DEVOPS CA2 Report

Group 12

Group Members

Janmejay Pandya (22070122086)
Sachin Mhetre (22070122119)
Mihir Hebalkar (22070122120)
Onkar Mendhapurkar (22070122135)

Batch : CSE 2022 - 26

1. Architecture Overview

1.1 System Design

WealthWise follows a microservice architecture pattern with clear separation of concerns between frontend and backend services. The system is containerized using Docker and orchestrated using Kubernetes, providing scalability, resilience, and portability.

Core Components:

- **Backend Service**: Django-based REST API with Gunicorn WSGI server
- **Container Registry**: Azure Container Registry (ACR) for private image storage
- **Orchestration Platform**: Azure Kubernetes Service (AKS) with 2-node cluster
- **CI/CD Pipeline**: GitHub Actions for automated build and deployment
- **Monitoring Stack**: Prometheus for metrics collection, Grafana for visualization
- **Secrets Management**: Kubernetes Secrets for sensitive configuration

1.2 Technology Stack

Layer	Technology	Purpose
Frontend	React, Node.js	User interface
Backend	Django, Python, Gunicorn	Business logic and API
Containerization	Docker	Application packaging

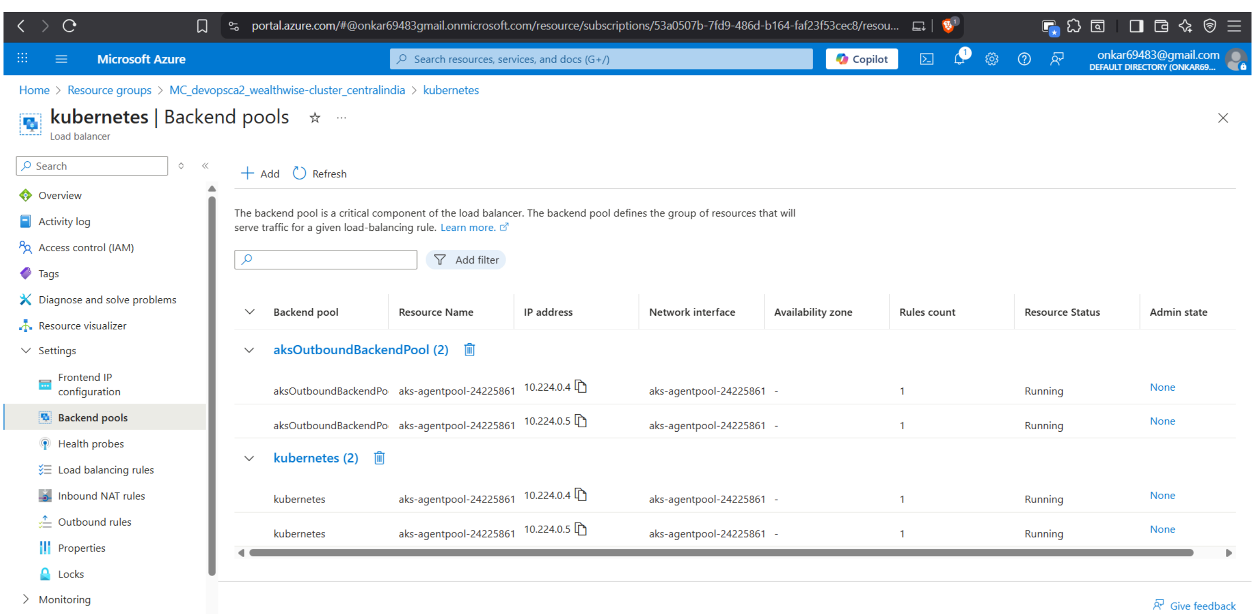
Orchestration	Kubernetes (AKS)	Container management
Registry	Azure Container Registry	Image storage
CI/CD	GitHub Actions	Automation pipeline
Monitoring	Prometheus, Grafana	Observability
Instrumentation	django-prometheus	Metrics exposure
Cloud Provider	Microsoft Azure	Infrastructure hosting

1.3 Infrastructure Architecture

The infrastructure is deployed on Azure with the following specifications:

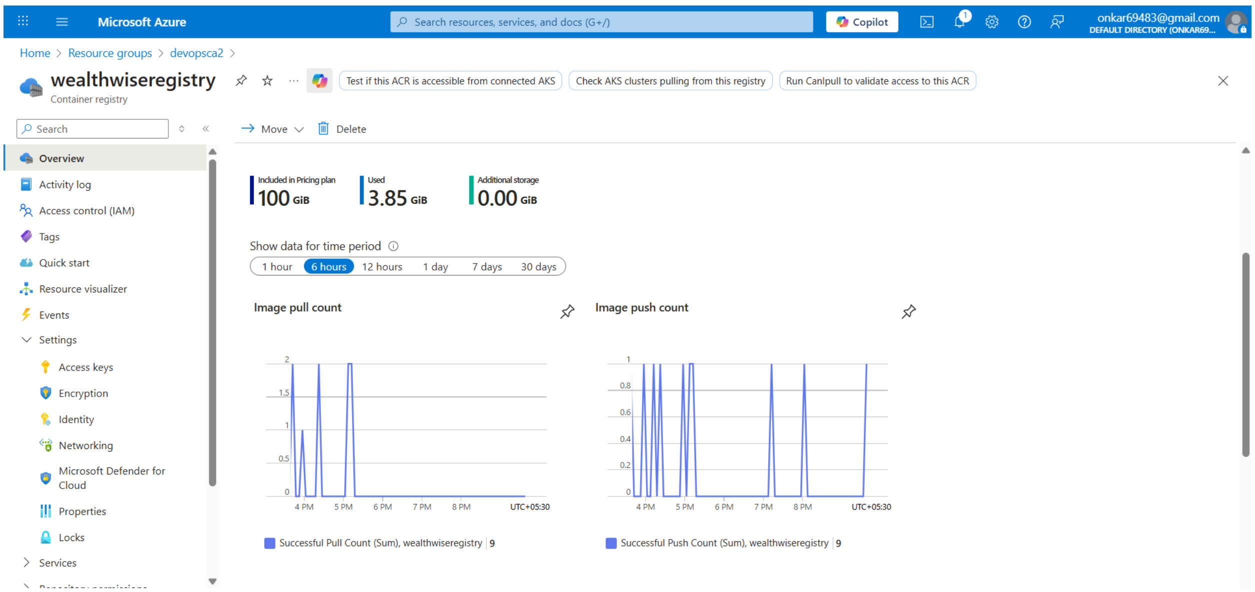
Azure Kubernetes Service (AKS):

