F21AO REPORT

Ops Part

Group Members

- Mohamed Aman – ma2305@hw.ac.uk
- Faisal Chaudhry- fmc4000@hw.ac.uk
- Mohamed Aqib Abid - ma4034@hw.ac.uk
- Syed Arif Ali - sa4001@hw.ac.uk
- Chen Bin - cb3000@hw.ac.uk

Table Of Contents

1 Overview

This report details the Ops Phase of the Patient Information System (PIS). The system is designed for efficient and secure patient data management, including authentication, patient registration, and treatment records.

The Ops Phase is a crucial component of the DevOps lifecycle, where we focus on ensuring automated deployment, security enforcement, and continuous monitoring of the system. This phase is dedicated to implementing strategies for seamless software deployment, efficient resource utilisation, and high system availability. Through containerisation, CI/CD pipelines, automated testing, and security audits, we ensure that the application remains stable and scalable in different environments, including development, staging, and production.

This phase also ensures that best DevOps practices such as automated workflows, infrastructure as code (IaC), and active monitoring are in place, thereby reducing manual intervention and increasing deployment reliability. We employ Docker for containerization, Jenkins for CI/CD automation, SonarQube for static code analysis, and Nagios for real-time monitoring, among other tools.

1.1 Key DevOps Tools Used

- Docker - For containerization of microservices.
- Jenkins - Automating CI/CD pipelines.
- SonarQube - Performing static code analysis.
- Mocha & Chai - Conducting unit and integration testing.
- OWASP ZAP - Identifying security vulnerabilities.
- Nagios - Enabling real-time system monitoring.
- Digital Ocean – Cloud Hosting and deployment of the application using virtualization.
- Azure Kubernetes Service (AKS) – Cloud deployment of Kubernetes

By implementing these tools, we ensure the system remains scalable, secure, and maintainable with efficient operational workflows.

2 DevOps Strategy

2.1 Key Objectives:

Our DevOps project strategy emphasizes collaboration between development and operations teams to enhance reliability, scalability, and efficiency while enabling faster software delivery lifecycles. Key components of the strategy include:

- **Agile Methodology -** The project follows Agile principles, allowing for iterative development and continuous feedback. This approach ensures that the system can adapt to changing requirements and improve over time.
- **Automation** - Automation is a core principle of DevOps, enabling faster and more reliable deployments. Tools like Jenkins are used to automate the CI/CD pipeline, while Docker facilitates containerization.
- **Scalability** - Leveraging containerized microservices on cloud infrastructure using Docker.
- **Reliability** - Ensuring system stability via continuous monitoring and alerting using Nagios.
- **Collaboration** - The use of tools like JIRA for project management fosters collaboration among team members, ensuring that everyone is aligned on project goals and progress.
- **Security -** Integrating security tools at every stage. Security is integrated into the DevOps process through continuous testing and monitoring. Tools like OWASP ZAP are employed to identify vulnerabilities early in the development cycle.

2.2 Critical Analysis and Implementation Plan

A key aspect of our DevOps approach was containerization with Docker. By packaging the application into Docker containers, we eliminated environment inconsistencies and facilitated smooth deployments across development, testing, and production environments.

To automate our development pipeline, we implemented CI/CD (Continuous Integration and Continuous Deployment) using Jenkins. With this in place, new code changes are automatically built, evaluated, and deployed, reducing manual intervention and accelerating release cycles.

Tools & Techniques Implemented:

- **Testing** - Used Mocha and Chai for unit and integration testing, allowing us to catch issues early before they reach production.
- **Static Security Application Testing** - Integrated SonarQube for static code analysis to identify vulnerabilities and enforce coding standards.
- **Dynamic Security Penetration Testing** - Implemented OWASP ZAP scans to detect potential security threats, ensuring the system remains protected against attacks.
- **Monitoring** - Integrated Nagios for real-time tracking of system health, enabling active resolution of potential failures before they impact users.

By implementing this DevOps strategy, we streamlined our development process, improved security, and ensured the system remained reliable and scalable. This approach also allowed teams to work more collaboratively, ultimately delivering a robust and well-maintained patient information system. We have hosted the system on Digital Ocean droplets to provide a secure, scalable, and easily accessible platform for our teams.

