DEVOPS-BASED PATIENT MANAGEMENT SYSTEM

Student Name: Mbarushimana Danny
Registration Number: 25RP20201
Course / Program: Bachelor of Information Technology
Semester: Semester I
Academic Year: 2025 – 2026

MODULE INFORMATION

Module Title: DevOps
Module Code: ITLDO801

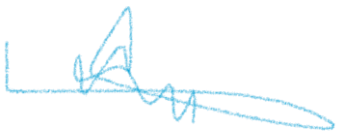
PROJECT DETAILS

Project Title:
Design and Implementation of a Patient Management System Using DevOps Practices

Project Description:
This project demonstrates a complete DevOps lifecycle including application development, version control, continuous integration, containerization, Kubernetes deployment, monitoring, and continuous deployment using Railway.

DECLARATION

I declare that this project is my original work and has not been submitted elsewhere for academic credit.
All sources used have been properly acknowledged.

Student Signature: 

Date: 12/21/2025

Phase 1: Infrastructure & Environment Setup

Status: ✅ Completed

1. Introduction

Phase 1 established the foundational development environment for the Patient Management System. The objective was to configure a robust, containerized infrastructure using WSL2, Docker, Kubernetes, and PHP to ensure a seamless transition into the development lifecycle.

2. Environment Preparation

The following core technologies were successfully integrated and verified within the workspace:

- WSL2 & Ubuntu 22.04 LTS: Installed to provide a native Linux kernel environment on Windows, ensuring high-performance tool compatibility.
- Docker Desktop: Configured with the WSL2 backend enabled. This serves as the primary engine for containerization.
- Kubernetes (K8s): Orchestration enabled via Docker Desktop settings to manage future application scaling.
- Version Control (Git): Installed within the Ubuntu environment to manage source code and collaborative branching.
- PHP 8.1+: Configured as the backend CLI environment for the system's core logic.

Verification Checklist:

| Tool | Command | Result | Status |

| :--- | :--- | :--- | :--- |

| Git | git --version | Version confirmed | ✅ Pass |

| PHP | php -v | v8.1+ confirmed | ✅ Pass |

| Docker | docker --version | Engine active | ✅ Pass |

| Kubernetes | kubectl version | Client active | ✅ Pass |

---

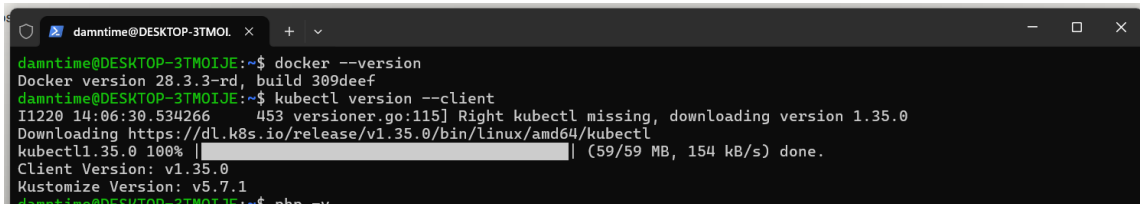
3. Networking & Configuration

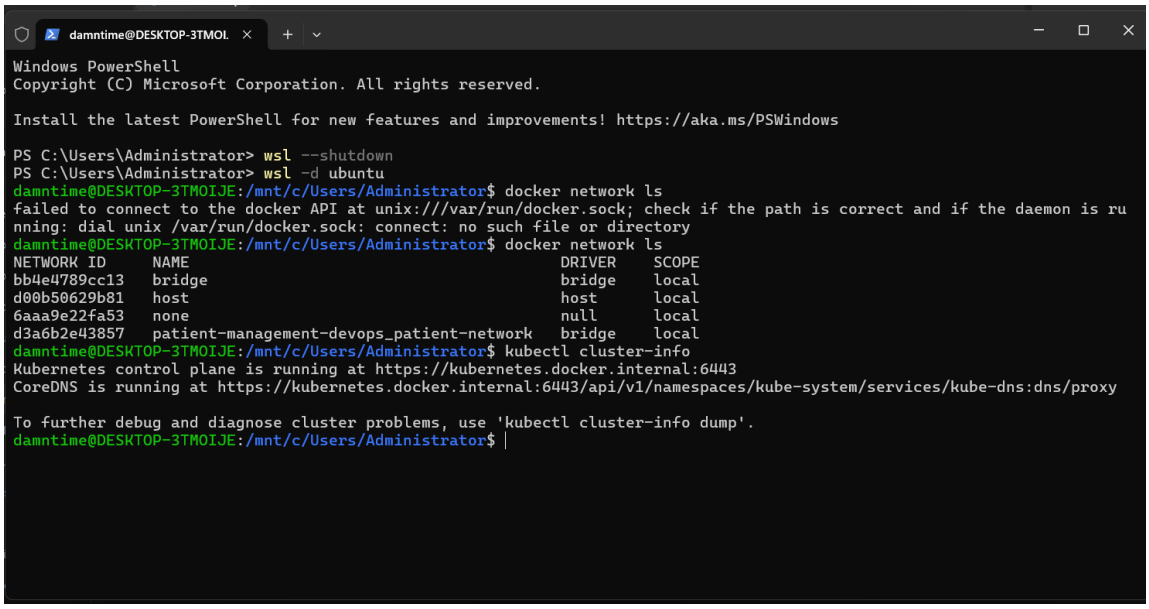
To ensure the application can communicate across services and be accessed by users, the following network audits were performed:

- Docker Networking: Verified via docker network ls. The default bridge network is active, allowing container-to-container communication.
- K8s Cluster Integrity: Verified via kubectl cluster-info. The control plane and core services are operational.
- Port Allocation: A manual check confirmed Port 80 is available. This port is reserved for exposing the web application's frontend.

4. Summary & Evidence

The development environment is fully synchronized. All tools respond to CLI queries, and the networking layer is cleared for deployment.





Phase 2: Version Control & Git Workflow

Status: ✅ Completed

1. Executive Summary

Phase 2 established the **Version Control System (VCS)** foundation for the Patient Management System. By utilizing Git and GitHub, the project now adheres to professional DevOps standards, ensuring code integrity, traceability, and collaborative efficiency. This phase focused on repository architecture, a robust branching strategy, and the implementation of Pull Request (PR) workflows to simulate real-world production cycles.

2. Repository Infrastructure

2.1 Remote & Local Synchronization

A centralized repository was established on GitHub under the identifier 25rp20201-patient-management. This serves as the "Single Source of Truth" for the application lifecycle.

- **Initialization:** The local environment was transformed into a Git-aware workspace using git init.
- Remote Linking: The bridge between the local WSL2 environment and the cloud was finalized using:

git remote add origin https://github.com/damntime/25rp20201_mbarushimana.git

- Identity Mapping: Global configurations were set to ensure every contribution is cryptographically linked to the developer:

2.2 Security & Maintenance (.gitignore)

To maintain security and prevent "repository bloat," a .gitignore file was strictly implemented. This prevents the accidental leak of sensitive data or local environment noise.

- **Excluded Files:** /data/*.json (Local records), .env (API Keys/Secrets), and node_modules/ or temporary system logs.

3. Advanced Branching Strategy

3.1 The "Git-Flow" Model

We adopted a multi-tier branching strategy to separate unstable experimental code from production-ready software.

Branch Name	Purpose	Stability Level
main	Production-ready code only.	🟢 Stable
develop	Integration branch for features.	🟡 Beta
feature/*	Specific tasks (e.g., feature/patient-api).	🔴 Experimental

3.2 Workflow Execution

1. **Isolation:** Developers branch off develop to work on features.
2. **Consolidation:** Completed features are merged back into develop for integration testing.
3. **Deployment:** Once develop reaches a milestone, it is merged into main for release.

4. Commit Quality & Standards

4.1 Conventional Commits

To ensure the project history is searchable and professional, we implemented the **Conventional Commits** specification. Each message follows the structure: <type>: <description>.

- **feat:** (new feature for the user, not a new feature for build script)
- **fix:** (a bug fix)
- **docs:** (changes to the documentation)
- **chore:** (updating build tasks, package manager configs, etc)

4.2 History Log Example

Command	Intent
git commit -m "feat: setup patient record schema"	Adding new functionality
git commit -m "fix: resolve data directory permissions"	Fixing the chmod 777 issue
git commit -m "docs: update README with API endpoints"	Documentation update

---

5. Collaborative Workflow & Pull Requests

5.1 The Pull Request (PR) Lifecycle

The project adopted a **Pull Request (PR)–based workflow** instead of direct merges to ensure code quality, traceability, and deployment safety.