- Cluster Name: `wealthwise-cluster`
- Region: Central India
- Kubernetes Version: 1.32.7
- Node Pool: 2 nodes (Ubuntu, Standard_D4ds_v5)
- Networking: Azure CNI Overlay
- Authentication: Local accounts + Kubernetes RBAC
- Service Exposure: LoadBalancer type



Azure Container Registry (ACR):

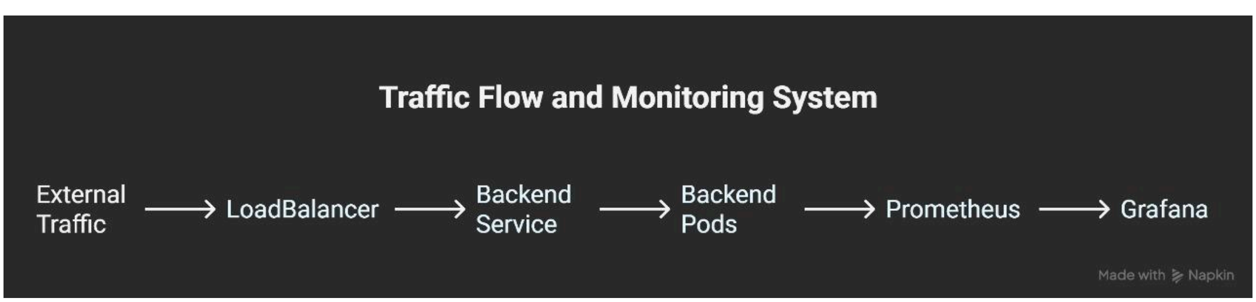
- Registry Name: `wealthwiseregistry`
- SKU: Standard
- Access: RBAC-enabled
- Integration: Direct attachment to AKS cluster



1.4 Networking Model

The application uses Azure LoadBalancer service type to expose the backend API publicly. Internal service-to-service communication occurs through ClusterIP services. The monitoring stack operates in a dedicated `monitoring` namespace with separate networking policies.

External Traffic → LoadBalancer → Backend Service → Backend Pods
↓
Prometheus (scrape)
↓
Grafana (visualize)



2. Ansible

Ansible is an open-source automation tool used for configuration management, application deployment, and task automation. It simplifies IT operations by allowing users to define infrastructure as code through simple, human-readable YAML files called playbooks. Ansible operates agentlessly, using SSH or WinRM to communicate with remote systems, making it lightweight and easy to deploy across various environments.

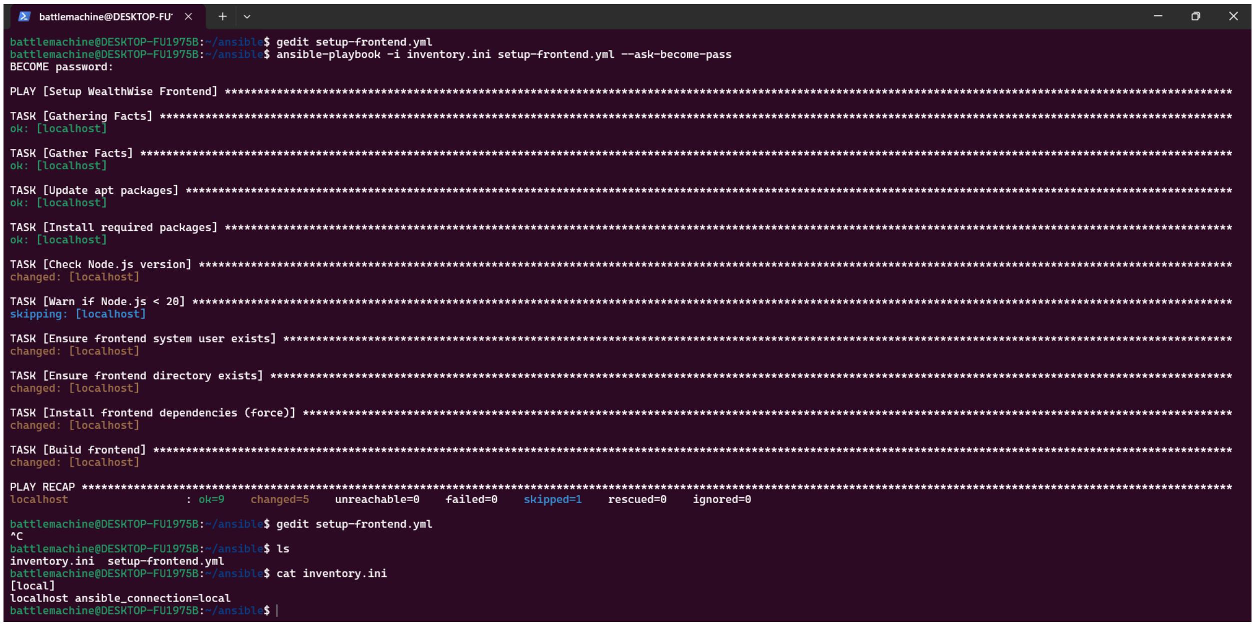
2.1 Objectives:

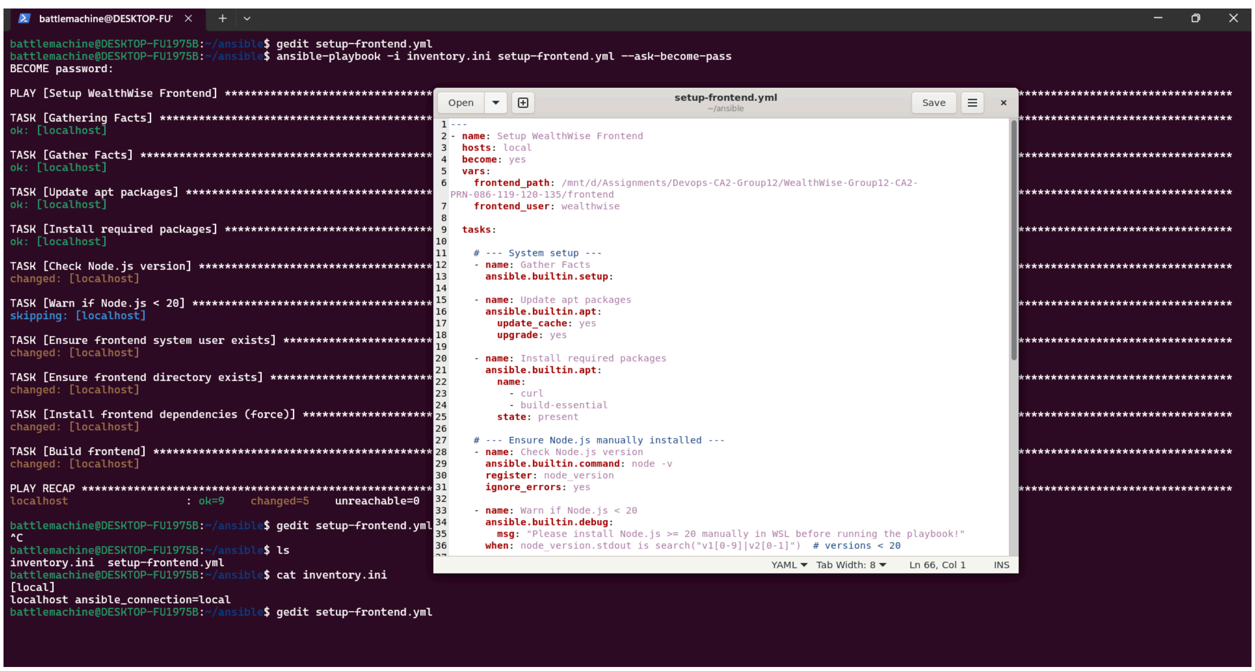
- Automate repetitive IT tasks (e.g., configuration, deployment, updates).
- Ensure consistency and reliability across multiple systems.
- Simplify complex workflows through playbooks and roles.
- Enable Infrastructure as Code (IaC) for better version control and scalability.
- Minimize human error and improve operational efficiency.

2.2 Architecture and Approach

Target Environment: Ubuntu (WSL2)
Controller Node: Local machine running Ansible
Inventory: Single host setup (`localhost`)
Execution Mode: Local execution using `ansible-playbook -i inventory.ini setup-frontend.yml --ask-become-pass`

The playbook provisions the frontend environment end-to-end — installing Node.js, ensuring system packages, managing directories, creating a dedicated user, installing dependencies, and building the frontend project.





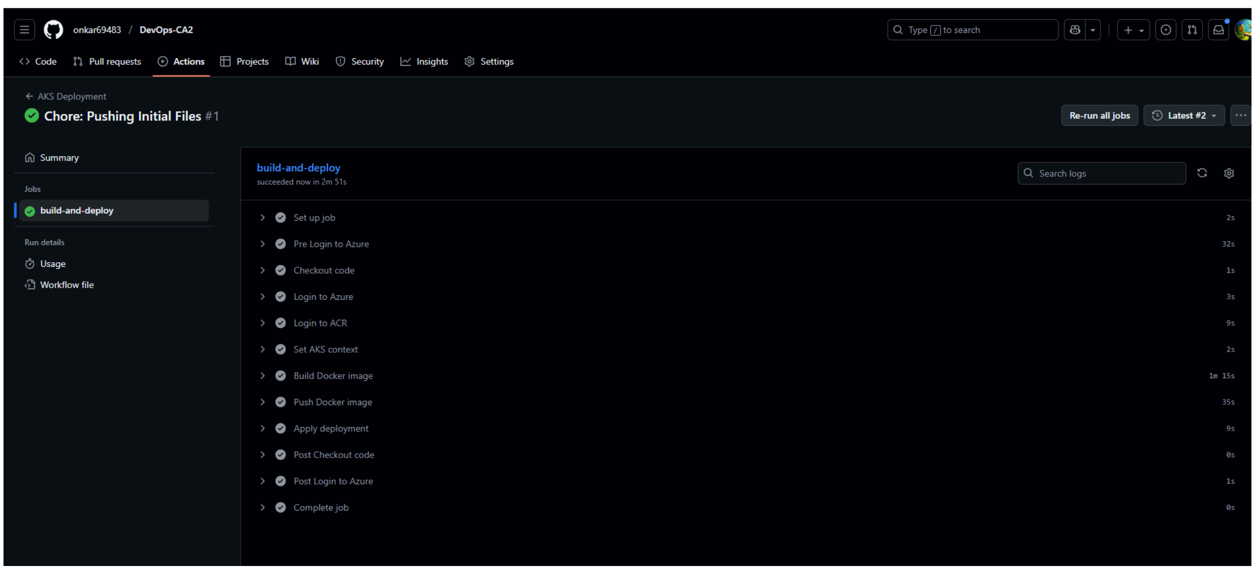
3. CI/CD Pipeline Flow

3.1 Pipeline Architecture

The CI/CD pipeline implements a GitOps workflow where every commit to the main branch triggers an automated build, test, and deployment sequence. This ensures continuous delivery of features with minimal manual intervention.