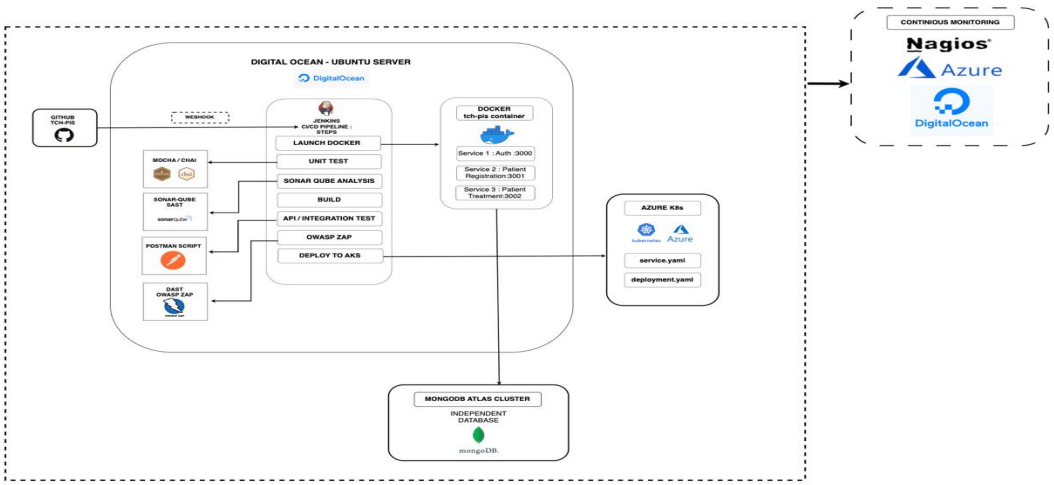

Fig. 1 – Overall Architecture Diagram of Application, Tools and Pipeline

3 Containerization & Virtualization

Containerization and virtualization are essential for deploying the application in a scalable and efficient manner. Docker containers were chosen due to their lightweight and portable nature, allowing for efficient resource utilization and consistent environments across development, testing, and production.

Digital Ocean virtual machines offered practical virtualization for development and testing environments, enabling realistic simulation outside local setups. Cloud deployment leveraged Azure Kubernetes Service (AKS) for its advanced orchestration capabilities, auto-scaling features, and robust monitoring via Azure Monitor.

3.1 Docker Containerization:

- Docker is used to create lightweight, portable containers where all the microservices are wrapped in a single Docker container.
- The Dockerfile defines base image (Node.js Alpine), working directory, dependencies installation, and a start script for concurrent service execution.
- Containers can be easily scaled up or down based on demand, allowing the system to handle varying loads efficiently.
- Docker ensures that the application runs the same way in development, testing, and production environments, reducing the risk of environment-related issues.

3.2 Virtualization:

- Digital Ocean VMs (Droplets) were used to simulate a real-world virtualized environment for testing service behavior outside development machines.
- Jenkins, OWASP ZAP, SonarQube and docker daemon are used on this VM

3.3 Cloud Deployment:

- Final deployment was done using Azure Kubernetes Service (AKS) for its managed orchestration features, scalability, and integration with Azure DevOps.
- **Automated Management** - AKS automates the deployment, scaling, and management of containerized applications, simplifying operations.
- **Load Balancing** - AKS provides built-in load balancing, ensuring that traffic is distributed evenly across containers.
- **Monitoring and Logging** - Integration with Azure Monitor allows for comprehensive monitoring and logging of containerized applications, facilitating proactive management.
- **deployment.yaml** - Describes a Recreate strategy, deploying all services under one container image.
- **service.yaml** - Exposes the ports 3000 (auth), 3001 (registration), and 3002 (treatment) via a Load Balancer.
- **Container Registry -** Docker images are hosted on Docker Hub (faisalmch/tch-pis-k8s:1.2).

3.4 Deployment Steps:

1. Jenkins builds and tags Docker image.
2. Pushes image to Docker Hub.
3. Runs kubectl apply commands to deploy YAML configurations.
4. Azure AKS provisions required resources and starts the pods.

4 CI/CD Implementation

The CI/CD pipeline is a critical component of the Ops phase, enabling rapid and reliable software delivery.

Tools Used: Docker , Jenkins using Groovy for pipeline definition, SonarQube , Mocha, Chai, and Sinon

4.1 CI/CD Pipeline Steps

Jenkins is the primary tool used for implementing the CI/CD pipeline. Key features include:

- Automated Builds: Jenkins automates the build process, ensuring that code changes are compiled and tested automatically.
- Integration with Version Control: Jenkins integrates with Git to trigger builds on code commits, ensuring that the latest changes are always tested.
- Pipeline as Code: Jenkins allows for defining the CI/CD pipeline using code, making it easier to version and manage.

Fig. 2 – Pipeline Diagram of Application on Jenkins

Stage 1: Code Commit and Launch Docker Container

- Developers commit code changes to the Git repository.
- Uses docker run with port mapping.
- Prepares the container for the build and test stages.
- **Pipeline Triggering:** Triggered on commit/push to main GitHub Repository
- Jenkins triggers a build process to compile the code and run unit tests.

Stage 2: Unit Testing

- Executes npm run test on all services sequentially using Mocha, Chai, and Sinon.
- Test failures halt the pipeline.

Stage 3: Static Code Analysis

- SonarQube scanner runs with credentials from sonar-project.properties.
- Reports are sent to the SonarQube dashboard.