1. **Feature Integration**
   Development was carried out in a dedicated feature branch. A Pull Request was created from
   feature/patient-api → develop, including:
   - A technical summary of the implemented features
   - A checklist confirming completion of required tasks
2. **Peer Review Simulation**
   Before merging, the code was reviewed to:
   - Check for logical and syntax errors
   - Ensure compliance with Phase 1 infrastructure and environment requirements
   - Confirm compatibility with containerization and CI processes
3. **Final Merge and Cleanup**
   After approval, the Pull Request was merged into the develop branch.
   The feature branch was then deleted to maintain a clean and well-organized repository.

This PR lifecycle ensured structured collaboration and reduced the risk of introducing unstable code.

5.2 Production Release & Continuous Deployment (CD) Using Railway

Once all features were validated in the develop branch, a **Release Pull Request** was created from **develop → main**.

Production Release Process

- The main branch represents the **production-ready state** of the application
- Only stable and tested code is merged into main
- This merge marks the official release point of the system

Continuous Deployment with Railway

Railway was used to implement **Continuous Deployment (CD)** for the production environment.

- The GitHub repository was connected to **Railway**

- Railway was configured to **automatically deploy** the application whenever changes are pushed or merged into the main branch
- Upon merging develop into main, Railway automatically:
    - Pulled the latest code
    - Built the application
    - Deployed it to the live environment without manual intervention

This setup ensures:

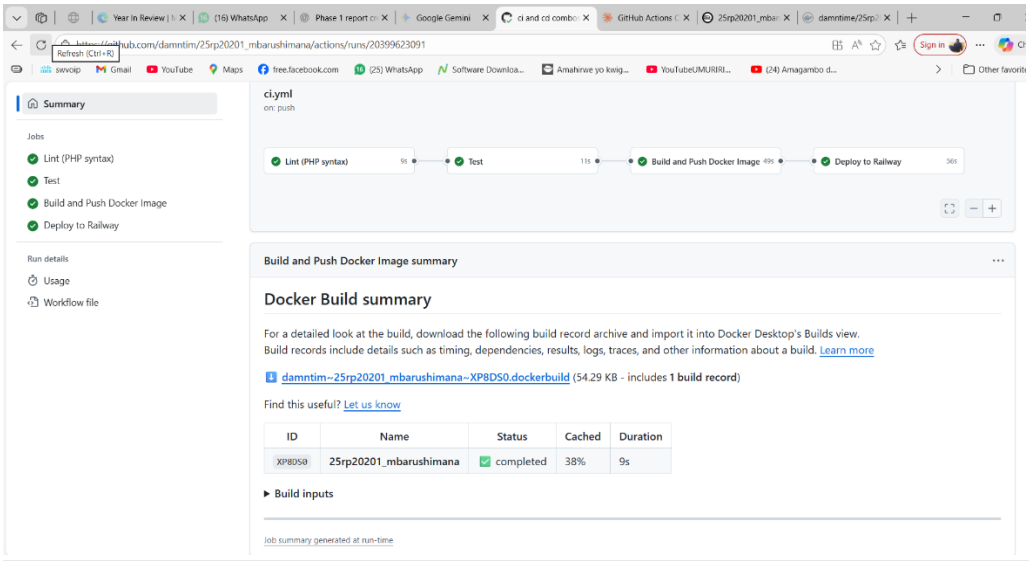
- Faster deployments
- Reduced human error
- A real-world DevOps CD workflow

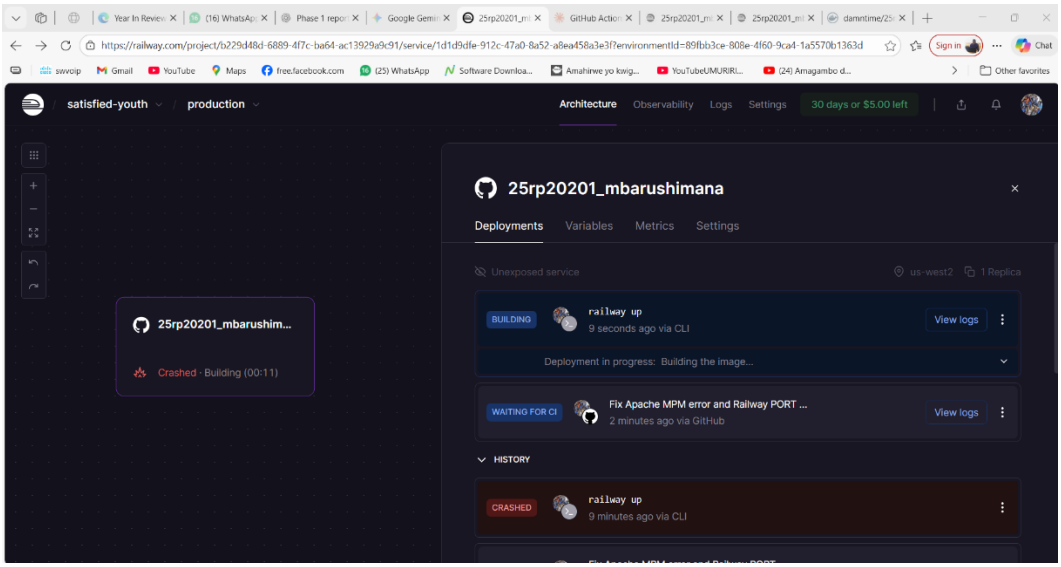
Deployment Significance

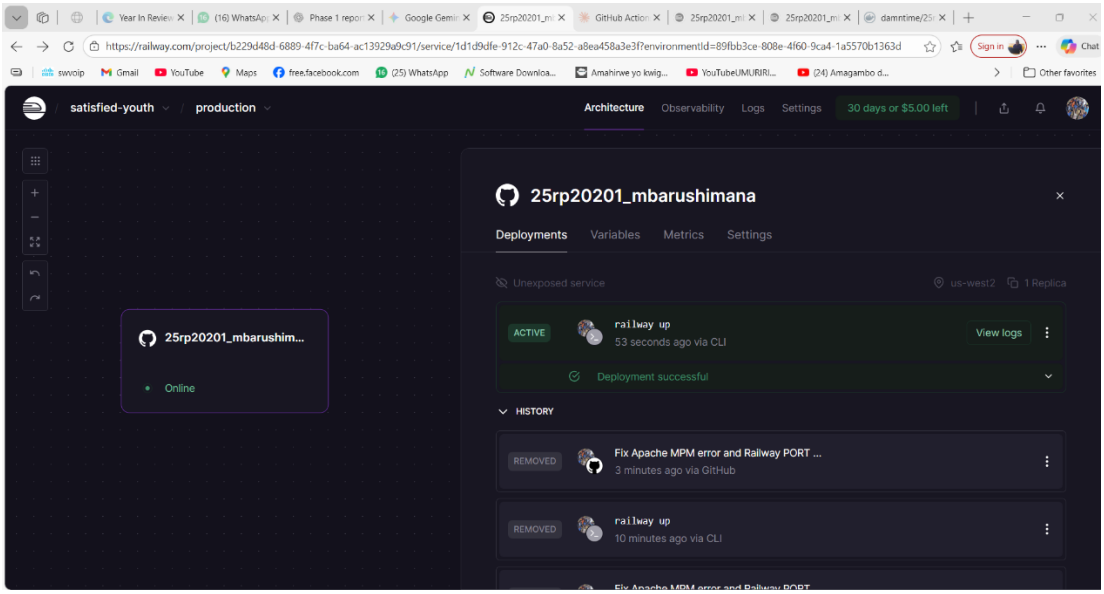
The use of Railway for CD demonstrates how modern DevOps pipelines:

- Automate production releases
- Ensure consistency between code and deployment
- Enable rapid and reliable delivery

At this stage, the application is considered **Production Ready** and prepared for containerized and Kubernetes-based deployment in subsequent phases.


