DevOps Report
Group F

Nicklas Jeppesen
nicje@itu.dk

Markus Larion Schlögl
mlsc@itu.dk

David Falces Monton
dmon@itu.dk

Gert Luzkov
gelu@itu.dk

Pierre Rouvillain
piro@itu.dk
s223174@student.dtu.dk

May 2024

Contents

List of Figures

Introduction

In this report we present our MiniTwit-project, which was developed during the course *DevOps, Software Evolution and Software Maintenance* at the IT University of Copenhagen.

Our project consists of a software system, namely MiniTwit, and a DevOps system that supports the deployment, operation, development, and maintenance of the software system.

Chapter 1

System Perspective

1.1 Technology Choices

1.1.1 Choice of Web framwork

In our project, we have choosen to work with *.NET*, the Microsoft open source web framework, in conjunction with Postgres, an open source database. Both options are widely used and well-maintained and documented, which were the primary reasons for our choice.

1.1.2 Virtualization Techniques and Deployment Targets

We elected not to use a dedicated virtualization solution, since we are already using Docker, and thus we assessed that achieving further encapsulation of the environment was not a teneable investment.

Using Docker was an easy choice: We avoid problems with incompatible developer machines, and it is also very easy to deploy docker images to our VM's. Furthermore, the choice allowed us to use the same Docker Compose configuration for deploying a Docker Swarm (see section 2.5 on page 20).

Instead of a virtualization tool, we developed and evolved a custom provisioning script, as we found it to be a more understandable configuration.

We were able to get student credits on DigitalOcean, and the infrastructure was cheap enough for us to run our system throughout the course. Furthermore, We found the Digital Ocean Python binding package `pydo`[1] to be the most well-documented, as the offical Digital Ocean API documentation[2], which lead us to write our provisioning script in Python.

[1] See `https://pypi.org/project/pydo/` (accessed on the 20th of May, 2024)

[2] See `https://docs.digitalocean.com/reference/` (accessed on the 20th of May, 2024)

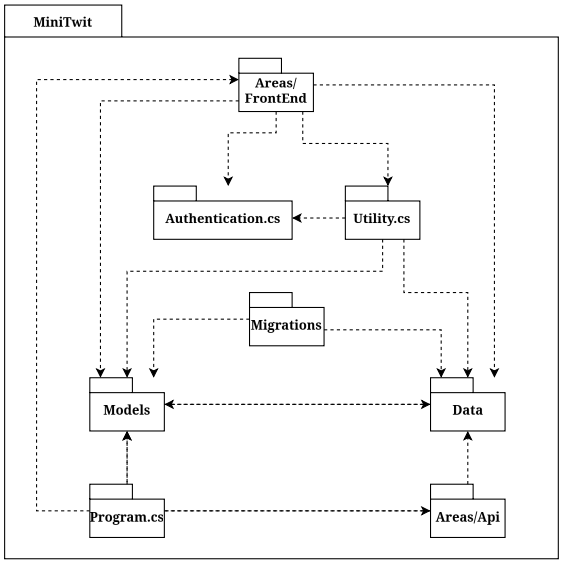

Figure 1.1: Module view of the top-level modules of the MiniTwit application component.

1.1.3 CD/CI reason

Since we use github for storing our source code, we choose to choice github for CD/CI tool, since these two work very well together.

1.2 Design and Architecture

Our final implementation of MiniTwit takes the form of a monolith.However, all the auxilliary components are deployed as seperate services that operate in adjecency the the MiniTwit application component.

1.2.1 MiniTwit Application Component

The monolithic app is delegated into two *areas*:

- `Api`: Provides a RESTFUL API in accordance with the specifications provided by the course.
- `FrontEnd`: Provides a web-interface for the human users.

Both areas use the same object-relational mapping and data abstraction

1.3 Systems dependencies

The list below, is the technolgies we are using in our application, group into categories of technogy.

Project Start - choose framework

- Microsoft AspNetCore v2 as the foundational framework.
- Dotnet API framework: For handling RESTFul APIs
- Swagger: For dotnet API visualization
- Microsoft EntityFrameworkCore version 8.x: Secure and reiable data transactions
- Postgress v.8: Stable data storage
- Docker : For containerized applications
- gravatar: For provide user profile images.
- Github: For storing and deploying source code
- DigitalOcean: Production server for our application

Monitoring

- prometheus/prometheus-net/prometheus-net.AspNetCore: For provide metrics to Grafana.
- Grafana: For Data Visualization of data from prometheus and direct data from Postgress database.

Logging

- Serilog AspNetCore v.8: For logging.
- Fluentd: For collecting logs from different sources.