Stage 4: Continuous Testing & Security testing leading to a Conditional Build/Deploy

- If tests and scans succeed, npm run start-all is executed to start services.
- API integration testing is carried out and if it passes then deployment is carried out otherwise on failure the pipeline is stopped and goes to post condition.
- Continuous testing is integrated into the CI/CD pipeline to ensure that the application is secure, and functions as expected.

Stage 5: Deployment

- Kubectl deployment command triggers container deployment to AKS.
- Containers are deployed to the AKS environment.
- Automated tests are run to validate the deployment.

Stage 5: Cleanup

- Stops and removes Docker containers to ensure a clean environment.

4.2 Pipeline Technical Guidelines

Each stage of the pipeline has a specific technical rationale: :

- **Dockerization**: Ensures consistency across different environments.
- **Unit Testing**: Early bug detection and validation of business logic.
- **Static Analysis with SonarQube**: Improves code quality, reduces technical debt.
- **Security Testing with OWASP ZAP**: Essential for proactively mitigating vulnerabilities.
- **Kubernetes Deployment**: Manages containers efficiently at scale with minimal downtime.

Technical guidelines include structured YAML files (deployment.yaml, service.yaml), Dockerfile specifying node.js alpine base images for reduced vulnerabilities, and Jenkins pipelines scripted in Groovy for version control and clarity.

5 Continuous Testing and Security

5.1 Testing Implementation Overview

Continuous testing is practice of automating tests at every stage of software delivery pipeline to ensure regular feedback and high-quality releases.

In our pipeline we have performed 3 types of tests at distinct stages namely:

1. Unit testing: Performed immediately after containerizing the application using Docker. Mocha, Chai, Sinon tests verifying logic at granular levels.
2. API testing and Integration testing: Performed after all services were up and running in Docker and Kubernetes. Validates interactions between services and business logic via Postman automated tests.
3. Continuous Integration testing performed for all our three critical services:
    a. Auth Service
    b. Patient Registration Service
    c. Patient Treatment Service

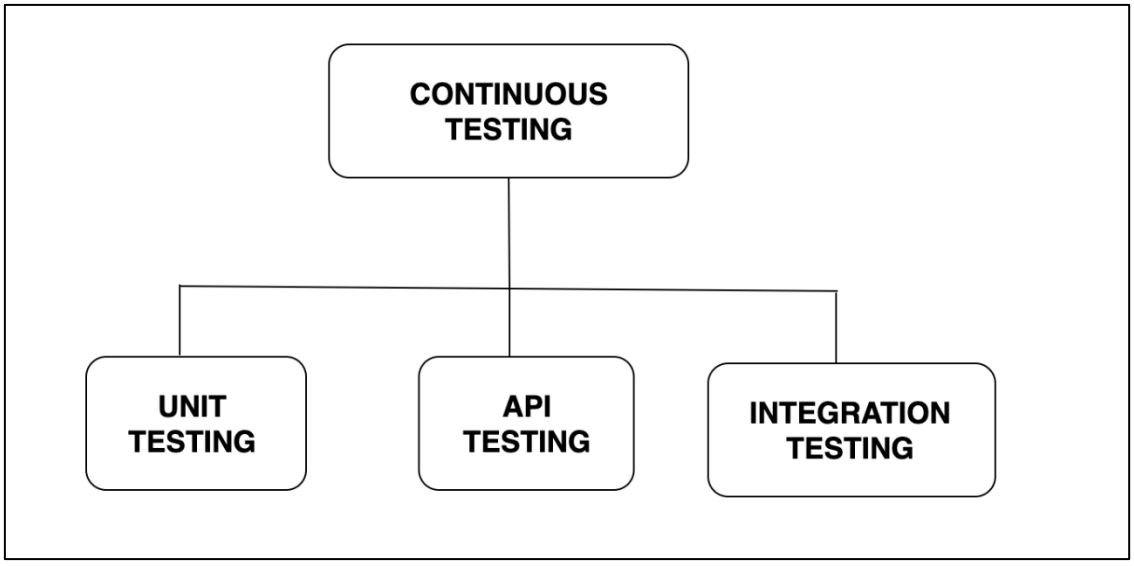

Fig. 3 – Overall Testing Architecture Diagram of Application

5.1.1 Unit testing

This test particularly targeted individual code components such as functions in isolations.

Helped in our application:

1. To catch bugs early
2. Validate the core logics of all our 3 services.
3. Improved the reliability of authentication, patient registration, and treatment workflows.

Tools Used:

1. Moch and Chai: For writing test cases and assertions
2. Sinon: For mocking purpose

Coverage: Covered both positive and negative test cases.