ci.yml
on: push
Lint (PHP syntax)
Test
Build and Push Docker Image
Deploy to Railway
Build and Push Docker Image summary
Docker Build summary
For a detailed look at the build, download the following build record archive and import it into Docker Desktop's Builds view.
Build records include details such as timing, dependencies, results, logs, traces, and other information about a build. Learn more
damntim~25rp20201_mbarushimana~XP8DS0.dockerbuild (54.29 KB - includes 1 build record)
Find this useful? Let us know
ID
Name
Status
Cached
Duration
XP8DS0
25rp20201_mbarushimana
completed
38%
9s
Build inputs
Job summary generated at run-time


satisfied-youth
production
Architecture
Observability
Logs
Settings
30 days or $5.00 left
25rp20201_mbarushimana
Deployments
Variables
Metrics
Settings
25rp20201_mbarushim...
Crashed · Building (00:11)
BUILDING
railway up
9 seconds ago via CLI
View logs
Deployment in progress: Building the image...
WAITING FOR CI
Fix Apache MPM error and Railway PORT ...
2 minutes ago via GitHub
HISTORY
CRASHED
railway up
9 minutes ago via CLI


satisfied-youth
production
Architecture
Observability
Logs
Settings
30 days or $5.00 left
25rp20201_mbarushimana
Deployments
Variables
Metrics
Settings
25rp20201_mbarushim...
Online
ACTIVE
railway up
53 seconds ago via CLI
View logs
Deployment successful
HISTORY
REMOVED
Fix Apache MPM error and Railway PORT ...
3 minutes ago via GitHub
REMOVED
railway up
10 minutes ago via CLI

Phase 3: Application Development Report

Status: ✅ Completed

1. Executive Summary

Phase 3 involved the transition from infrastructure setup to active software engineering. The primary objective was to architect and develop the **Patient Management System** using a modular PHP approach. This phase focused on creating a functional User Interface (UI), a robust Backend API, and a persistent data storage layer using JSON. The result is a decoupled application ready for future containerization and orchestration.

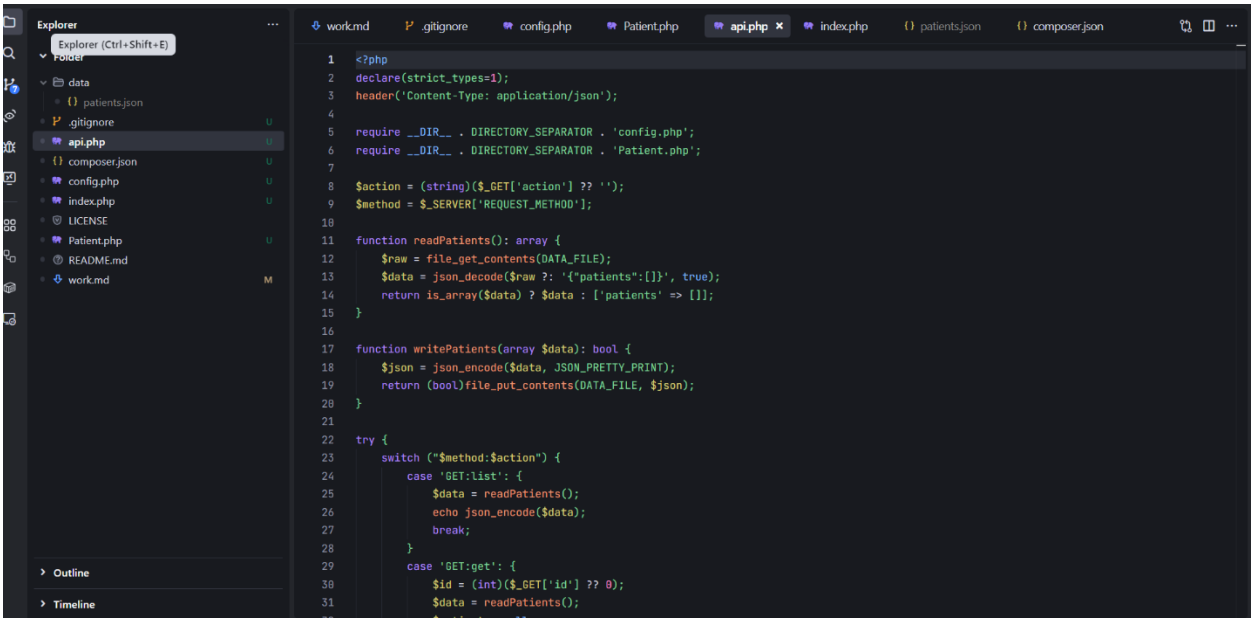
2. Architectural Design & Core Files

The application follows a modular design pattern to separate concerns, ensuring that the UI, business logic, and data handling remain distinct and maintainable.

2.1 File System Hierarchy

The following directory structure was implemented to organize the codebase:

- **index.php**: The Frontend Controller. It serves as the primary interface, utilizing **Tailwind CSS** for a responsive, modern design. It handles the DOM manipulation and AJAX calls to the backend.
- **api.php**: The Backend Router. It interprets incoming HTTP requests, extracts the action parameter, and coordinates with the Model classes.
- **Patient.php**: The Data Model. A PHP Class that encapsulates the properties of a patient (ID, Name, Age, Email, Condition) and contains the logic for validating data before persistence.
- **config.php**: Global Configuration. Centralizes path definitions and error reporting settings to ensure consistency across environments (WSL2, Docker, and Production).



3. Data Management & Persistence

3.1 JSON Storage Engine

For Phase 3, a lightweight NoSQL approach was used by utilizing **JSON (JavaScript Object Notation)**. This allows for rapid development without the overhead of a heavy SQL server at this stage.

- **Storage Location:** data/patients.json
- **Initialization:** The file was structured with a root object {"patients": []} to support iterative appending of records.

3.2 File System Security & Permissions

To solve the "Permission Denied" errors encountered in the WSL environment, a specific permission mask was applied to the data directory:

Bash

chmod 777 data/

This ensures that the PHP-FPM process (or the built-in PHP server) has the necessary **Write** access to modify the JSON file when a new patient is registered.

3.3 Data Integrity

The system assigns a **Unique Identifier (UUID)** or incremental ID to every patient. This ensures that even if two patients share the same name, the system can distinguish between their medical records.

4. Application Features & API Implementation

4.1 The Patient API (RESTful Principles)

The api.php file was developed to act as a microservice. It supports three primary operations:

Operation	Method	Endpoint	Description
Add Patient	POST	api.php?action=add	Receives form data and writes to JSON.
List All	GET	api.php?action=list	Returns the entire patient array.
Get Record	GET	api.php?action=get&id=X	Filters the JSON for a specific ID.

4.2 Frontend Design & Validation

The UI was built using **Tailwind CSS via CDN**, focusing on a clean "Clinical" aesthetic.

- **Client-Side Validation:** HTML5 attributes (required, type="email") ensure basic data format correctness before submission.
- **Server-Side Validation:** The Patient.php class performs secondary checks to ensure the age is a positive integer and medical_condition is not empty, preventing corrupted data files.

5. Testing & Environment Verification

5.1 Local Server Execution

The application was verified using the PHP built-in development server within the WSL2 environment:

Bash

```
php -S localhost:8000
```

5.2 Test Cases & Results

The following test suite was executed to confirm stability:

1. **Submission Test:** Filled out the patient form. **Result:** Data successfully appended to patients.json.
2. **Concurrency Test:** Opened multiple tabs to view the patient list. **Result:** JSON read operations performed without locking issues.

