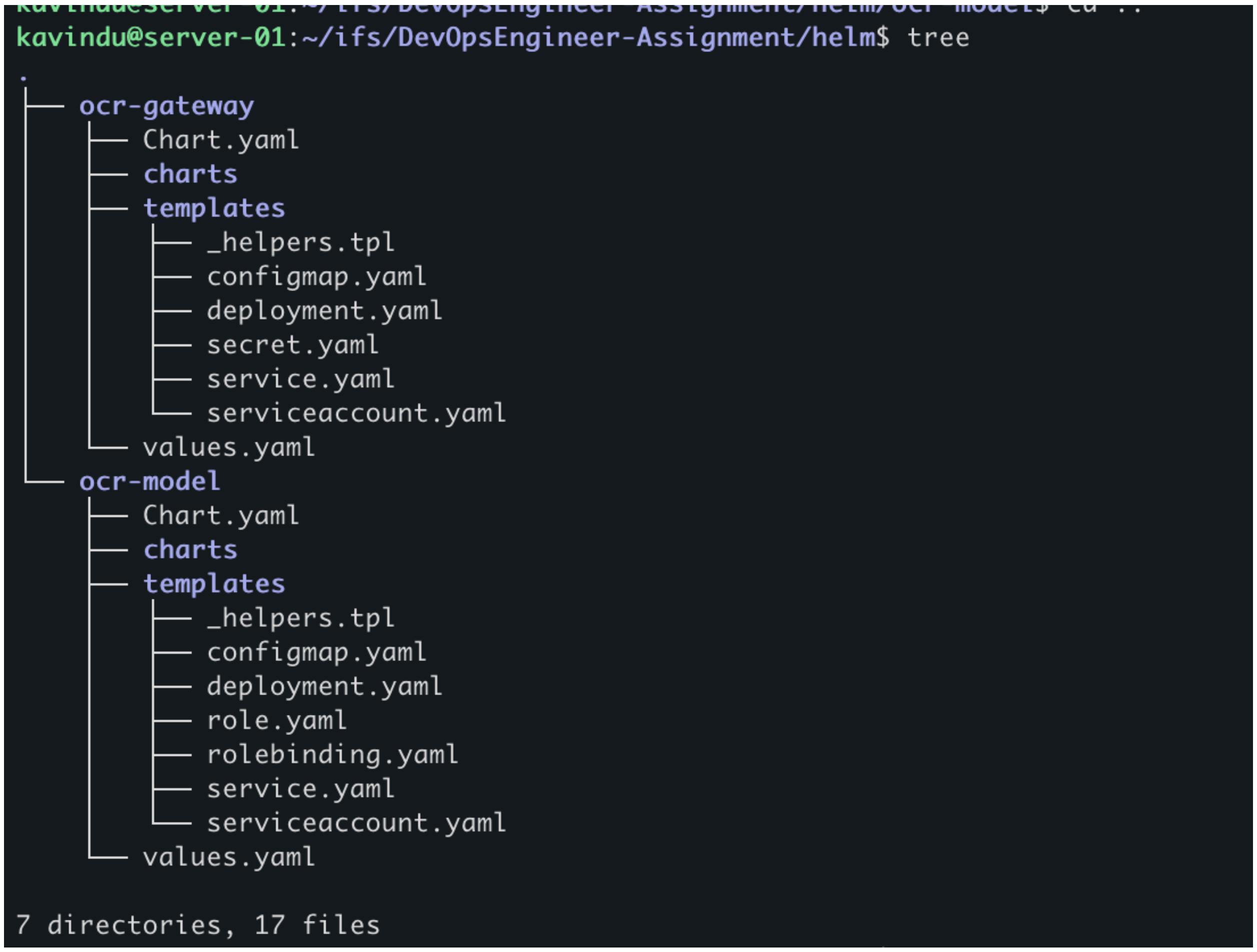
Non-root User

4. Kubernetes Resource Management

[NOTE: As the first stage I build a working solution with Deployment and Service only. But assignment required secrets/configmaps,SA,RBAC. Hence added those resources also.]

4.1 Helm Chart Architecture

Template-driven Deployment: Helm charts provide reusable, configurable deployments.



```
kavindu@server-01:~/ifs/DevOpsEngineer-Assignment/helm/ocr-model$ cd ..
kavindu@server-01:~/ifs/DevOpsEngineer-Assignment/helm$ tree
.
├── ocr-gateway
│   ├── Chart.yaml
│   ├── charts
│   ├── templates
│   │   ├── _helpers.tpl
│   │   ├── configmap.yaml
│   │   ├── deployment.yaml
│   │   ├── secret.yaml
│   │   ├── service.yaml
│   │   └── serviceaccount.yaml
│   └── values.yaml
└── ocr-model
    ├── Chart.yaml
    ├── charts
    ├── templates
    │   ├── _helpers.tpl
    │   ├── configmap.yaml
    │   ├── deployment.yaml
    │   ├── role.yaml
    │   ├── rolebinding.yaml
    │   ├── service.yaml
    │   └── serviceaccount.yaml
    └── values.yaml

7 directories, 17 files
```

4.2 Configuration Management Strategy

- ConfigMaps for Non-sensitive Data
- Secrets for Sensitive Information
- RBAC (Role-Based Access Control) Implementation

5. GitOps Implementation with ArgoCD

5.1 GitOps Workflow

Git as single source of truth for infrastructure and applications

Workflow:

- Developer commits configuration changes to GitHub
- ArgoCD detects changes through Git polling
- ArgoCD compares desired state (Git) vs actual state (Cluster)
- ArgoCD synchronizes cluster to match Git repository
- Automatic healing corrects any configuration drift

5.2 ArgoCD Resource Architecture

- AppProject: Organizational and security boundary
- ApplicationSet: Template-based application generation
- Applications: Individual service deployments

5.3 GitOps Benefits

- Declarative: Infrastructure as code approach
- Automated: No manual deployment steps
- Auditable: All changes tracked in Git history
- Reliable: Consistent deployments across environments
- Recoverable: Easy rollback using Git history

6. Monitoring and Observability Implementation

6.1 Metrics Collection Strategy

- Framework: Prometheus with KServe built-in metrics Collection Method: ServiceMonitor for automatic discovery

6.2 Metrics Analyzed

- **Total Inference Requests**:

Metric: `rate(python_gc_collections_total{generation="0"}[5m])`

- **Resource Utilization**:

```
CPU: rate(process_cpu_seconds_total[5m]) * 100
```

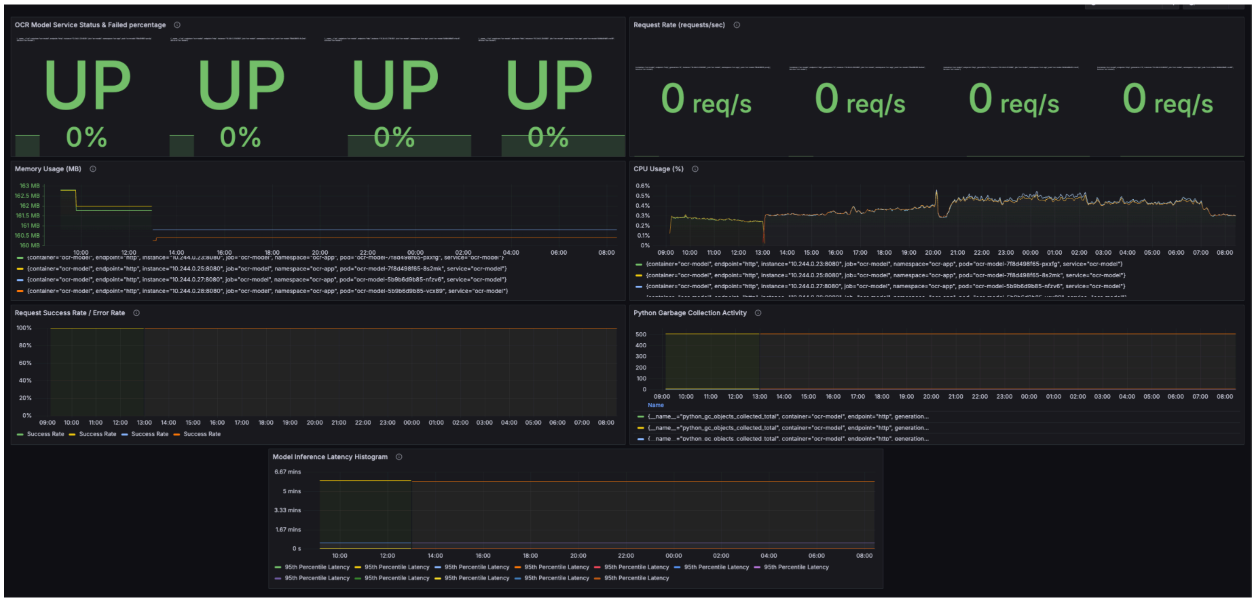
```
Memory: process_resident_memory_bytes / 1024 / 1024
```

- **Model Inference Latency**

```
Metric: python_gc_objects_collected_total
```

- **Error Rate/Request Status**:

```
Metric: up{job="ocr-model"} * 100
```


OCR Model Service Status & Failed percentage
UP 0%
UP 0%
UP 0%
UP 0%
Request Rate (requests/sec)
0 req/s
0 req/s
0 req/s
0 req/s
Memory Usage (MB)
163 MB
162.5 MB
162 MB
161.5 MB
161 MB
160.5 MB
160 MB
{container="ocr-model", endpoint="http", instance="10.244.0.23:8080", job="ocr-model", namespace="ocr-app", pod="ocr-model-7f8d498f65-pxxfg", service="ocr-model"}
{container="ocr-model", endpoint="http", instance="10.244.0.25:8080", job="ocr-model", namespace="ocr-app", pod="ocr-model-7f8d498f65-8s2mk", service="ocr-model"}
{container="ocr-model", endpoint="http", instance="10.244.0.27:8080", job="ocr-model", namespace="ocr-app", pod="ocr-model-5b9b6d9b85-nfzv6", service="ocr-model"}
{container="ocr-model", endpoint="http", instance="10.244.0.28:8080", job="ocr-model", namespace="ocr-app", pod="ocr-model-5b9b6d9b85-vcx89", service="ocr-model"}
CPU Usage (%)
0.6%
0.5%
0.4%
0.3%
0.2%
0.1%
0%
Request Success Rate / Error Rate
100%
80%
60%
40%
20%
0%
Success Rate
Success Rate
Success Rate
Success Rate
Python Garbage Collection Activity
500
400
300
200
100
0
Name
Model Inference Latency Histogram
6.67 mins
5 mins
3.33 mins
1.67 mins
0 s
95th Percentile Latency