Software quality - Testing

- Sonarqube: Static analyzer for code quality and securit issues
- CodeClimate: Another static analyzer also for code quality and securit issues
- Playwright: UI testing
- Cypress: End-2-end testing
- Xunit: For Basic Unit testing

1.4 Systems Interactions

In figure 1.2, 1.3, 1.4, and 1.5, we show the flow of different user interactions.

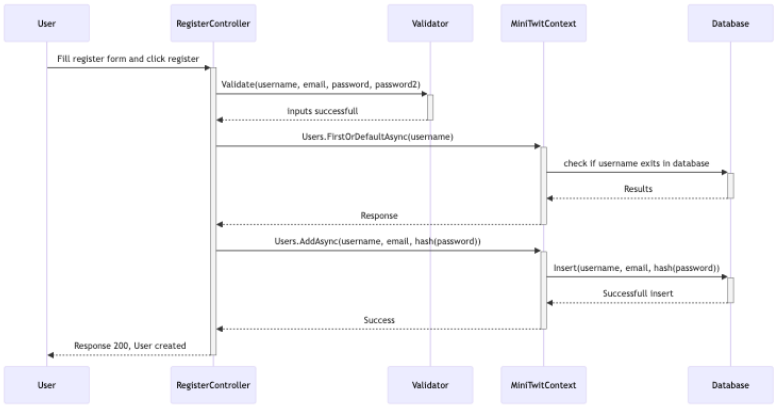

Figure 1.2: Sequence diagram demostrate the flow of registerings of a user

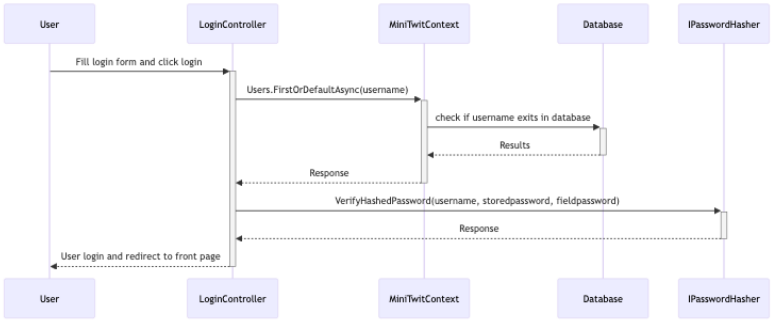

Figure 1.3: Sequence diagram demostrate the login procedure for a user

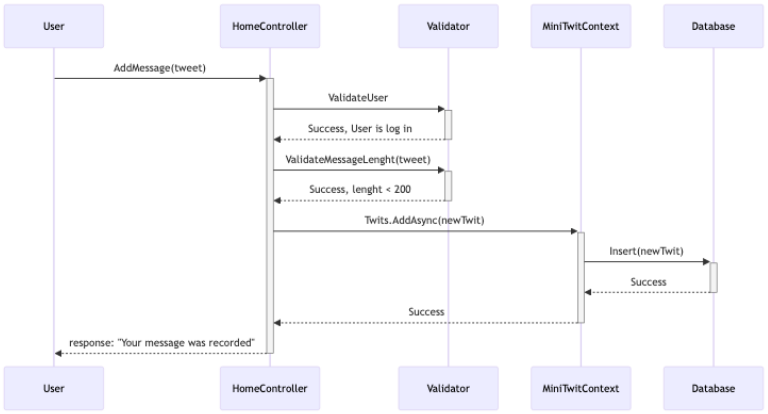

Figure 1.4: Sequence diagram demostrate the procedure for create a new twit

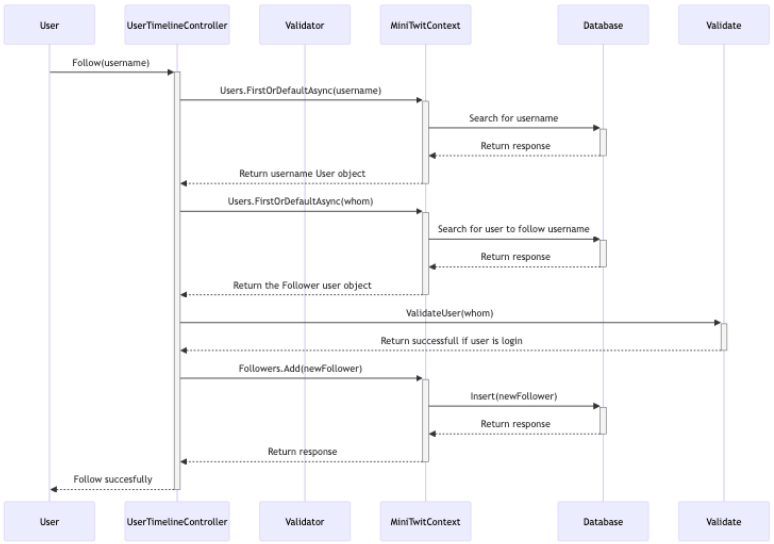

Figure 1.5: Sequence diagram of the procedure for a user follow another user

1.5 Current state of our application

We have use Sonarqube and codeclimate to analyze our code base. This section will describe the current state of our application, acording to the two analyze tools.