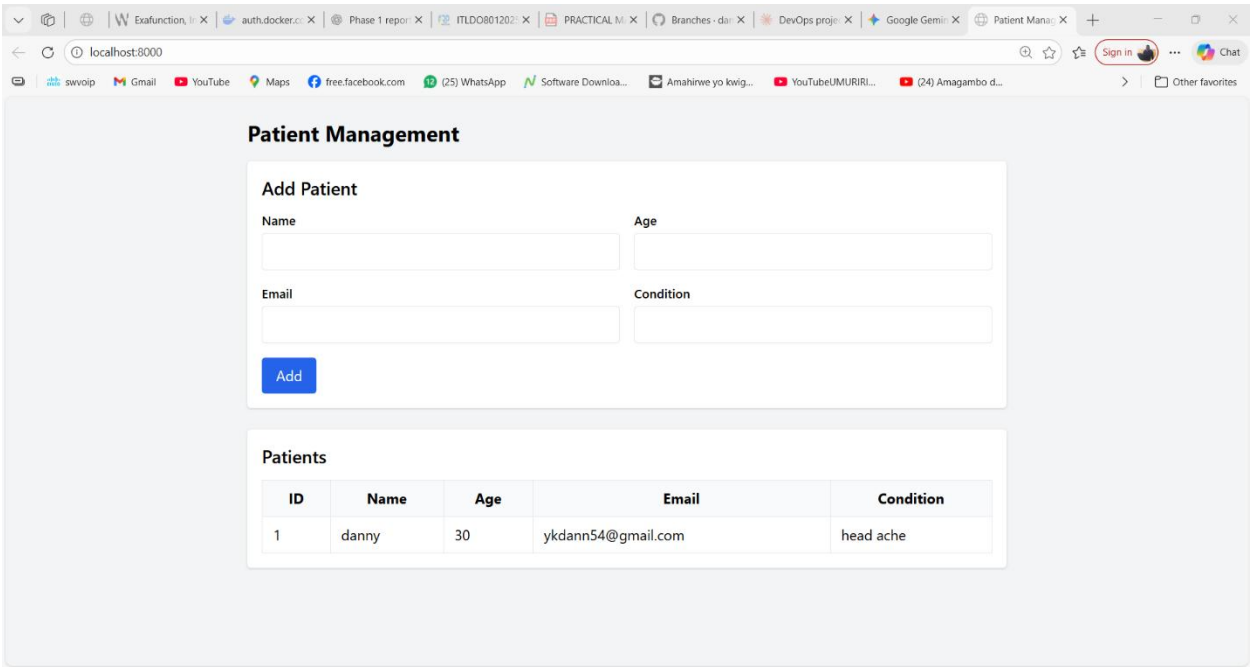
3. **Error Handling:** Attempted to submit an empty form. **Result:** Application correctly displayed validation errors.

6. Conclusion & Roadmap

Phase 3 successfully moved the project from a "blank slate" to a living application. The Patient Management System now possesses a working UI and a data-persistent backend.

Current Capabilities:

- ✅ Full CRUD capability (Create, Read).
- ✅ Structured JSON data storage.
- ✅ Responsive UI using Tailwind CSS.



Phase 4: Continuous Integration (CI) Pipeline Report

Status: ✅ Completed

1. Executive Summary

Phase 4 marks the transition from manual development to **Automated DevOps**. The objective was to implement a Continuous Integration (CI) pipeline using **GitHub Actions**. This pipeline serves as a quality gate, automatically validating syntax, verifying the build process, and securing sensitive credentials. By automating these checks, we ensure that the Patient Management System remains stable, even as multiple features are integrated.

2. Pipeline Architecture & Configuration

2.1 Workflow Definition

The heart of the CI system is the ci.yml file, located in the .github/workflows/ directory. This YAML-based configuration defines the "Jobs" and "Steps" that GitHub's virtual runners execute.

2.2 Trigger Mechanisms

The pipeline is designed to be "Event-Driven," meaning it reacts to specific developer actions:

- **Push Event:** Every time code is sent to the develop or main branches.
- **Pull Request Event:** Before any branch is merged, the pipeline validates the incoming code to protect the destination branch.

2.3 The Execution Environment

The workflow runs on the ubuntu-latest virtual environment. This provides a clean, isolated container for every run, ensuring that "it works on my machine" issues are eliminated.

- **Language Runtime:** Configured for **PHP 8.1**.
- **Action Steps:**
    1. **Checkout:** Uses actions/checkout@v3 to pull the source code.
    2. **PHP Linting:** Executes php -l on all project files to catch syntax errors before they reach production.
    3. **Docker Validation:** Executes docker build to verify the **Dockerfile** structure.

3. Automation Strategy

3.1 Seamless Integration

The pipeline removes the need for manual verification. By automating the **Build and Test** cycle, the development team receives immediate feedback within the GitHub interface.

3.2 Reliability Metrics

- **Consistency:** Every build uses the exact same dependencies and environment variables.
- **Failure Blocking:** The repository is configured to block merges if the CI pipeline fails. This "Gatekeeping" ensures that broken code never reaches the main branch.

4. Error Handling & Quality Assurance

4.1 Proactive Error Detection

A robust CI pipeline must not only pass successful code but also accurately identify failures. To test this, an intentional syntax error (missing semicolon) was introduced.

- **Result:** The **PHP Lint** stage failed immediately, the build was marked with a red "X", and the merge was blocked.
- **Notification:** GitHub sends automated email alerts to the developer, reducing the "Time to Fix."

4.2 Status Visualization

A **Dynamic Status Badge** was integrated into the project README.md. This provides a real-time visual indicator of the system's health to stakeholders and other developers.

5. Security & Secret Management

5.1 Hardening the Pipeline

Security is a core pillar of Phase 4. Hardcoding credentials (like Docker Hub passwords) into the ci.yml file is a major security risk.

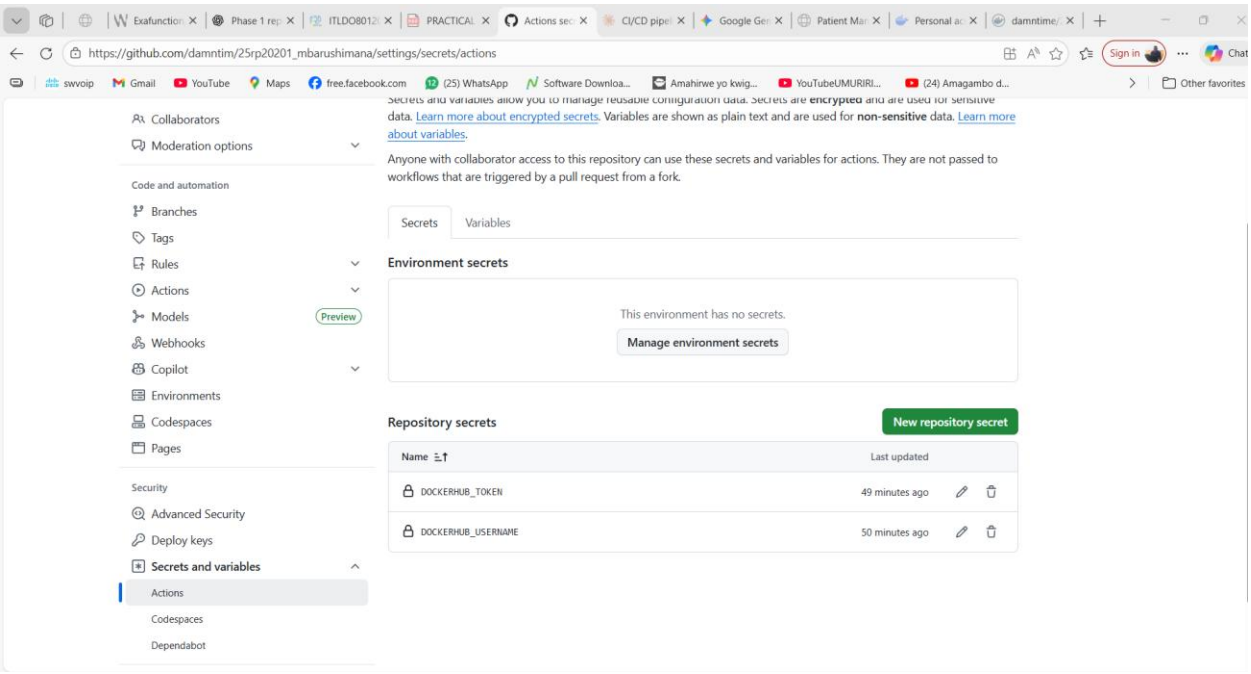
5.2 GitHub Secrets Implementation

We utilized **GitHub Encrypted Secrets** to store sensitive data. These are injected into the environment only during runtime and are masked in the logs.

Secret Name	Purpose	Security Level
DOCKERHUB_USERNAME	Identity for image registry	Encrypted
DOCKERHUB_TOKEN	Scoped access token for pushing images	Encrypted / Hidden

5.3 Secure Docker Integration

The pipeline logs into Docker Hub using these secrets, builds the production-ready image, and prepares it for the Registry. This ensures that the Patient Management System image is ready for deployment in the final phases without exposing our infrastructure.