5.1.1.1 Tests Implemented

Details of the implemented unit tests are:-

1) AUTH – REGISTRATION:
    i) POST AUTH – REGISTRATION
        (a) Positive test case
            (i) Successful registration: Validates happy path where all fields are correct.
        (b) Negative test cases:
            (i) Username already exists: Ensures that duplicate usernames are blocked.
            (ii) Fields missing: Will throw an error if either of fields are missing.
            (iii) Wrong role: Will return an error if the role is invalid.
    ii) AUTH – LOGIN
        (1) POST AUTH – LOGIN
            (a) Positive test cases
                (i) Successful login-in: Validates happy path.
            (b) Negative test cases:
                (i) Username not found: Should return an error.
                (ii) Password incorrect: Validates if password not matching database, will throw an error.
                (iii) Server error: When fails to fetch from server, should throw an error.

2) PATIENT- REGISTRATION
    i) POST PATIENT RECORD
        (a) Positive test case:
            (i) Successful patient registration: happy flow
        (b) Negative test cases:
            (i) Mobile number already exists: When 2 users try to register using same mobile-number will throw an error, duplication not allowed.
            (ii) Server error: Should throw an error.
            (iii) Incomplete patient registration: Any of mandatory fields are missing, will throw an error.
    ii) GET – PATIENT REGISTRATION RECORD
        (a) Positive test case
            (i) Successfully returning all patient record
        (b) Negative scenarios:
            (i) Server error: Should throw an error.
            (ii) Return empty record: When no such patient record exists.
    iii) GET PATIENT REGISTRATION RECORD BY ID
        (a) Positive test case
            (i) Successful retrieval
        (b) Negative test cases:
            (i) Patient record not found.
            (ii) Server error
            (iii) Validating all patient fields.
3) PATIENT TREATMENT
    a) POST – CREATE DIAGNOSIS
        (a) Positive test case:
            (i) Successfully create a new diagnosis
        (b) Negative scenario
            (i) Missing mandatory fields: Should return an error.
            (ii) Server error.

b) UPDATE MEDICATION
    (a) Positive scenario:
        (i) Successfully update medications.
    (b) Negative scenario:
        (i) Throw an error if the medication is missing.
        (ii) Medication send should be an array.
        (iii) If no such medication exist, should throw an error.
        (iv) Database error
        (v) Remove medication.
c) REMOVE MEDICATION
    (a) Positive test case:
        (i) Successfully remove medication.
    (b) Negative Scenarios:
        (i) Deleting a record which doesn't exist should throw an error.
        (ii) If medication doesn't exist in the database, should throw an error.
        (iii) Database error
d) GET TREATMENT RECORD:
    (a) Positive scenario:
        (i) Should return medication successfully.
    (b) Negative scenario:
        (i) Throw an error if no such record exists.
        (ii) Should throw an error if treatment exist but no medication present.
        (iii) Server error
e) ADDING VITALS POST
    (a) Positive scenario:
        (i) Successfully add vitals.
        (ii) Creating a new record if no record exists
    (b) Negative scenario:
        (i) Mandatory fields are missing.
        (ii) Database error.

f) GET PATIENT TREATMENT BY ID:

(a) Positive scenario:

(i) Successfully return a record for a particular valid patient id

(b) Negative scenario

(i) Throw an error if no patient treatment record found.

(ii) Should throw an error if invalid patient-id is passed.

(iii) Database server error.

5.1.2 Api Testing

This process involves validating the functionality, reliability, performance, and security of API's. This is important to ensure that all our services are running correctly and meeting expected requirements.

Helped us:

1. Validate our business logic.
2. Reduced our testing time.
3. Increase test coverage for core services.

5.1.3 Integration Testing

Integration testing services used our application work well together. It focuses on how multiple services talk to each other.

Helped us:

1. Detect the issues between different component interacting with each other.
2. Helped us validate integrated components.
3. Verify if system infrastructure and pipeline work as intended collectively.
4. Identify the issue which unit tests missed

5.1.3.1 Integration testing Implementation strategy

Integration testing follows a specific workflow, particularly for patient-treatment and patient record functionality:

1. First we call the authentication service.
2. Obtain token from auth-service.
3. Token will be placed in bearer token authorization header.
4. Then we run the patient-treatment and patient record API calls.