Codeclimate

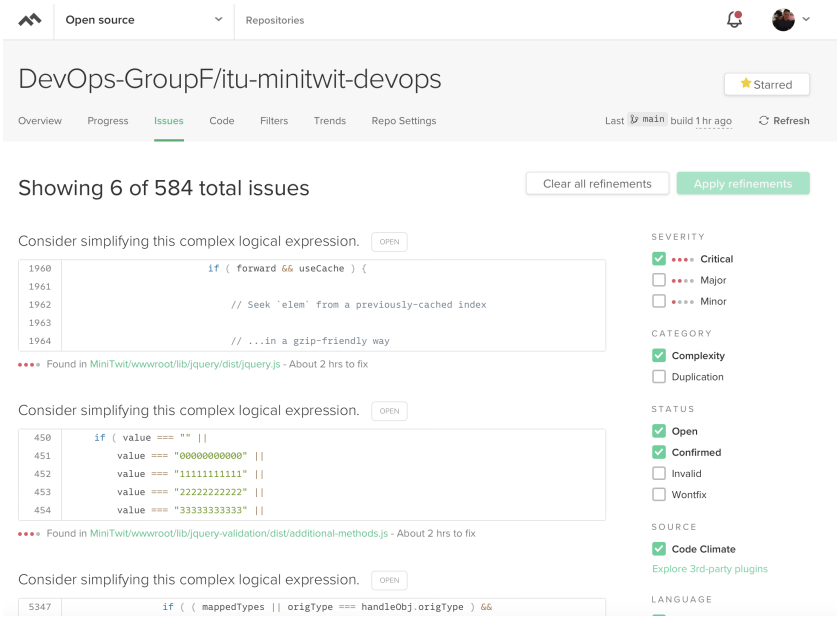

Figure 1.6: Code climate report

Codeclimate general reports 323 code smells, and 261 duplications, but by filtering out included implementations of JavaScript libraries, the real numbers might be reduced significantly. After the filtering, we found 6 critical issues, which were all related to our usage of JavaScript libraries.

Sonarqube

Sunarqube report several issues. Mostly it report security issues, because we are logging to much in our application, which could cause security issues.

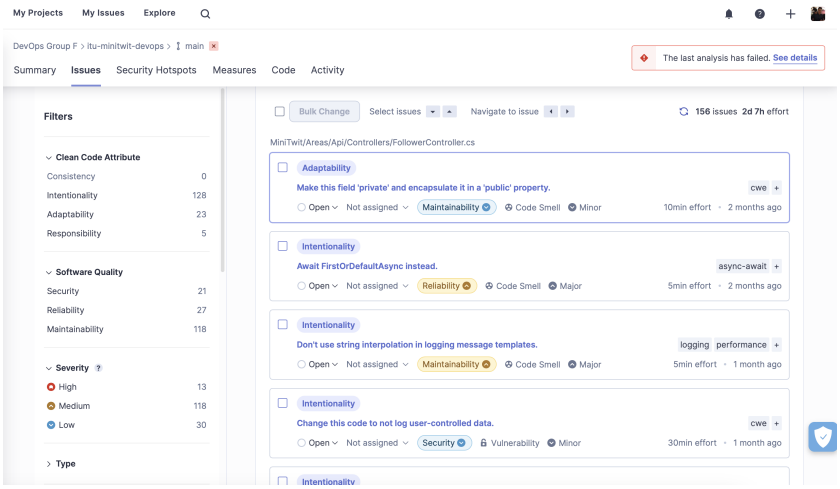

Figure 1.7: Sonarqube report

Chapter 2

Process Perspective

2.1 CI/CD pipelines

To configure and execute our CI/CD pipeline we use GitHub Action, because we were already using GitHub for our repository, and GH Actions provided a seamless integration in our workflow in GitHub.