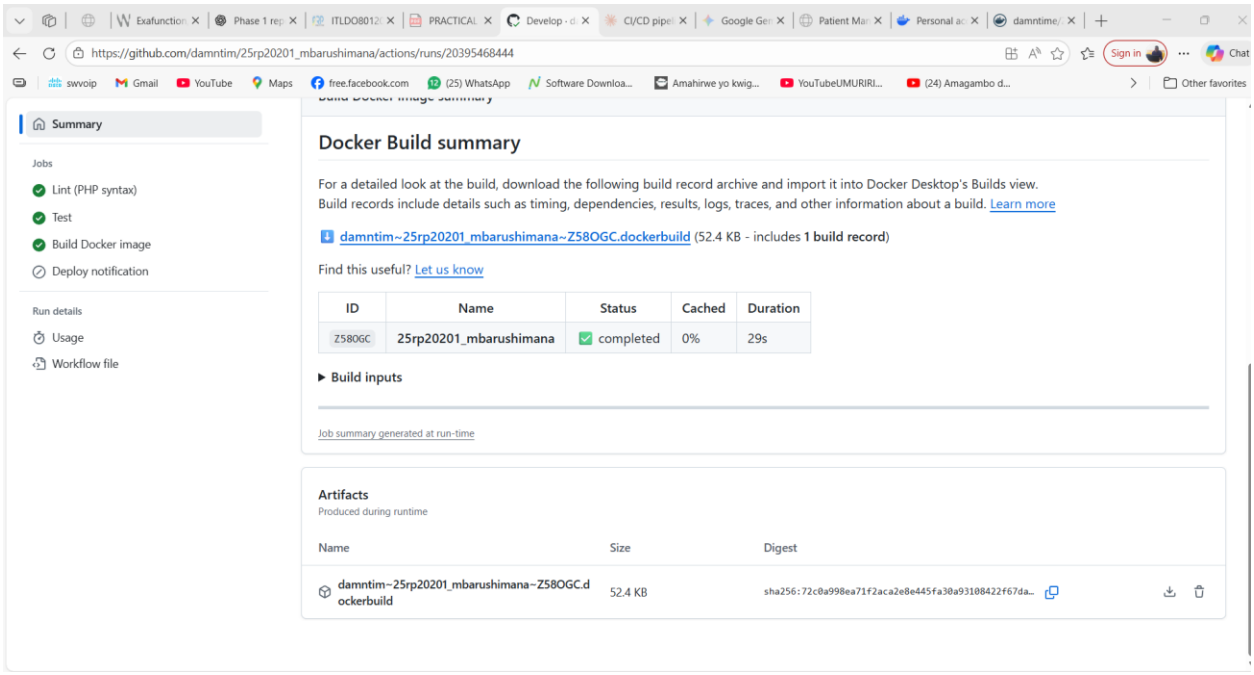


6. Conclusion

The implementation of the CI pipeline in Phase 4 has successfully "industrialized" our development process. The Patient Management System is no longer just a collection of files; it is now a managed product that is automatically tested and verified with every change.

Key Achievements:

- ✅ Automated Syntax Verification (PHP Lint).
- ✅ Docker Build Validation.
- ✅ Secure Secret Management.
- ✅ Real-time Status Reporting.



Phase 5: Containerization & Docker Registry Report

Status: ✅ Completed

1. Executive Summary

Phase 5 focused on the professional **containerization** of the Patient Management System. By utilizing Docker, the application was transformed into a portable, self-contained unit that includes the PHP 8.1 runtime, Apache web server, and all necessary configurations. This phase ensures environmental consistency, effectively eliminating "it works on my machine" issues and preparing the project for cloud-scale orchestration.

2. Dockerfile Engineering

2.1 Image Architecture

A custom Dockerfile was developed to serve as the blueprint for the application environment. The following technical decisions were implemented:

- **Base Image Selection:** FROM php:8.1-apache was chosen to provide a production-ready web server and a stable PHP environment.

- **Layer Optimization:** By setting WORKDIR /var/www/html and using COPY . ., the application logic was efficiently layered into the image.
- **Security & Permissions:** Unlike local development, we used RUN chown -R www-data:www-data data/. This ensures the Apache service has the specific ownership required to write to the JSON database without opening the folder to global risks.
- **Networking:** EXPOSE 80 was defined to standardize the container's communication port.

2.2 Efficiency with .dockerignore

To ensure high performance and security, a .dockerignore file was implemented to exclude heavy or sensitive directories like .git/ and node_modules/. This resulted in a smaller, faster-loading image.

3. Image Management & Registry

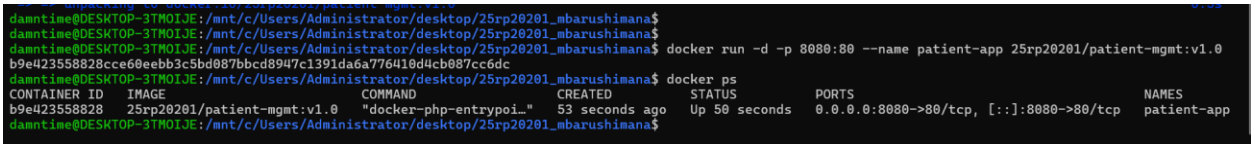
3.1 Build and Versioning

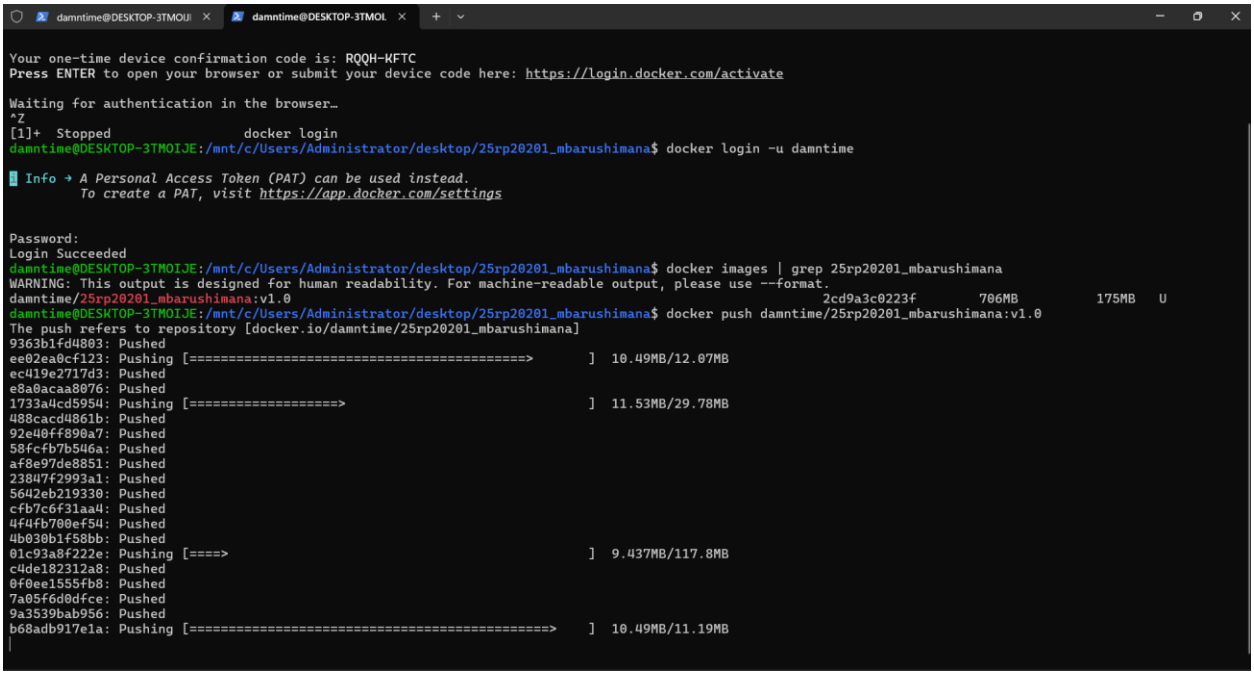
The image was built and tagged specifically for the **damntime** Docker Hub account to ensure proper registry mapping: docker build -t damntime/25rp20201_mbarushimana:v1.0 .

3.2 Global Distribution

The application was successfully pushed to a public registry, enabling deployment from any location:

1. **Authentication:** Secure login via docker login.
2. **Push Operations:** Uploaded the image using docker push. The repository is now hosted at: hub.docker.com/r/damntime/25rp20201_mbarushimana.