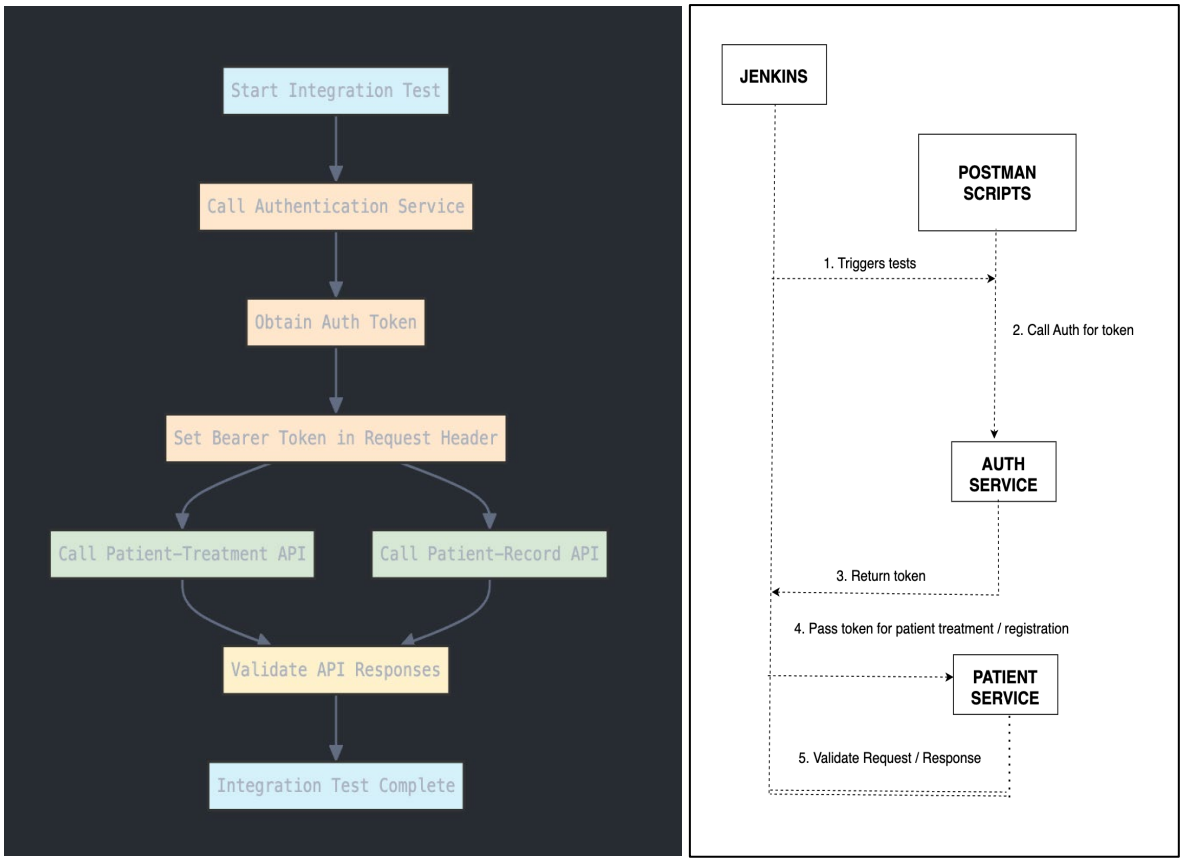

Fig. 4 – Flow for Integration test , API test and Flow in Jenkins pipeline

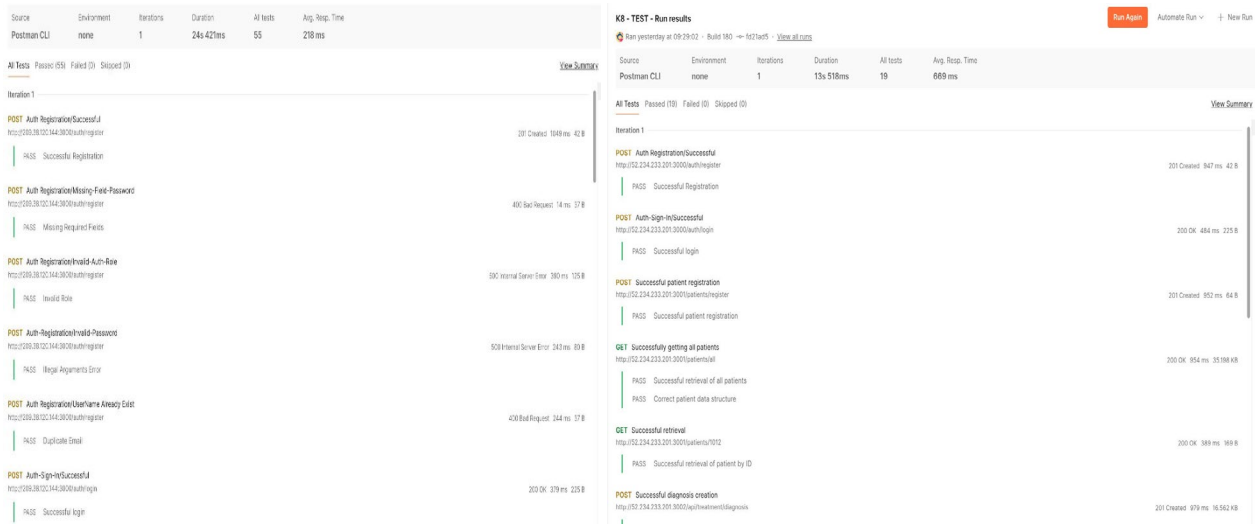

Fig. 5 – API / Integration Testing and Sanity Check Post Deployment on Kubernetes

5.2 Security Measures

- **Static Analysis:** SonarQube reports vulnerabilities like SQL injections, weak hash functions, and code smells.
- **OWASP ZAP:** Actively scans running containers for common exploits such as XSS, broken authentication, and token exposure.
- **JWT Auth:** All routes require valid JWT tokens and role checks (clerk, doctor, nurse).
- **Data Handling:** Sensitive fields are validated and sanitized at model and controller level.