We have configured two different variants of the pipeline: one that is executed when a PR is opened or updated, and one that is triggered on pushes to main, which, in practice, is only triggered on PR merges to main, since we never push directly to main. The tasks performed by the pipeline are as follows:

1. **Build and test the application**: In this step we build the application, run the unit tests, UI test and end-to-end tests. The unit test are run during the building of the image, while the UI and end-to-end tests are each run by using a specially-crafted compose.yml file, which allows us to run the application, the database, and the UI or E2E tests simultaneously.

2. **Linting**: In this step we execute linters for our code and Dockerfiles, with the use of the third-party GH action "super-linter".

3. **Build and push image**: In this step, a production docker image is build and pushed to our image repository. This image will then be used on the deployment step.

4. **Create release**: In this step, a release is created on the GH repository, with the contents of the commit triggering the pipeline.

5. **Deployment**: In this step, our image is deployed to our server. To do so in a safe way, we have devised a deployment system which is triggered by a simple SSH login:

- In our VM we have an used called deploy. When someone logs in as this used, either by SSH or by using su, a script is executed. When the script finishes, the user is logged out.
- This script connects to GitHub and clones our "server-files" repository, and executes the script deploy.sh, which contains the deployment logic.
- This script then pulls the necessary images, including our image created on a previous step, and deploys them in the server.
- The scripts also use secrets, for example to clone the repository from GH, pull the image, or to configure the app to properly connect to our database.

In this step, we simply log to our server via SSH from GH Actions, which triggers the whole deployment procedure on the server.

We use GitHub Secrets to store and inject the deployment secrets into the deployment script.

In figures 2.1 and 2.2 the flow of the different steps can be seen, both in the case of a PR-triggered pipeline and in the case of a pipeline triggered by a push to main.

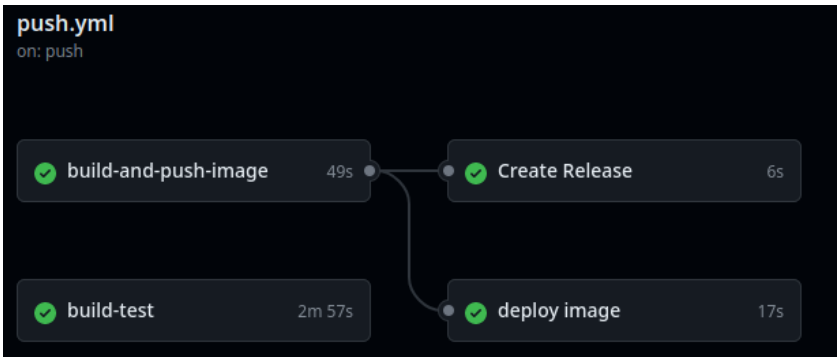

Figure 2.1: Diagram with the different steps executed by the pipeline on a push to main.

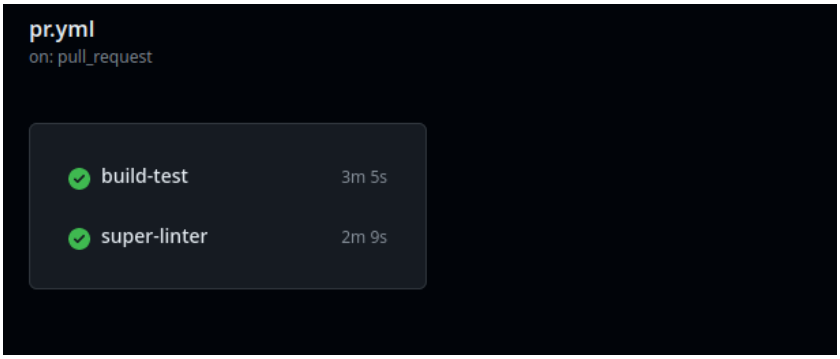

Figure 2.2: Diagram with the different steps executed by the pipeline when a PR is opened or updated.

2.2 Monitoring

2.2.1 Prometheus

Decision Log

We chose Prometheus as it is a is a powerful and widely used open-source tool for systems monitoring. Setting up Prometheus really was as easy as just getting a docker image of it running. All the data source configuring was done through Grafana-

2.2.2 Grafana

Decision Log

Grafana is an open-souce tool, that we decided to use for its ability to visualize all our required metrics.

2.2.3 Implementation of monitoring

A docker container of Prometheus and Grafana is running on the manager node in our docker swarm. Prometheus has been programmed through Grafana to gather and display metrics on all MiniTwit instances.In addition, Grafana gathers data from a managed DigitalOcean database, and displays metrics such as total followers and total tweets.

The following figure describes how monitoring was implemented.