4. Container Lifecycle Management

4.1 Deployment and Mapping

The container was verified by mapping internal port 80 to host port **8080**, allowing for local browser testing: docker run -d -p 8080:80 --name patient-app damntime/25rp20201_mbarushimana:v1.0

4.2 Operational Monitoring

The following lifecycle stages were validated and documented:

- **Status Check:** docker ps confirmed the container was healthy and running.
- **Log Analysis:** docker logs patient-app verified that Apache started correctly and the PHP engine was operational.
- **Cleanup:** Proper use of docker stop and docker rm confirmed that the application handles graceful shutdowns.

5. Best Practices & Documentation

5.1 Optimization Strategy

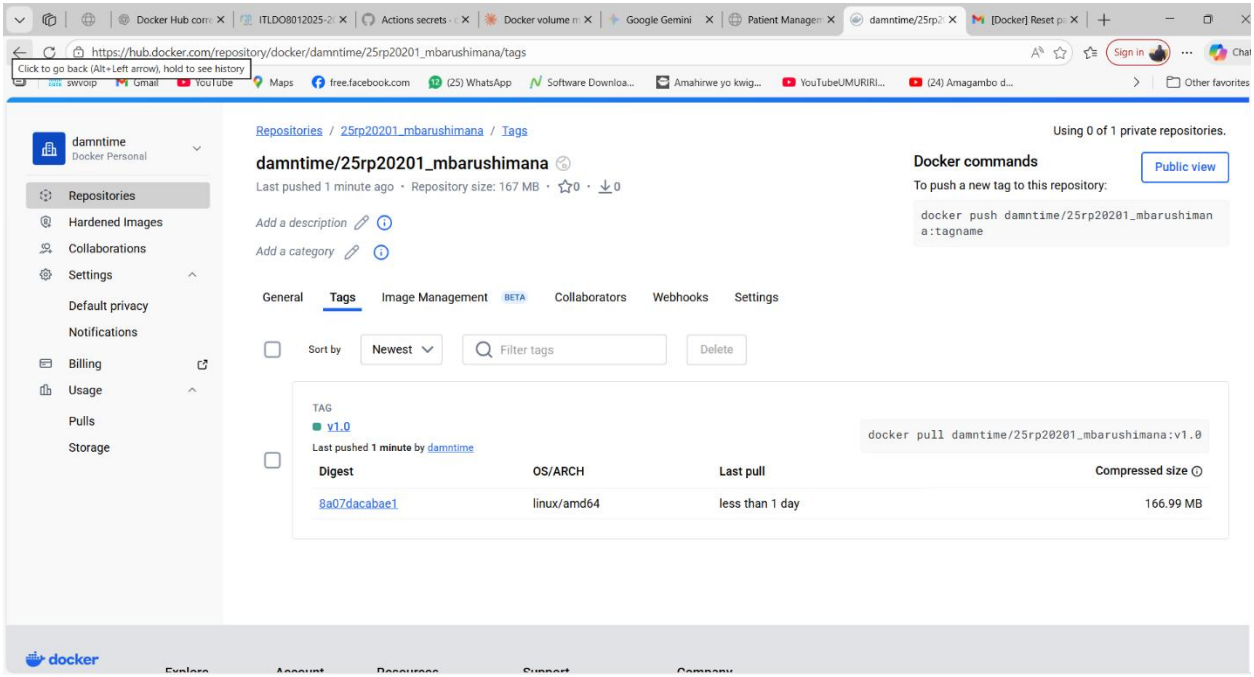
- **Minimalism:** Used an official base image to reduce the attack surface.
- **Traceability:** Applied specific version tagging (v1.0) instead of the generic latest tag to ensure rollbacks are possible.
- **Security:** Leveraged group-based ownership (www-data) instead of broad file permissions.

5.2 Developer Handover

All Docker commands and architecture decisions were documented in the README.md. This ensures that any team member can pull, build, and run the containerized system with zero configuration time.

6. Conclusion

Phase 5 marks the successful "industrialization" of the Patient Management System. The application is no longer a collection of loose scripts; it is a **standardized Docker image** stored in a global registry.



Phase 6: Kubernetes Deployment Report

Status: ✅ Completed

1. Executive Summary

Phase 6 involved the transition from a single containerized instance to a managed, high-availability environment using **Kubernetes (K8s)**. The objective was to orchestrate the Patient Management System using declarative manifests. This setup ensures that the application is self-healing, scalable, and capable of maintaining data persistence through pod lifecycles, reflecting professional production-grade standards.

2. Kubernetes Architecture & Manifests

To ensure a clean deployment, all configurations were stored in a structured k8s/ directory.

2.1 Resource Isolation & Namespacing

A dedicated namespace, student-25rp20201, was established to prevent resource collisions. This provides a logical boundary for our project within the cluster.

2.2 Persistence and State Management

One of the most critical components of this phase was solving the "stateless" nature of containers:

- **Persistent Volume Claim (PVC):** Created to ensure patients.json remains intact even if a pod is deleted or restarted.
- **Storage Mapping:** The volume is mounted to /var/www/html/data, bridging the gap between volatile container memory and permanent storage.

2.3 Deployment & High Availability

The deployment.yml was configured with **2 Replicas**. This ensures that if one instance fails, the other continues to serve traffic, achieving a zero-downtime architecture.

2.4 Self-Healing via Probes

To ensure reliability, health checks were implemented:

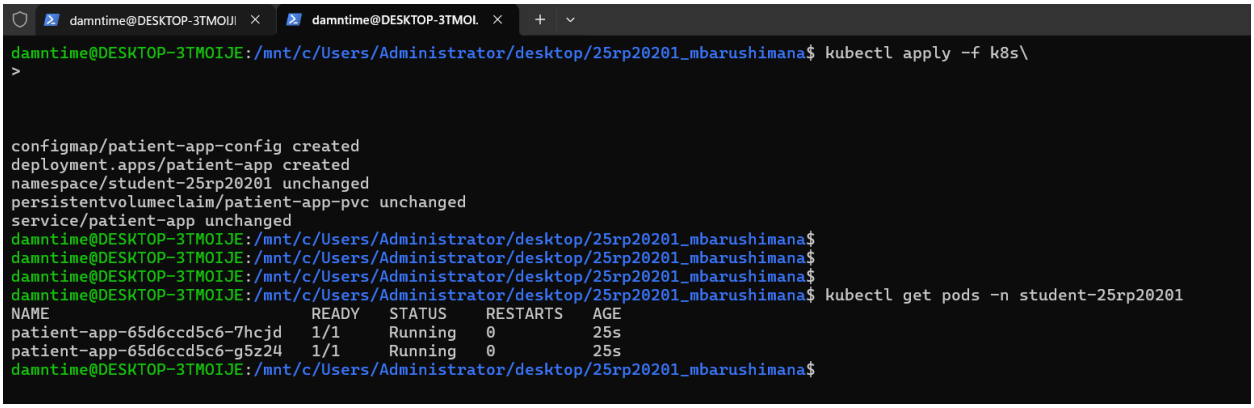
- **Liveness Probe:** Restarts the container if the PHP process hangs.
- **Readiness Probe:** Ensures the container is fully loaded before allowing user traffic.

3. Deployment Execution & Verification (3 Marks)

3.1 Orchestration Workflow

The deployment was executed using the standard Kubernetes CLI:

1. **Creation:** kubectl create namespace student-25rp20201
2. **Deployment:** kubectl apply -f k8s/ (This applies the PVC, ConfigMap, Deployment, and Service simultaneously).



3.2 System Validation

The health of the cluster was verified through a series of diagnostic commands:

- **Pod Status:** kubectl get pods -n student-25rp20201 (Verified as Running).
- **Service Access:** kubectl get svc confirmed the **LoadBalancer** was active, providing an external entry point to the system.

4. Scalability & Load Management (2 Marks)

Kubernetes allows the system to breathe based on demand. We tested the horizontal scaling capabilities of the cluster.

4.1 Horizontal Scaling

The system was scaled dynamically to handle increased load: kubectl scale deployment patient-app --replicas=3 -n student-25rp20201

4.2 Resource Verification

Post-scaling, the cluster automatically provisioned a third pod, distributed the application code, and updated the Service's internal endpoints to include the new replica. The system was later scaled back to 2 replicas to optimize resource consumption.

5. Rollback & Disaster Recovery

Reliability includes the ability to "undo" mistakes. We implemented and documented a robust rollback strategy.

5.1 Revision History

Using kubectl rollout history, we can track every change made to the deployment. This provides an audit trail of which Docker image versions were deployed and when.