Pipeline Stages:

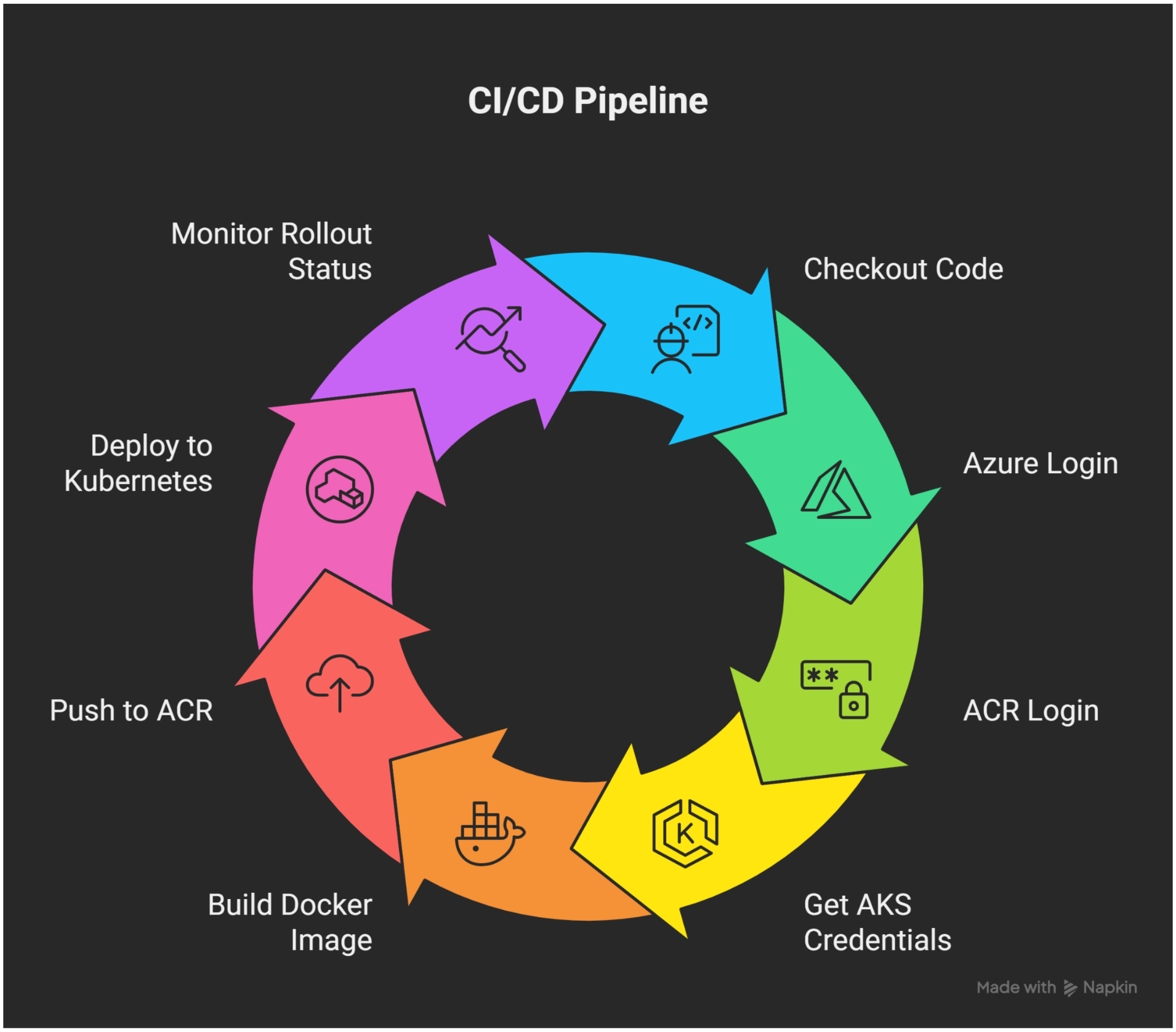
1. **Source Control**: Code changes pushed to GitHub repository
2. **Trigger**: GitHub webhook initiates workflow execution
3. **Authentication**: Azure service principal credentials validated
4. **Build**: Docker image constructed from Dockerfile
5. **Push**: Image pushed to Azure Container Registry with latest tag
6. **Deploy**: Kubernetes manifests applied to AKS cluster
7. **Verification**: Deployment rollout status monitored



3.2 GitHub Actions Workflow

The workflow is defined in `.github/workflows/ci-cd.yml` and executes the following steps:

Checkout Code → Azure Login → ACR Login → Get AKS Credentials
→ Build Docker Image → Push to ACR → Deploy to Kubernetes
→ Monitor Rollout Status



Key Features:

- Automated trigger on main branch commits
- Secure credential management using GitHub Secrets
- Atomic deployments with rollout verification
- Zero-downtime deployments using Kubernetes rolling updates
- Automatic rollback on deployment failures

3.3 Docker Image Management

Each backend service is packaged as a Docker image containing:

- Base Python runtime environment
- Application dependencies (requirements.txt)

- Django application code
- Gunicorn WSGI server configuration
- Health check endpoints

Images are tagged with `latest` and pushed to ACR, enabling AKS to pull authenticated images directly without additional configuration.

3.4 Kubernetes Deployment Strategy

The deployment uses Kubernetes native resources:

Deployment Resource:

- Manages 2 replica pods for high availability
- Implements rolling update strategy
- Defines resource requests and limits
- Configures liveness and readiness probes
- Injects secrets as environment variables

Service Resource:

- Exposes pods via LoadBalancer
- Maps external port 80 to container port 8000
- Provides stable endpoint for external access
- Enables automatic load distribution

3.5 Continuous Verification

Post-deployment verification includes:

- Pod health status checks
- Service endpoint availability
- Rollout status monitoring
- Automatic rollback on failure detection

4. Monitoring and Observability

4.1 Monitoring Strategy

Comprehensive monitoring was implemented to ensure system reliability and performance visibility. The monitoring stack provides real-time insights into application health, performance metrics, and error rates.