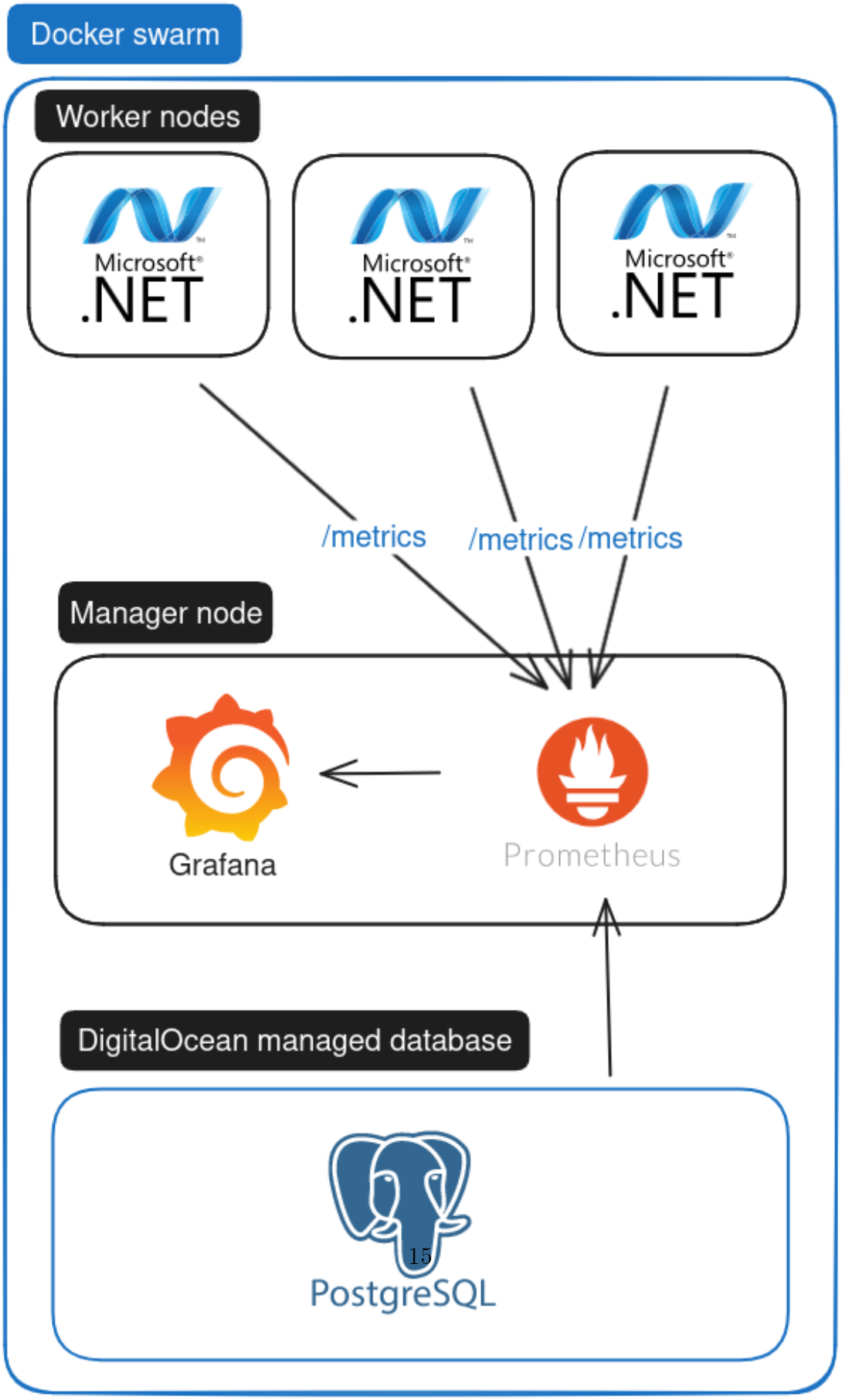

Figure 2.3: Diagram describing our monitoring implementation

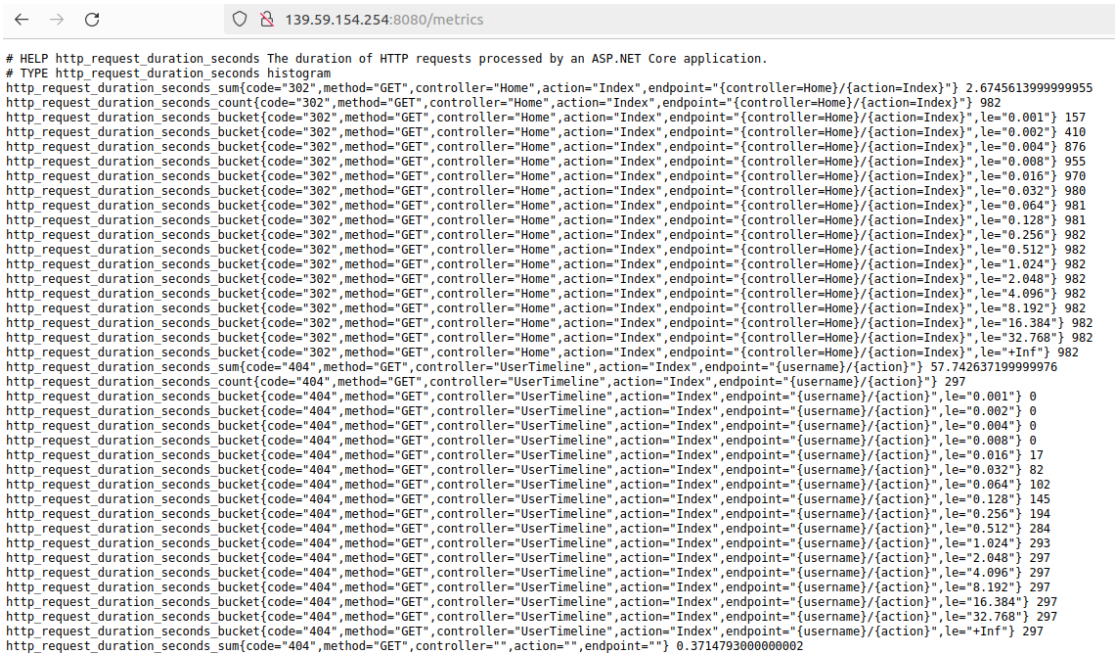

Figure 2.4: Example of a worker nodes metrics page

2.2.4 Monitoring configuration

Grafana and Prometheus are configured in a way, so that when initiating a docker swarm of with a manager nodes running docker images of them, then all necessary configuration gets set up automatically.

Prometheus configuration

Prometheus gets configured using a yml file that gets inserted into the manager nodes Prometheus Docker image as a volume through a compose.yml file.

Grafana configuration

Grafana data sources are set up using a `.yml` file that gets inserted into the manager into the manager nodes Grafana Docker image using volumes. This file also contains variables about how to set up a Postgres database connection and a Prometheus monitoring connection.

Additionally, two dashboards get added to the Grafana Docker image as json files, which contain the configuration of them.

Users are set up using a shell script. After the Grafana Docker image is running, a shell script sends a curl API call to the running Docker image every 5 seconds

until success, in order to add the neccesary users.

2.3 Logging

2.3.1 Fluentd

Decision Log

Fluentd is an open source data collector which processes the data from various sources, making it easier to manage and analyze. The following aspects show why we used Fluentd:

- **Unified Logging Layer:** Provides a unified platform for collecting, transforming, and routing logs from diverse sources.
- **Data Agnosticism:** Seamlessly collects and processes data in a unified format regardless of the data source.
- **Configurable Routing:** Effortlessly configure log routing based on application or service, directing logs to specific destinations for centralized storage and analysis.
- **Flexibility:** Enables us to send data from any source to any destination.

Implementation of Fluentd

A diagram describing our use and implementation of Fluentd can be seen in figure 2.5.