Recommendations for enhancement include integrating automated threat modeling frameworks (e.g., OWASP Threat Modeling), and expanding risk assessments proactively.

5.2.1 SonarQube (Static Application Security Testing - SAST)

SonarQube is an essential tool for maintaining code quality and ensuring that the application adheres to best practices. Its integration into the CI/CD pipeline provides several benefits:

5.2.1.1 Static Code Analysis

Code Quality Metrics: SonarQube analyses the codebase for various quality metrics, including code smells, bugs, and vulnerabilities. This analysis helps maintain high standards of code quality throughout the development process.

Continuous Feedback: By integrating SonarQube into the Jenkins pipeline, developers receive immediate feedback on code quality after each commit. This allows for quick identification and resolution of issues before they escalate.

5.2.1.2 Technical Debt Management

Tracking Technical Debt: SonarQube provides insights into technical debt, allowing the team to prioritize refactoring efforts and improve the overall maintainability of the codebase.

Quality Gates: SonarQube can enforce quality gates, which are thresholds that the code must meet before it can be merged into the main branch. This ensures that only high-quality code is deployed to production.

5.2.1.3 Reporting and Visualization

Dashboards: SonarQube offers dashboards that visualize code quality metrics, making it easier for the team to monitor progress and identify areas for improvement.

Historical Analysis: The tool tracks changes in code quality over time, allowing the team to assess the impact of their efforts and make data-driven decisions.

5.2.2 OWASP ZAP (Dynamic Application Security Testing - DAST)

OWASP ZAP is utilized for dynamic application security testing (DAST). It helps identify vulnerabilities in the application during the testing phase. Key aspects include:

Automated Scanning: OWASP ZAP can be configured to run automated scans against the application, identifying common security issues such as SQL injection, cross-site scripting (XSS), and insecure configurations.

Integration with CI/CD: The tool is integrated into the Jenkins pipeline, allowing security tests to run automatically after each deployment, ensuring that vulnerabilities are detected early.

Reporting: OWASP ZAP generates detailed reports on identified vulnerabilities, which can be used to prioritize fixes and improve the overall security posture of the application.