Objectives:

- Track application uptime and availability
- Monitor request latency and throughput
- Identify and alert on error conditions
- Enable data-driven capacity planning
- Support troubleshooting and debugging

4.2 Metrics Instrumentation

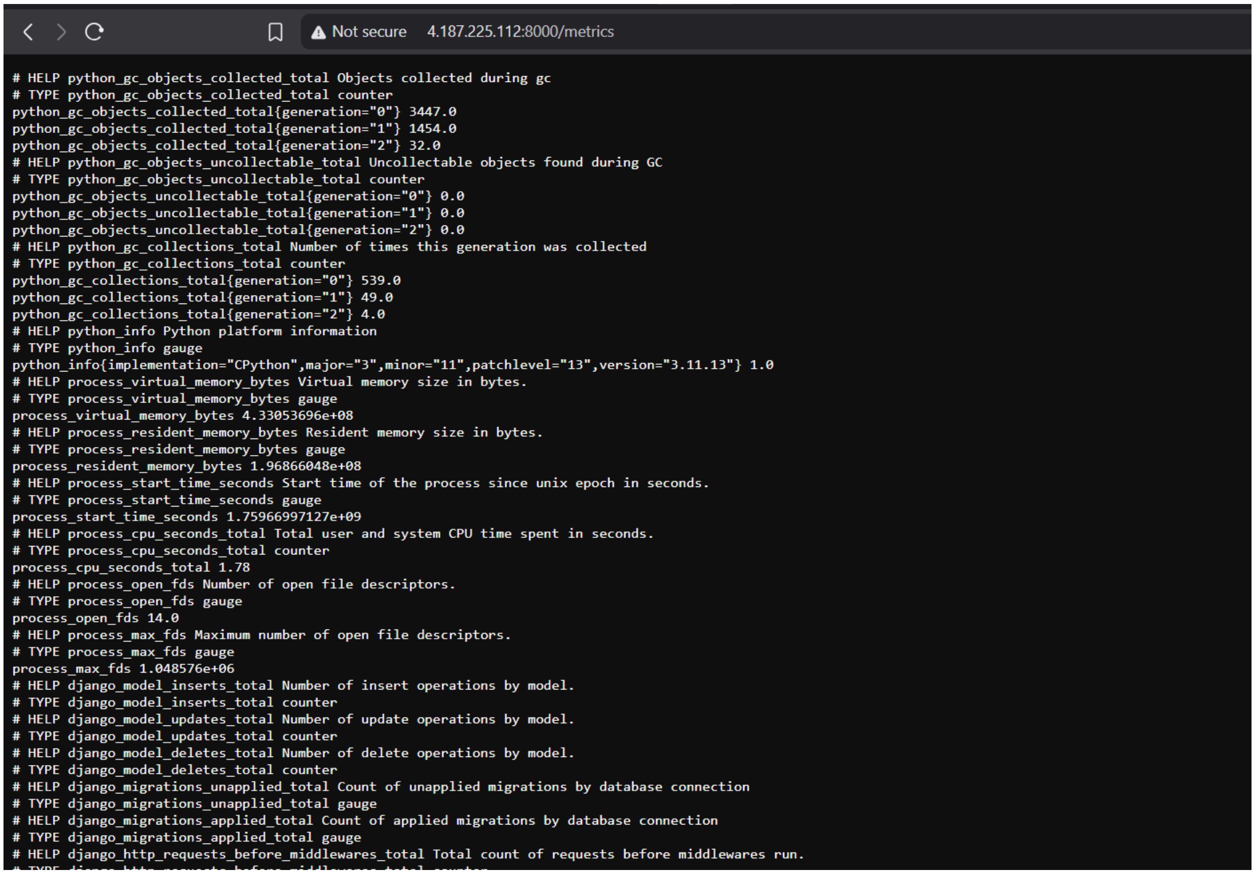
The Django backend exposes Prometheus-compatible metrics using the `django-prometheus` library:

Implementation:

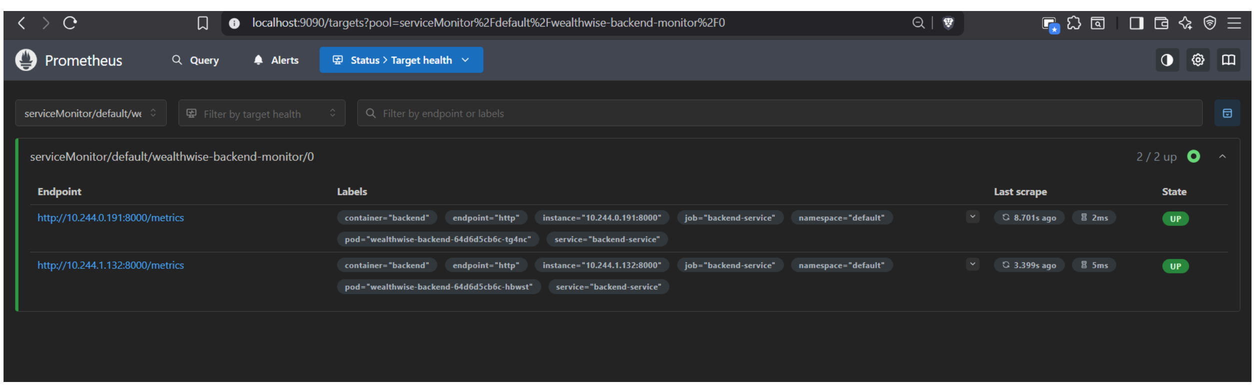
```
# Middleware integration
MIDDLEWARE = [
   'django_prometheus.middleware.PrometheusBeforeMiddleware',
   *MIDDLEWARE,
   'django_prometheus.middleware.PrometheusAfterMiddleware',
]

# Metrics endpoint exposed at /metrics
```

Exposed Metrics:

- HTTP request count and duration
- Response status code distribution
- Database query performance
- Application-specific business metrics



4.3 Prometheus Deployment



Prometheus was deployed using Helm charts in a dedicated monitoring namespace:

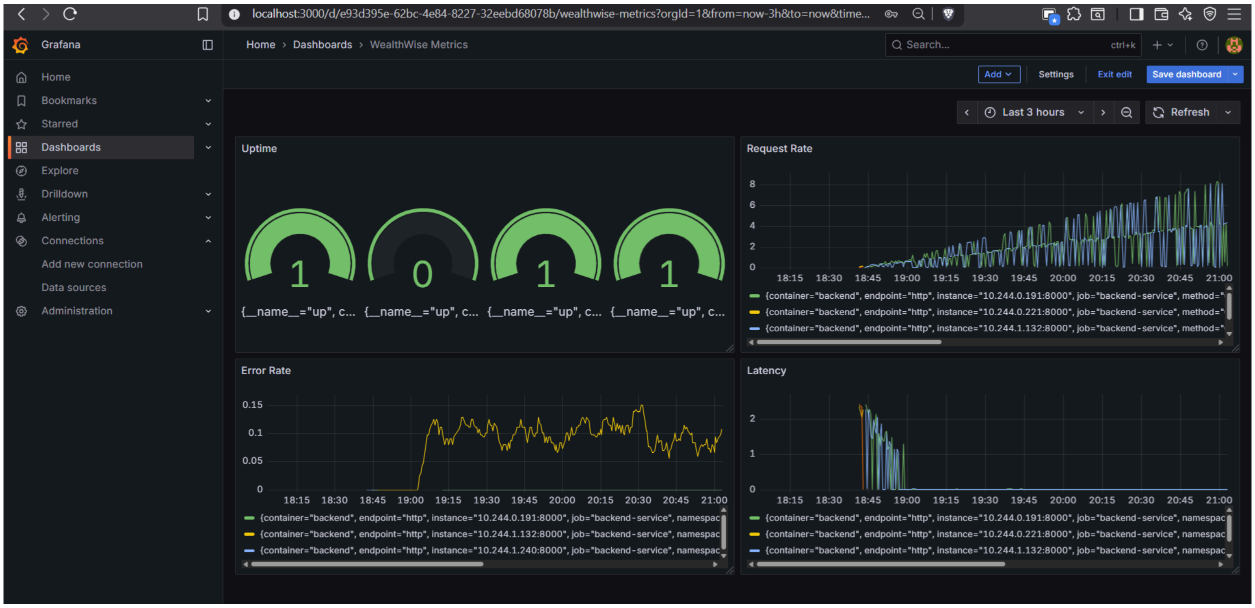
helm install prometheus prometheus-community/prometheus --namespace monitoring

Configuration:

- Automatic service discovery for Kubernetes targets
- 15-second scrape interval for real-time monitoring

- Data retention period of 15 days
- Target filtering based on labels

4.4 Grafana Dashboard



Grafana provides visualization and alerting capabilities:

Installation:

```
helm install grafana grafana/grafana \
  --namespace monitoring \
  --set service.type=LoadBalancer
```

Dashboard Panels:

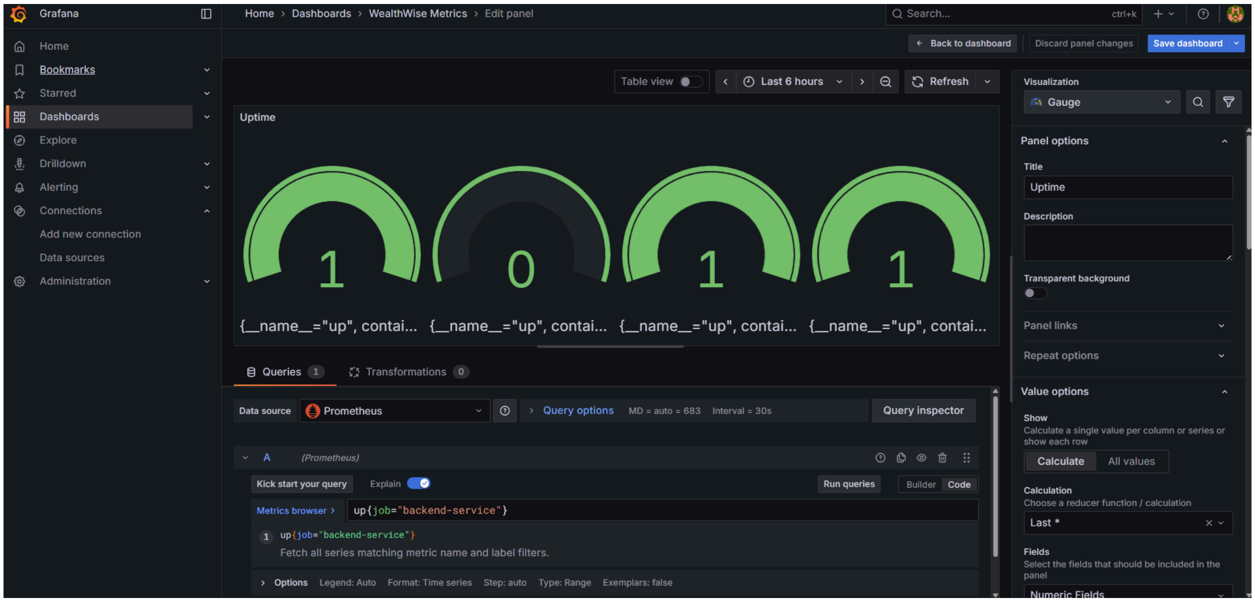
1. **Uptime Monitor**: Displays service availability using `up{job="django"}` query
2. **Request Latency**: Shows P50, P95, P99 latency percentiles
3. **Error Rate**: Tracks 5xx errors using `rate()` function
4. **Request Volume**: Visualizes requests per second

PromQL Queries Used:

```
# Service uptime
up{job="django"}

# Average latency
http_server_requests_seconds_sum / http_server_requests_seconds_count
```

```
# Error rate (5xx responses)
sum(rate(http_server_requests_seconds_count{status=~"5.."}[1m]))

# Request throughput
rate(http_server_requests_seconds_count[5m])
```

Example screenshot of one of the queries



5. Challenges and Solutions

5.1 ServiceMonitor Configuration

Challenge: Initial ServiceMonitor configuration failed to discover backend targets in Prometheus.

Root Cause: Port specification used string format (`"8000"`) instead of integer format in ServiceMonitor YAML.

Solution:

```
# Incorrect
port: "8000"

# Correct
port: 8000
```

Learning: Kubernetes API specifications require strict type adherence. All numeric values must be specified as integers, not strings.

5.2 Metrics Endpoint Accessibility

Challenge: Prometheus showed targets as "down" despite pod health.

Root Cause: Metrics endpoint `/metrics` not properly exposed in Django URL configuration.

Solution: Added django-prometheus URLs to Django project:

```
urlpatterns += [
    path('', include('django_prometheus.urls')),
]
```

Learning: Application-level configuration must align with infrastructure expectations.

5.3 WSL and Docker Integration

Challenge: Docker commands failed in WSL environment during local development.

Root Cause: Docker daemon not running or improper WSL-Windows Docker Desktop integration.

