End Report

1. Instructions for the Teaching Assistant

You can login to the management console using http://86.50.22.0:8198/login and enter the username and password as below

- **Username:** admin
- **Password:** password
- **Optional Features Implemented:**
    - Data migration between versions
    - Advanced monitoring (Docker API)
    - JWT-based HTTP authentication

2. CI/CD Pipeline Description

- **Pipeline Steps:**

  Build: The application code is compiled to ensure all components are correctly built and ready for deployment. Compilation errors are detected early, preventing faulty builds from progressing further.

  Test: Automated tests verify the functionality, correctness, and stability of the application. Any issues or bugs are identified before packaging the application.

  Package: The application is packaged into Docker containers, creating a consistent and portable runtime environment. These containers can be reliably deployed across different machines or environments.

  Smoke Test: The created Docker containers are started to verify that essential services run correctly. This step helps detect configuration or dependency issues before delivery.

  Deliver: Tested Docker containers are pushed to the Harbor registry. This ensures centralized, versioned storage for easy access and rollback if needed.

Deploy: Docker and Docker Compose are used to deploy the application on the target VM. This final stage ensures the application runs correctly in the intended environment, ready for use.

- **Tools Used:**
    - GitLab CI/CD
    - Docker / Docker Compose
    - Nginx (API Gateway)
    - Harbor registry
    - Docker API

3. Reflections

- **Main Learnings:**

During this project, I gained practical, end-to-end experience with modern DevOps workflows. I learned how to design and implement a CI/CD pipeline, build and manage containerized microservices, and operate a blue-green deployment strategy. I also learned how to integrate a JWT-based authentication system, create a management and monitoring dashboard, and work with reverse proxies through Nginx.

Implementing the monitoring features taught me how to collect and display key metrics such as log sizes, container uptime, CPU and memory utilization, and how advanced monitoring can be extended through the Docker SDK. Overall, the project solidified my understanding of how different DevOps components fit together into a full deployment pipeline.

- **Difficulties Encountered:**

    I encountered several practical challenges during the implementation:

    CI/CD pipeline

    This was the most challenging part

    - Writing GitLab CI stages correctly, handling variables, registry authentication, and fixing YAML errors took time.
    - Deploying via SSH inside CI required debugging permission issues, SSH keys, Harbor login, and ensuring the correct tag/version was deployed.
    - The blue/green deployment logic (deciding active backend based on tag) required careful testing.
    - **Port Conflicts:**

      Running multiple services and containers created conflicts—especially when gateway ports were already allocated by another process. I had to learn how to identify and free ports or reconfigure the Docker Compose files.
    - **Networking & Routing Issues:**

      Configuring Nginx as the gateway required multiple iterations to get the routing correct, including handling separate ports for UI, API, and internal service traffic.
    - **Integration of HTTPS:**

      Preparing the system for HTTPS required understanding DNS requirements, the Certbot flow, and that all certificate work must be done on a publicly reachable server (not on a local machine). This was initially confusing.
    - **Understanding Docker SDK / Monitoring APIs:**

      Collecting metrics per container, computing averages, and integrating Docker API functions took effort, especially making sure the monitoring code did not block the UI.

  These difficulties forced me to debug deeply, read documentation, and test multiple configurations, which ended up strengthening my understanding.

- **Things You Would Do Differently:**

  If I were to redo the project, I would:

    - **Automate more of the workflow:**

Use scripts or CI/CD jobs to automate service deployment, certificate renewal, blue-green switching, and log cleanup to reduce manual steps.

- **Adopt best-practice folder structure:**

  Start with a cleaner microservice directory layout to avoid confusion as the project grew.

- **Test HTTPS and DNS earlier:**

  Set up the domains and certificates earlier to avoid delays at the end of the project.

Overall, the project was a valuable hands-on experience that helped me understand real-world DevOps deployment challenges and strengthened both my debugging skills and my ability to integrate multiple services together.

4. Effort

Estimated Hours Spent:

I spent around 40 hours. I worked for more than a week and spent more than 7 hours each day.

The total effort includes time spent on:

- Designing and developing the application
- Debugging issues in both local and VM environments
- Testing Docker setups on localhost
- Setting up the virtual machine and configuring networking
- Building and refining the full CI/CD pipeline (GitLab → Harbor → VM)
- Implementing monitoring and analyzing resource usage
- Writing documentation and finalizing the project report

The workload reflects both the technical complexity of the system and the iterative troubleshooting required throughout the project.

5. Summary of Optional Features

1. Data Migration:

Purpose:

The goal of this feature is to ensure compatibility of persistent data when moving between different application versions.

Migration Process

To prevent these issues, a data migration step is executed during version switching when required. The migration process reads the existing persistent data, converts it into the format expected by the target version (for example, by normalizing units or restructuring data), and then writes the updated data back to storage.

You can see this when you click on “Switch Version” and the units(minutes/hours) remain same.

2 .**JWT Authentication**

Purpose:

The JWT authentication ensures that only authorized users can access the protected HTTP API endpoints. It enhances security by verifying the identity of the user and preventing unauthorized access.

Process:

After successful login to the management interface, the user can request a token via the

```
/get_token
```

endpoint, which returns a string in the format:

```
JWT token: <unique token>
```

.

All protected HTTP API endpoints require this token to be included in the request header as

```
Authorization: Bearer <unique token>
```

.

If the token is missing, malformed, expired, or invalid, the request is rejected with an HTTP 401 response.

The token includes an expiration time to improve security and prevent unauthorized reuse.

This approach ensures that only authenticated users can access sensitive service endpoints such as

```
/status
```

and

```
/log
```

3. **Advanced Monitoring Using Docker SDK**

Purpose:

The advanced monitoring provides real-time insights into the API and container performance. It tracks important metrics such as the minimum, maximum, and average response times of API calls, CPU and memory usage of individual containers, and the time since each container was last observed as “living.”

Process:

The implementation uses the Docker SDK to collect metrics from running containers. CPU utilization and memory usage are monitored continuously, while response times of API calls are recorded to calculate min, max, and average values. The system also checks the last observed timestamp of each container to determine if it is active or inactive. These metrics are presented in an organized format for easy monitoring of system performance.