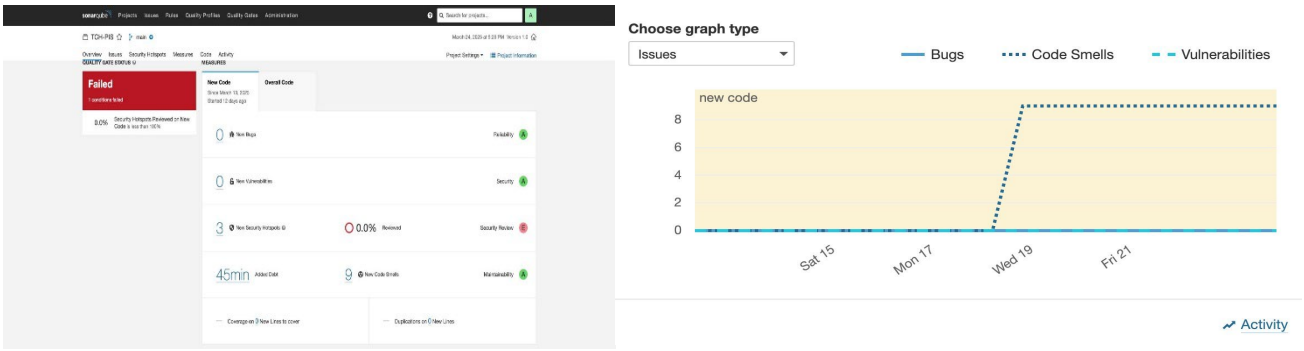

Fig. 6 – SonarQube Report Dashboard and Alerts

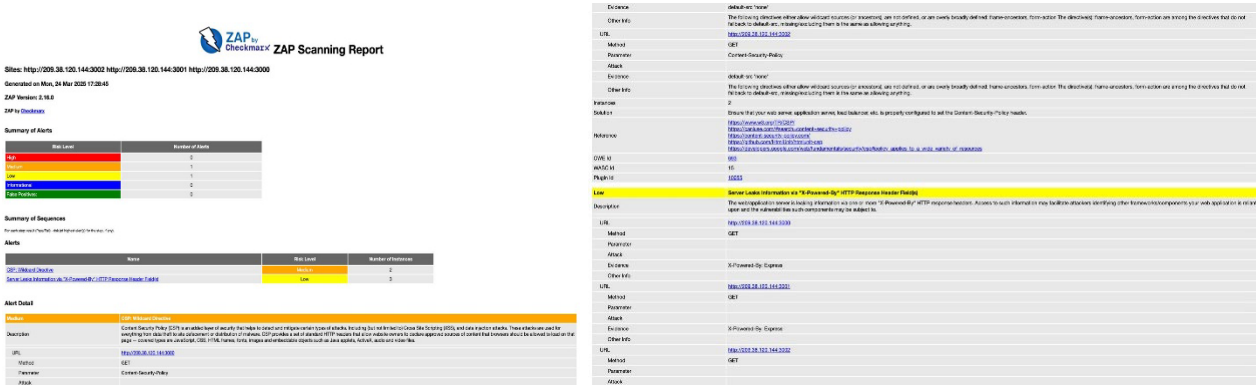

Fig. 7 – OWASP ZAP Report dashboard

6 Continuous Monitoring

In this project, we implemented continuous monitoring for TCH-PIS using Nagios, ensuring real-time tracking of system health and service availability. The main objective was to monitor Docker container status, disk usage, and critical service ports to maintain system reliability and reduce downtime.

As a first step, we defined TCH-PIS as a host in Nagios ensuring that it is monitored 24x7. The host is checked every few seconds, and if it becomes unresponsive, Nagios triggers an alert. We set a maximum of three check attempts before notifying administrators, preventing unnecessary false alarms. Notifications are sent every 30 minutes until the issue is resolved.

6.1 Nagios

Nagios is employed for monitoring the application and infrastructure. Key features include:

Real-Time Monitoring: Nagios provides real-time monitoring of application performance, server health, and network status, allowing for quick identification of issues.

Alerting: The tool sends alerts to the operations team when performance thresholds are breached, enabling proactive incident management.

Reporting: Nagios generates reports on system performance and availability, helping the team analyse trends and make informed decisions.

Following key metrics are monitored using Nagios core in our project.

1) **Docker Container Status Monitoring**
   Since our system runs services inside Docker containers, we implemented a check to monitor container status and resource utilization. Using the check_docker_container command, Nagios continuously verifies if the TCH-PIS container is running. Additionally, we set thresholds for CPU usage. A warning alert is triggered when CPU usage exceeds 80%. A critical alert is raised when it surpasses 90%. This ensures that the container remains operational and does not overload system resources. The check runs every minute, providing real-time failure detection.
2) **Service Port Monitoring**
   Services in our system run on Docker service ports 3000, 3001, and 3002. To ensure these services remain accessible, we implemented a port check. If any of these ports remain unresponsive for five consecutive checks, an alert is generated. This helps in detecting service failures early and ensuring uninterrupted access to web applications and APIs.

3) Disk Usage Monitoring

To avoid storage-related failures, we implemented disk space monitoring for the root (/) partition. If disk usage exceeds 80%, a warning is triggered, and if it reaches 90%, a critical alert is raised. This helps in preventing system crashes due to insufficient storage.

6.2 Azure Monitor

Azure Monitor complements Nagios by providing insights into the performance and health of applications hosted in Azure. Key benefits include:

Application Insights: This feature allows monitoring application performance, user behavior, and exceptions, providing valuable data for improving user experience.

Log Analytics: Azure Monitor collects and analyzes logs from various sources, enabling the team to troubleshoot issues effectively.