Solution:

- Enabled WSL 2 backend in Docker Desktop settings
- Configured Docker context to use WSL socket
- Verified with `docker ps` command

Learning: Modern development workflows require proper integration between Windows, WSL, and containerization tools.

5.4 Kubernetes Secret Management

Challenge: Sensitive credentials exposed in deployment manifests during initial setup.

Root Cause: Direct embedding of secrets in YAML files, which would be committed to version control.

Solution: Implemented Kubernetes Secrets:

```
kubectl create secret generic wealthwise-secrets \
  --from-literal=ENV_MODE=prod \
  --from-literal=GOOGLE_CLIENT_ID=<value>
```

Referenced in deployment:

```
envFrom:
  - secretRef:
      name: wealthwise-secrets
```

Learning: Never commit secrets to version control. Use proper secret management solutions.

6. Lessons Learned

6.1 Technical Insights

Infrastructure as Code (IaC):

- Ensured consistent, reproducible environments.

- Enabled version-controlled, auditable infrastructure changes.
- Reduced manual errors and improved recovery time.

Container Orchestration (Kubernetes):

- Achieved auto-scaling and self-healing deployments.
- Enabled zero-downtime rolling updates.
- Optimized resource utilization through scheduling.

Monitoring & Observability:

- Detected issues proactively via metrics and alerts.
- Informed capacity planning through data insights.
- Enhanced reliability via real-time visualization and alerting.

6.2 DevOps Best Practices

GitOps:

- Maintained single source of truth in Git.
- Used declarative configs with automated sync to production.
- Enabled change tracking through pull requests.

Security:

- Managed secrets securely outside version control.
- Applied RBAC and network policies for isolation.
- Scanned containers for vulnerabilities before deployment.

6.3 Tool-Specific Learnings

Kubernetes: Resource limits, probes, and namespaces ensure stability.
Prometheus: Dynamic service discovery and rule-based metric optimization.
Grafana: Reusable, versioned dashboards with proactive alerts.
Docker: Multi-stage builds, caching, and security scanning improved CI/CD efficiency.

6.4 Process Improvements

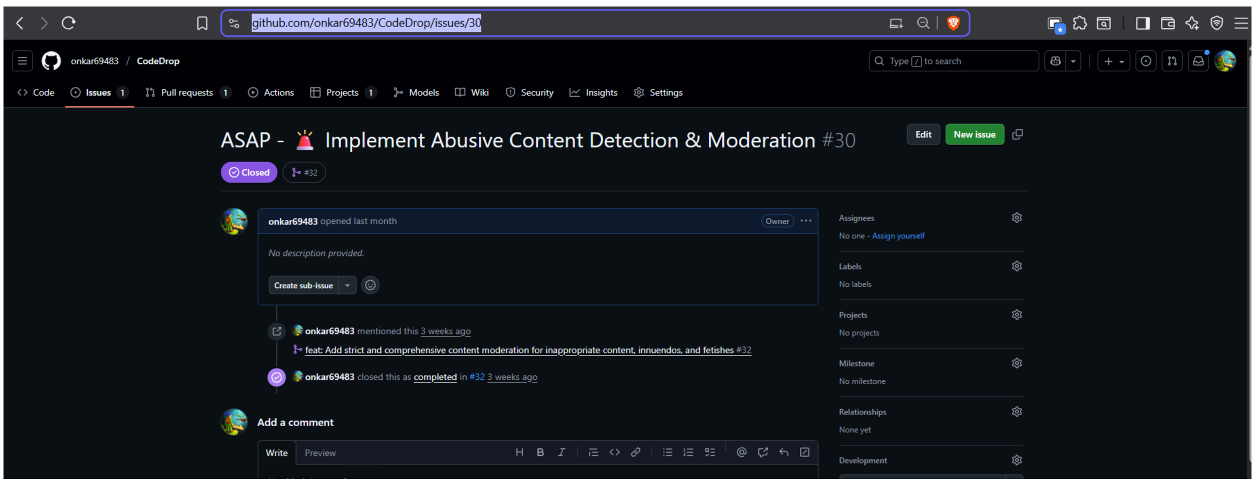
Collaboration: Regular syncs, Markdown docs, and peer reviews improved efficiency.
Continuous Improvement: Retrospectives, incremental changes, and metrics-based optimization enhanced project maturity.