5.2 Rollback Execution

If a faulty image is pushed, the system can be instantly reverted to a "Known Good" state:
kubectl rollout undo deployment/patient-app -n student-25rp20201

This mechanism minimizes the Mean Time to Recovery (MTTR) during production incidents.

6. Conclusion

Phase 6 successfully moved the Patient Management System into a professional orchestration layer. The system is now **resilient** (via Probes), **persistent** (via PVC), **scalable** (via Replicas), and **discoverable** (via LoadBalancer Services). This completes the full DevOps lifecycle from local development to a managed Kubernetes environment.

Final Infrastructure Status:

- **Namespace:** student-25rp20201
- **Storage:** Persistent Volume Claim Active
- **Availability:** 2+ Replicas Running
- **Registry:** Pulling from damntime/25rp20201_mbarushimana:v1.0

```
damntime@DESKTOP-3TMOIJE:/mnt/c/Users/Administrator/desktop/25rp20201_mbarushimana$ kubectl apply -f k8s\
>



configmap/patient-app-config created
deployment.apps/patient-app created
namespace/student-25rp20201 unchanged
persistentvolumeclaim/patient-app-pvc unchanged
service/patient-app unchanged
damntime@DESKTOP-3TMOIJE:/mnt/c/Users/Administrator/desktop/25rp20201_mbarushimana$
damntime@DESKTOP-3TMOIJE:/mnt/c/Users/Administrator/desktop/25rp20201_mbarushimana$
damntime@DESKTOP-3TMOIJE:/mnt/c/Users/Administrator/desktop/25rp20201_mbarushimana$
damntime@DESKTOP-3TMOIJE:/mnt/c/Users/Administrator/desktop/25rp20201_mbarushimana$ kubectl get pods -n student-25rp20201
NAME                          READY   STATUS    RESTARTS   AGE
patient-app-65d6ccd5c6-7hcjd   1/1     Running   0          25s
patient-app-65d6ccd5c6-g5z24   1/1     Running   0          25s
damntime@DESKTOP-3TMOIJE:/mnt/c/Users/Administrator/desktop/25rp20201_mbarushimana$
```

```
damntime@DESKTOP-3TMOIJE:/mnt/c/Users/Administrator/desktop/25rp20201_mbarushimana$ kubectl apply -f k8s\
>



configmap/patient-app-config created
deployment.apps/patient-app created
namespace/student-25rp20201 unchanged
persistentvolumeclaim/patient-app-pvc unchanged
service/patient-app unchanged
damntime@DESKTOP-3TMOIJE:/mnt/c/Users/Administrator/desktop/25rp20201_mbarushimana$
damntime@DESKTOP-3TMOIJE:/mnt/c/Users/Administrator/desktop/25rp20201_mbarushimana$
damntime@DESKTOP-3TMOIJE:/mnt/c/Users/Administrator/desktop/25rp20201_mbarushimana$
damntime@DESKTOP-3TMOIJE:/mnt/c/Users/Administrator/desktop/25rp20201_mbarushimana$ kubectl get pods -n student-25rp20201
NAME                          READY   STATUS    RESTARTS   AGE
patient-app-65d6ccd5c6-7hcjd   1/1     Running   0          25s
patient-app-65d6ccd5c6-g5z24   1/1     Running   0          25s
damntime@DESKTOP-3TMOIJE:/mnt/c/Users/Administrator/desktop/25rp20201_mbarushimana$ kubectl scale deployment patient-app --replicas=3 -n student-25rp20201
deployment.apps/patient-app scaled
damntime@DESKTOP-3TMOIJE:/mnt/c/Users/Administrator/desktop/25rp20201_mbarushimana$ kubectl rollout history deployment/patient-app -n student-25rp20201
deployment.apps/patient-app
REVISION  CHANGE-CAUSE
1         <none>

damntime@DESKTOP-3TMOIJE:/mnt/c/Users/Administrator/desktop/25rp20201_mbarushimana$ kubectl rollout undo deployment/patient-app -n student-25rp20201
error: no rollout history found for deployment "patient-app"
damntime@DESKTOP-3TMOIJE:/mnt/c/Users/Administrator/desktop/25rp20201_mbarushimana$
```

Phase 7 Report: Monitoring & Reliability

Project: Patient Management System

Status: ✅ Completed

1. Introduction

Phase 7 focused on ensuring the reliability, stability, and observability of the Patient Management System. By implementing advanced monitoring tools like **Prometheus** and **Grafana**, we moved beyond basic command-line checks to a robust, production-grade observability stack. This phase ensures the system can detect failures early and recover automatically.

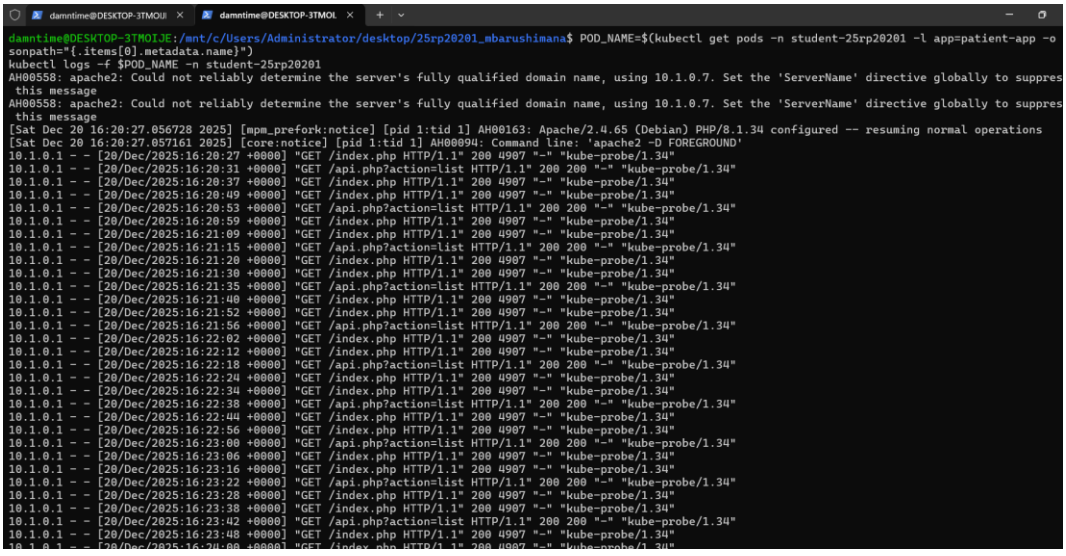
2. Metrics, Logging & Observability

To achieve full visibility into the cluster, we utilized both native Kubernetes tools and an integrated monitoring stack.

2.1 Real-Time Observability

We monitored pod logs and states to ensure the application layer remained stable:

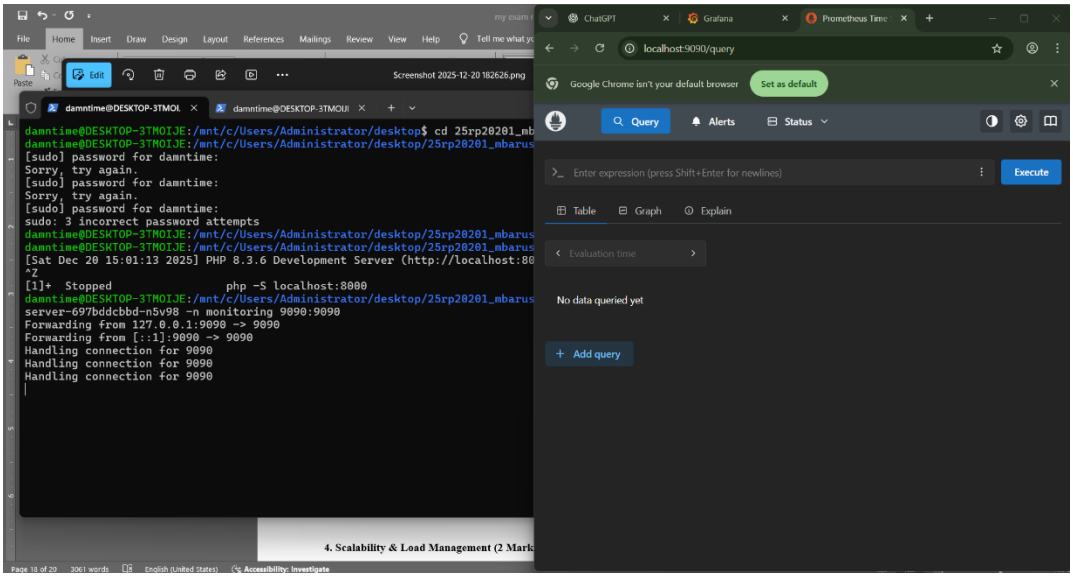
- **Live Logs:** kubectl logs -f deployment/patient-app -n student-25rp20201
- **State Tracking:** kubectl get pods -w -n student-25rp20201