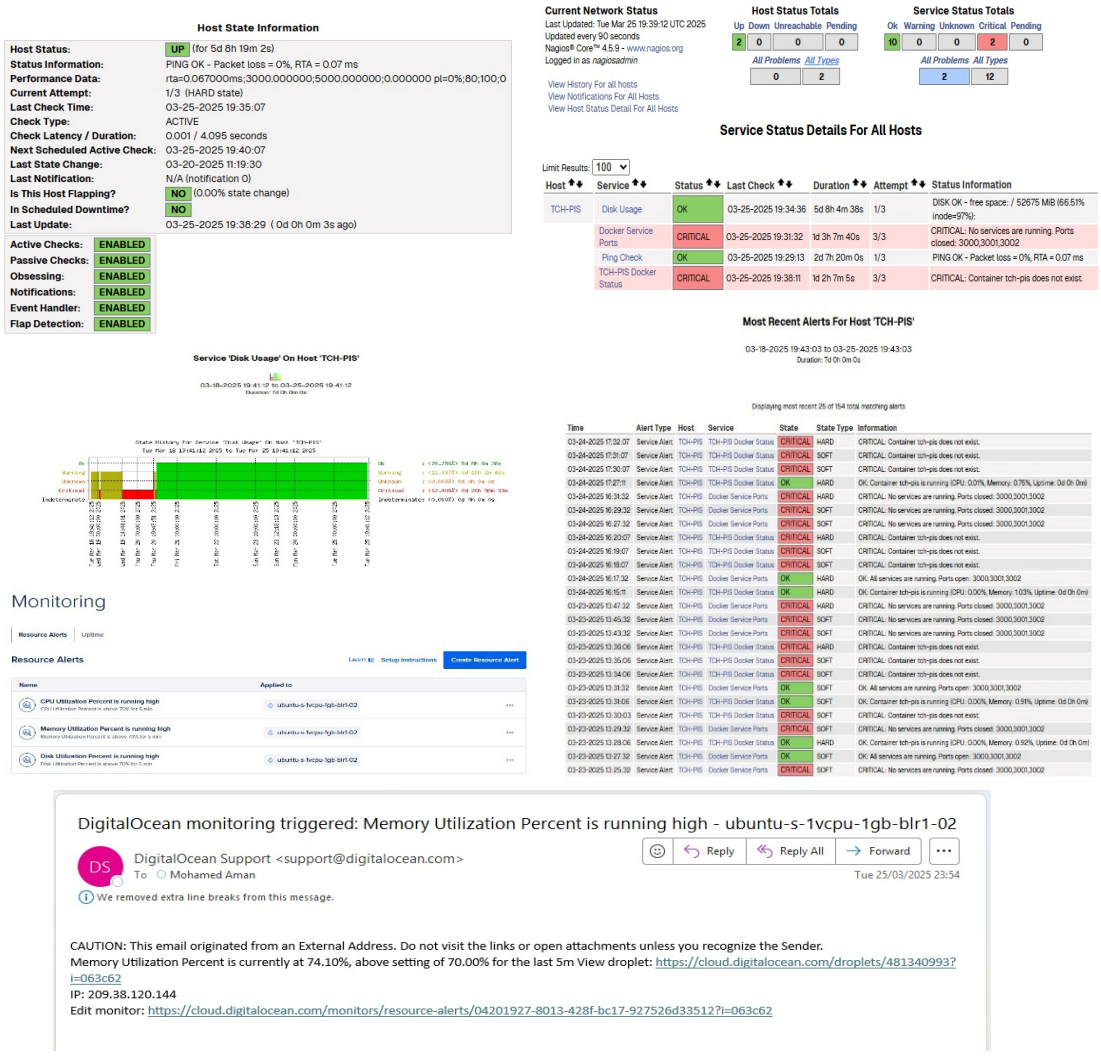

Fig. 8 – Nagios Report dashboard and Alerts

7 Tools and Methodology Support

The tools and methodologies used in the Ops phase support the overall goals of the project by enhancing collaboration, automation, and security.

7.1 Version Control – Git & GitHub

Repository Management: All code is maintained in a single GitHub repository, allowing for easy collaboration and version tracking.

Branching Strategy: Feature branches are used for development, ensuring that the main branch remains stable.

7.2 Agile & Scrum Implementation Using JIRA

Task Management: JIRA is used to manage tasks, sprints, and backlogs, facilitating effective project management.

Sprint Reviews: Regular sprint reviews ensure that the team reflects progress and adjusts plans as necessary.

7.3 Continuous Integration with Jenkins

Automated Builds: Jenkins automates the build process, ensuring that code changes are tested and validated continuously.

Pipeline as Code: The CI/CD pipeline is defined in code, allowing for version control and easy modifications

7.4 Security Testing with OWASP ZAP

Dynamic Testing: OWASP ZAP is integrated into the CI/CD pipeline to perform dynamic security testing, identifying vulnerabilities in real-time.

Automated Scans: Regular automated scans help maintain a secure application by detecting issues before they reach production.

7.5 Monitoring with Nagios and Azure Monitor

Proactive Monitoring: Nagios and Azure Monitor provide real-time insights into application performance and health, enabling quick responses to incidents.

Comprehensive Reporting: Both tools generate reports that help the team analyze system performance and make informed decisions for future improvements.

8 Resource Links

- **GitHub repository**

  https://github.com/faisalmc/TCH-PIS

- **MongoDB**

  URL - mongodb+srv://fmc4000:PcMk4CYYjrNQ9LW8@patient-info-system.qj5ku.mongodb.net/patientdb?retryWrites=true&w=majority

  JWT secret - e5b8a6d8f92c

- **Service APIs – Postman**

  https://f21ao-group.postman.co/workspace/TCH-PIS~f0968d44-b64a-4c37-b89c-4ac3cccd9974/collection/41575727-3be3f492-99d6-46af-ab3d-8d4422def8fc

- **Continuous Testing – Postman**

  https://f21ao-group.postman.co/workspace/TCH-PIS~f0968d44-b64a-4c37-b89c-4ac3cccd9974/collection/41575727-f92d10b6-d533-485f-be82-30a302bdf3d4

- **Docker Image**

  https://hub.docker.com/r/faisalmch/tch-pis-k8s

- **Jenkins**

  URL - http://209.38.120.144:8080/

  Jenkins credential: admin / admin123$Admin

- **SonarQube**

  SONAR-QUBE:

  http://209.38.120.144:9000/projects

  credential: admin

  password: admin123$Admin

- **Nagios**

  URL - http://209.38.120.144/nagios/

  ADMIN: nagiosadmin

  cred: admin123$Admin