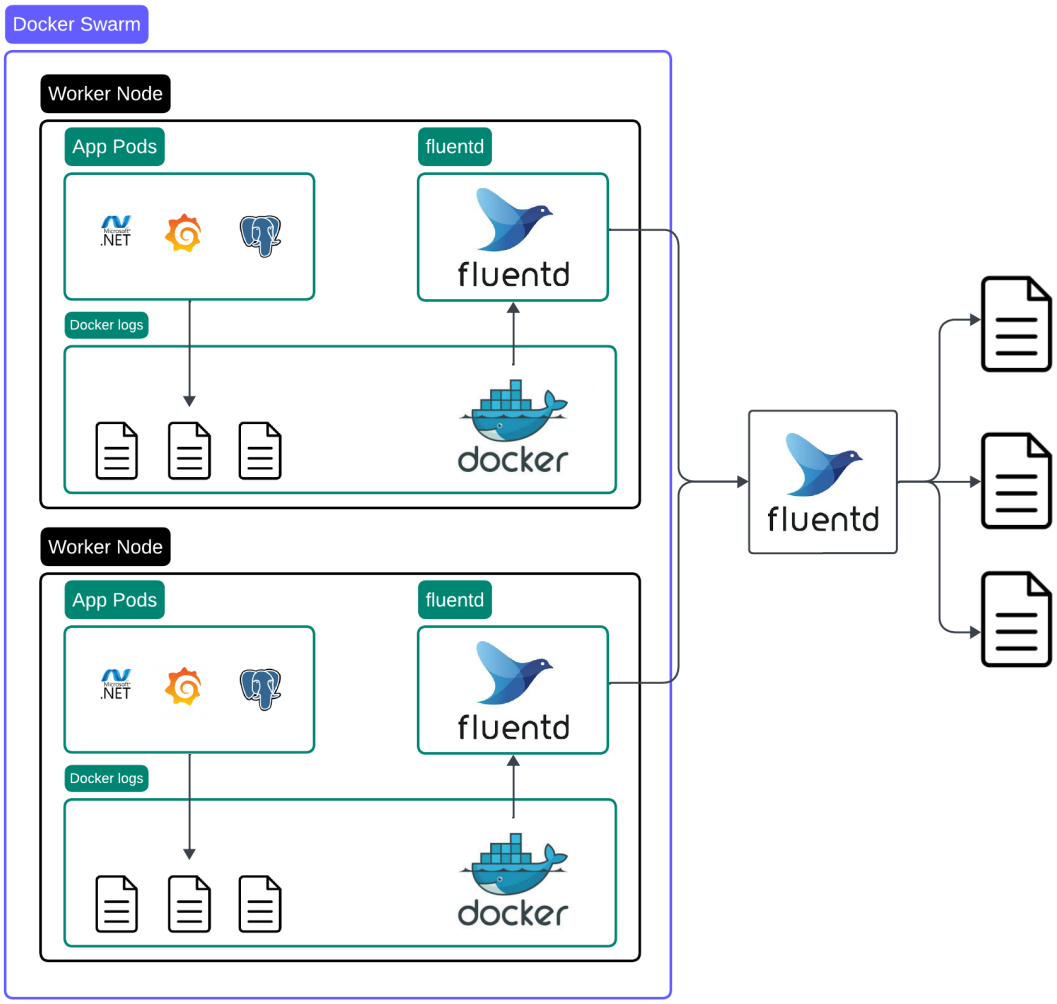

Figure 2.5: Diagram describing our implementation of Fluentd

Fluentd Configuration

Fluentd is configured to listen for incoming log data.Log data from different sources is matched using the match directive.Log data is buffered using the buffer directive to handle data spikes or network outages.

Docker Compose Configuration

A Fluentd container is defined and each service in the Docker Compose file depends on the Fluentd service for logging.

2.4 Security

2.4.1 Risk Identification

Assets

We have different assets that need to be protected:

- The web application.
- The API.
- The database.
- Development, debugging, and deployment tools.

Threat sources

Thread sources can be diverse, and acttack different layers of our application:

- Vulnerabilities in our code: Among the vulnerabilities that we **could** have, these are the most relevant ones:
  - SQL injection.
  - Cross Site Scripting (XSS).

  We need to emphasise that we have checked our code for such vulnerabilities, and we haven't found any.
- Missing HTTPS and firewall.
- Misconfigured firewall.
- Public Prometheus metrics.
- Stolen credentials for all dependencies that require authentication
- Vulnerabilities in the OS or Docker Images used.
- Supply chain attacks.

Risk scenarios

Some possible scenarios of attacks are as follows:

1. SQL injection.
2. XSS on web application.
3. Snooping on user traffic.
4. man-in-the-middle attack.
5. Stealing of SSH keys.
6. Attacker steals Graphana credentials.
7. Attacker connects to */metrics* and can surveil the system.
8. Attacker steals GitHub credentials.
9. Attacker steals DigitalOcean credentials.
10. Attacker steals DB credentials.
11. Attacker uses vulnerabilities of our the OS or Docker images to connect to the DB.

2.4.2 Risk Analysis

Risk Matrix

The risk matrix below helps visualize the possible risks in terms of impact and probability.

	High Impact	**Medium Impact**	**Low Impact**
Low Likelihood	5,8,9,10	-	-
Medium Likelihood	11	1,2	6
High Likelihood	-	3,4	7

Steps to take

In order to minimize our risk, we are going to take the following steps:

- Encrypt our SSH key with a passphrase.
- Hide Graphana behind a firewall.
- Limit the access to */metrics*.

- Use two-factor-authentication in all GitHub and DigitalOcean accounts.
- Continue using best practices when developing, regarding SQL injection and XSS attacks.
- Be aware of the dependencies we are using and their potential vulnerabilities, and keep them updated.