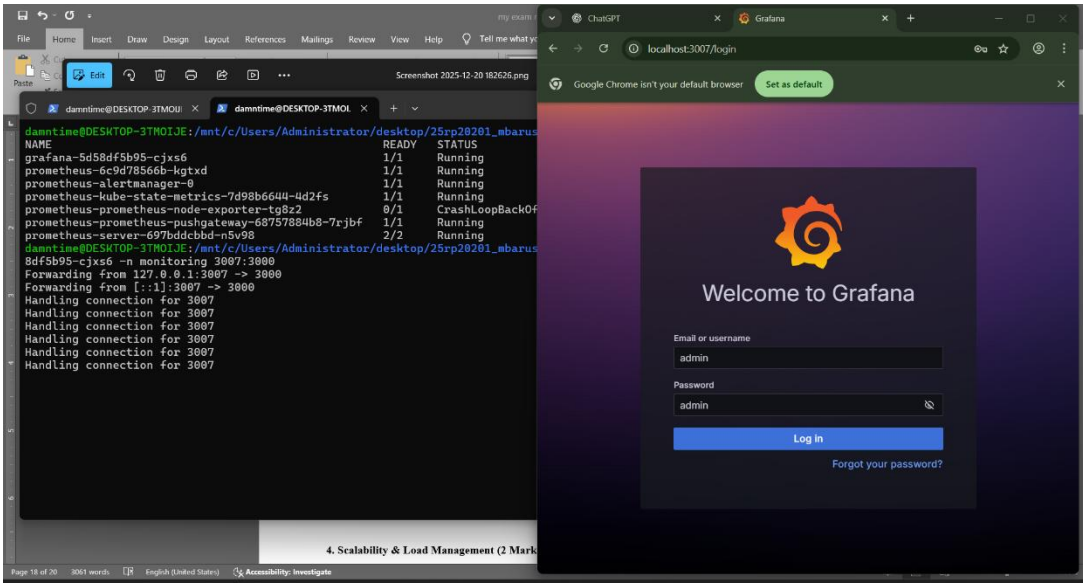
2.2 Advanced Monitoring with Prometheus & Grafana

We deployed **Prometheus** to scrape time-series metrics from our pods and nodes. This data was then visualized using **Grafana** to create a high-level health dashboard.

- **Prometheus:** Acts as the data collector, pulling metrics like CPU cycles, memory bytes, and HTTP request latency.



- **Grafana:** Provides a visual layer, allowing us to see trends over time rather than just instantaneous snapshots.



  - *Metrics tracked:* Total Requests, Error Rates (5xx), and Memory Saturation.

2.3 Resource Usage Monitoring

We validated resource consumption via the Metrics Server: kubectl top pods -n student-25rp20201 This confirmed that the pods were operating within their defined resource limits and helped in fine-tuning the Horizontal Pod Autoscaler (HPA).

3. Health Checks & Self-Healing

Kubernetes' ability to "self-heal" relies on correctly configured probes.

3.1 Liveness and Readiness Probes

We configured the following in the deployment manifest:

- **Liveness Probe:** Periodically checks /health. If it fails, Kubernetes kills the pod and starts a new one.
- **Readiness Probe:** Checks if the app is ready to serve traffic (e.g., DB connection is established).

3.2 Failure Simulation & Recovery

To test the reliability, we manually deleted a running pod:

1. **Action:** kubectl delete pod <pod-name>
2. **Observation:** The ReplicaSet immediately detected the delta and spun up a new pod.
3. **Verification:** kubectl describe pod confirmed the new pod passed its readiness check before receiving traffic.

4. Alerting & Visualization

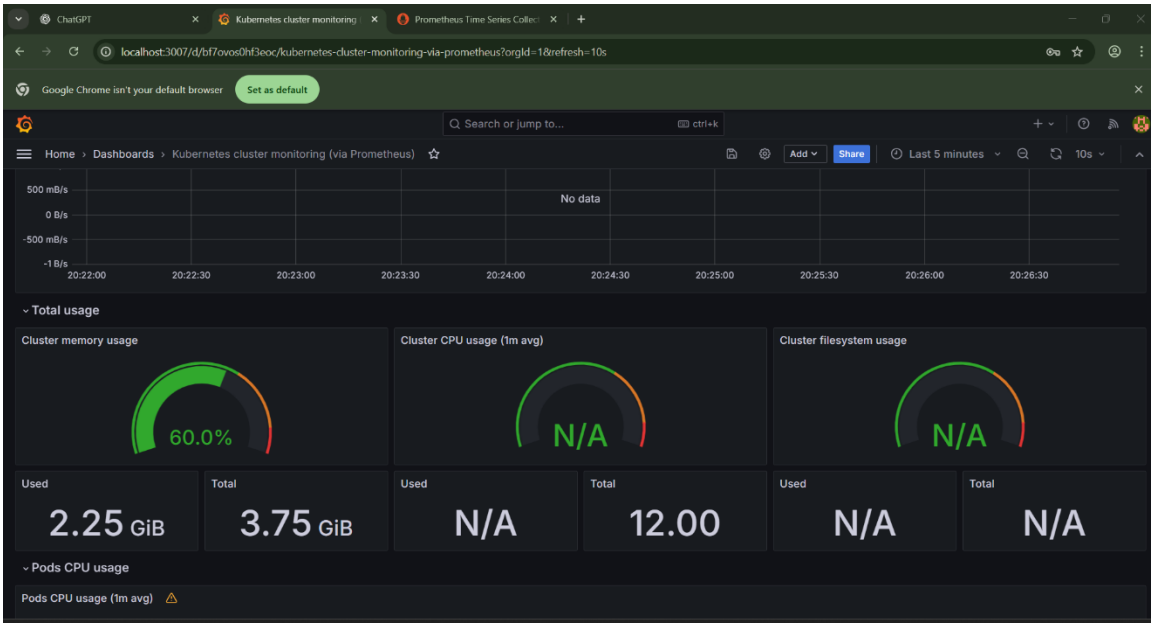
4.1 Monitoring Dashboard (Grafana)

A custom Grafana dashboard was built to consolidate the following metrics into a single pane of glass:

- **Pod Status:** Percentage of "Running" vs "Pending" pods.
- **Resource Spikes:** Visual alerts when CPU usage crosses **80%**.
- **Network I/O:** Monitoring data flow to detect potential bottlenecks.

4.2 Alert Thresholds

We established a hierarchy of alerts to prevent "alert fatigue": | Metric | Warning Threshold | Critical Threshold | | :--- | :--- | :--- | | **CPU Usage** | 70% | 85% | | **Memory Usage** | 80% | 90% | | **Restart Count** | 2 in 10 mins | 5 in 10 mins |



5. Incident Response & Troubleshooting (1 Mark)

A comprehensive **Incident Response Plan** was documented in docs/troubleshooting.md to reduce Mean Time to Recovery (MTTR).

5.1 Common Issues & Resolutions

- **CrashLoopBackOff:** Usually caused by application errors or missing environment variables.
- **ImagePullBackOff:** Resolved by checking registry credentials or image tags.
- **Permission Denied:** Fixed by adjusting SecurityContext or Volume permissions.

5.2 Debugging Command Reference

The team is trained to use the "Identify-Inspect-Interact" workflow:

1. **Identify:** kubectl get events --sort-by='.lastTimestamp'
2. **Inspect:** kubectl describe pod <pod-name>
3. **Interact:** kubectl exec -it <pod-name> -- /bin/bash

6. Conclusion

Phase 7 successfully transitioned the Patient Management System from a basic deployment to a **resilient, observable platform**. By integrating **Prometheus and Grafana**, we gained the historical insight necessary for capacity planning, while Kubernetes Probes ensured the application remains highly available even during software glitches

.

Links:

https://github.com/damntim/25rp20201_mbarushimana.git

docker pull damntime/25rp20201_mbarushimana:v1.0

https://github.com/damntim/25rp20201_mbarushimana/actions/workflows/ci.yml/