Bonus Task

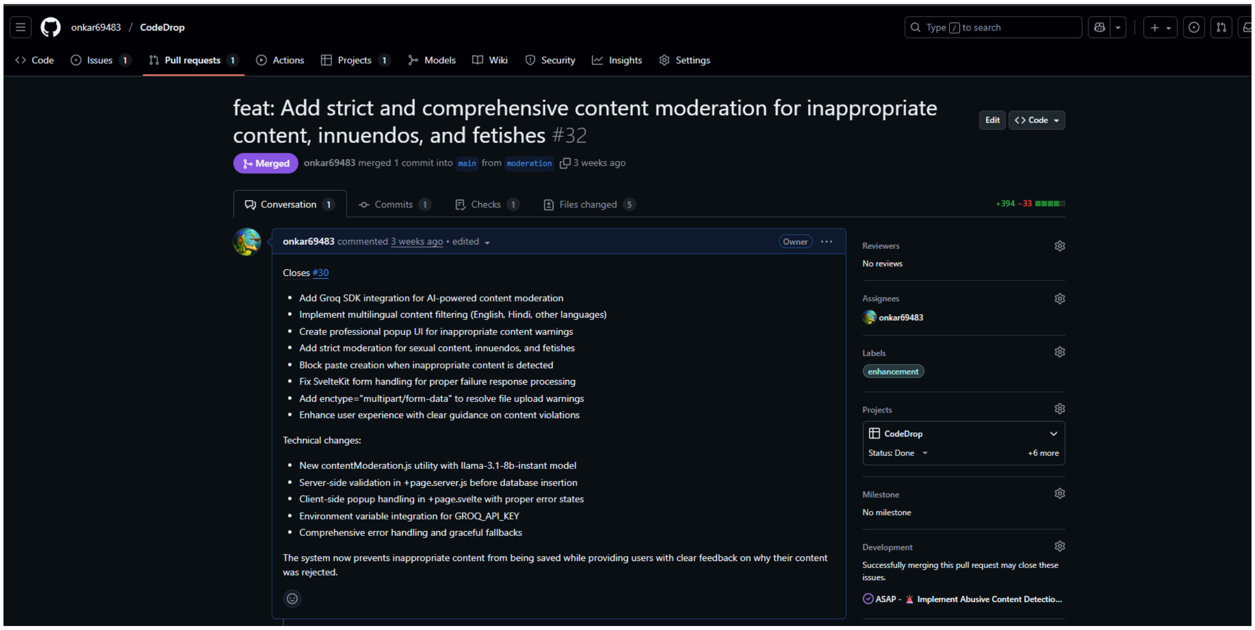
Open Source Contribution

Website - https://codedrop.vercel.app/

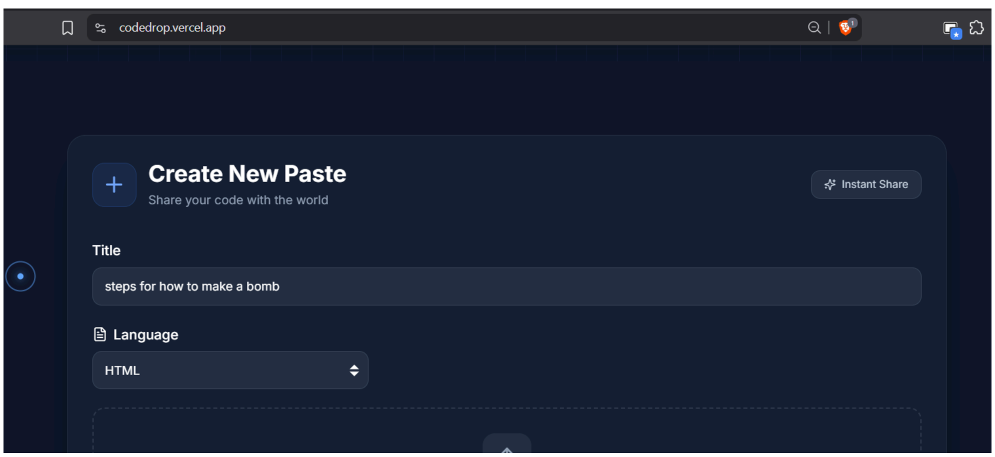
Issue - https://github.com/onkar69483/CodeDrop/issues/30

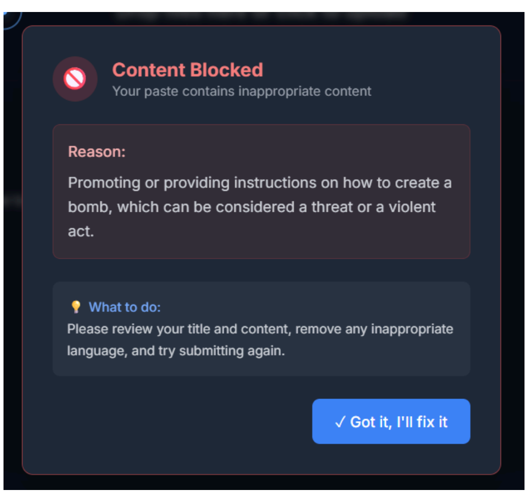


Pull request - https://github.com/onkar69483/CodeDrop/pull/32



How the issue was solved





- Add Groq SDK integration for AI-powered content moderation
- Implement multilingual content filtering (English, Hindi, other languages)
- Create professional popup UI for inappropriate content warnings
- Add strict moderation for sexual content, innuendos, and fetishes
- Block paste creation when inappropriate content is detected
- Fix SvelteKit form handling for proper failure response processing
- Add enctype="multipart/form-data" to resolve file upload warnings
- Enhance user experience with clear guidance on content violations
- Technical changes:
-
- New contentModeration.js utility with llama-3.1-8b-instant model
- Server-side validation in +page.server.js before database insertion
- Client-side popup handling in +page.svelte with proper error states
- Environment variable integration for GROQ_API_KEY
- Comprehensive error handling and graceful fallbacks

The system now prevents inappropriate content from being saved while providing users with clear feedback on why their content was rejected.

Appendices

Appendix A: Key Commands Reference

```
# Azure CLI Commands
az aks get-credentials --resource-group devopsca2 --name wealthwise-cluster
az acr login --name wealthwiseregistry

# Docker Commands
docker build -t wealthwise-backend:latest .
docker push wealthwiseregistry.azurecr.io/wealthwise-backend:latest

# Kubernetes Commands
kubectl apply -f deployment.yaml
kubectl get pods -o wide
kubectl logs -f <pod-name>
kubectl describe service backend-service

# Helm Commands
helm install prometheus prometheus-community/prometheus -n monitoring
helm list -n monitoring
```

```
# Monitoring Access
kubectl port-forward svc/prometheus-server 9090:80 -n monitoring
kubectl port-forward svc/grafana 3000:80 -n monitoring
```

Appendix B: Resource Specifications

AKS Cluster:

- Nodes: 2x Standard_D4ds_v5 (4 vCPU, 16 GB RAM each)
- Kubernetes Version: 1.32.7
- Total Capacity: 8 vCPU, 32 GB RAM
- Network Plugin: Azure CNI Overlay
- DNS Service: CoreDNS

Backend Deployment:

- Replicas: 3
- CPU Request: 250m per pod
- Memory Request: 512Mi per pod
- Image: wealthwiseregistry.azurecr.io/wealthwise-backend:latest

Monitoring Stack:

- Prometheus: 2 GB memory, 10 GB storage
- Grafana: 1 GB memory, 5 GB storage
- Retention: 15 days

Appendix C: Technologies and Versions

Tool	Version	Purpose
Kubernetes	1.32.7	Container orchestration
Docker	24.0+	Containerization
Python	3.11	Backend runtime
Django	4.2	Web framework
Prometheus	2.45+	Metrics collection

Grafana	10.0+	Visualization
Helm	3.12+	Package management
GitHub Actions	N/A	CI/CD automation