2.4.3 Steps taken

- We encrypted our SSH key with a strong passphrase.
- Most GitHub and DigitalOcean accounts already had 2FA enabled, and we enabled it in all the remaining ones.
- We decided against hiding Grafana behind a firewall since it needs to be access by third parties, namely the professors.

2.5 Strategy for Scaling and Updates

Our strategy for scaling and updates is based on our choice of deploying the app and the auxilliary services through an container orchestration platform, namely *Docker Swarm*, which was chosen as Docker was already installed on our infrastructure.

Currently, our provisioning script creates four virtual machines for a fresh provisioning of our entire project:

- One VM for a a Docker Swarm *manager*-node
- Three VM's as Docker Swarm *worker*-nodes

The three worker nodes run a replica of the app. The manager-node runs all the auxilliary services, and provides load balancing to the three worker nodes.

However, we have not implemented an automatic scaling strategy. For now, the strategy is to monitor the traffic and manually provision more worker nodes if needed.

We have chosen *rolling updates* as our update strategy, since it allows us to add many more worker-nodes, without much consideration for how the updates will perform. Furthermore, when using Docker Swarm, it is trivial to enable rolling updates: The addition of the few lines below in the `deploy`-section of the *Docker Stack* configuration achieves it, with a setup where only one worker is updated or reverted at a time:

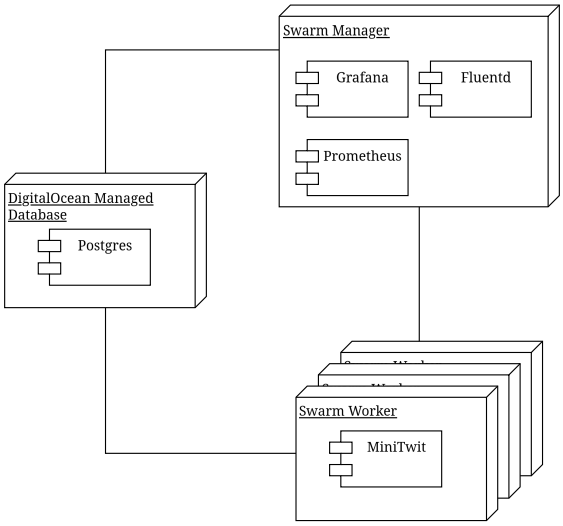

Figure 2.6: Deployment view of our system.

```
update_config:
  parallelism: 1
rollback_config:
  parallelism: 1
```

2.6 AI-Systems used during the project

During the project we have used the AI systems ChatGPT and GitHub Copilot. We have used them mainly to help in debugging, to fix simple errors, i.e. syntax errors, or to perform menial tasks such as translating from one language to another. This is due to the fact that these AI-assistants do not seem to work very well with bigger tasks, as they lack the view of the "big picture", which we experienced multiple cases.

Chapter 3

Lessons Learned

3.0.1 Evolution and Refactoring

We tried to implement elastic search with Kibana, which worked locally, but deploying it crashed the server due to limited memory. We decided to change to Fluentd, which in the end was the better solution. The lesson learned from this is that before implementing a new feature, it is good practice to look at all available technologies and compare them to each other in order to save a lot of time.

3.0.2 Operation

Our significant operative issues were all caused by our reliance on student credits for our infrastructure needs. The first hurdle occurred when we unexexpectedly depleted our credits on Azure, which made our application inoperable.

We decided to migrate to DigitalOcean. Since we were now very wary of the consequences of depleting our credits, we consistently opted for the cheapest resources available to us. This later caused major operational issues, as our DigitalOcean VM's would exhaust their storage capacity with logs

In retrospect, we hypothesize that, as a group, our stance on resource spenditure went from a laissez-faire approach, where we did not consider if resources were depletable, to a polar opposite position, where we were overly careful not to spend resouces. The lesson learned from this is that in DevOps, one should carefully consider how to spend the resources available: Spend too much, and the lights might go out. Spend too little, and one risks suffocating the system unnecesarily.

3.0.3 Maintenance

The first lesson we learned was maintenance of web application requires time and effort, because there can be many new technologies that programmer has to add

or change. Another lesson we learned was there exists several tools to performing the same task and choosing which tool could be a challenge.

Chapter 4

Reflection and conclusion

4.0.1 DevOps approach

One of the requirements was to make a release every week. This made us work more consistently on our tasks than in other projects before, where the working peak was most of the time shortly before the hand-in date. This, we think, simulated the "real working world" way better than the other projects we did before. Also, working in a group of five people, delegating our tasks to a Kanban board, coordinating our work, and automating our deployment processes fulfills the devops approach of this project.