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. The choice of .NET is based on several factors, .Net is free open source maintain by microsoft, which lower the cost and trust that the framwork is highest posible quality, its good for medium to large projects, has large community, using the langauge *C#* which is known to be a relative fast program and also maintain by microsft and last many companies in Denmark using *C#* and .Net so there is a high properbility that the knowloedge we gain is something we can use after University.

1.1.2 Virtualization Techniques and Deployment Targets

We elected not to use a dedicated virtualization solution, as we did not find them to provide enough benefit for the time it would take to setup. Since we are already using Docker, we assessed that achieving further encapsulation of the environment was not a teneable investment.

Using Docker seemed like a very obvious 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 of Dokcer paid dividends later in the project, as it allowed us to use the same Docker Compose configuration for deploying a Docker Swarm (see section 2.5 on page 23).

Instead, we developed and evolved a custom provisioning script, which started as a C# program, but ended up as a Python script. The initial choice

of C# was based on the fact that we were using Azure as our deployment target, and Microsoft provided an Azure library for C# that we found to be effective.

However, we quickly ran out of student credits on our Azure account after we began using the managed database offering on Azure, which resulted in our migration to Digital Ocean. We decided to continue using a custom provisioning script, but for Digital Ocean, we found the Digital Ocean Python binding package `pydo`[1] to be the most well-documented, as the offical Digital Ocean API documentation[2]. Also, the script-like nature of the Python language is more appropiate for the task of provisioning.

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, where all the busineess logic of the system system in production is contained in one component: The `MiniTwit` application. However, all the auxilliary components, such as monitoring, logging etc. are deployed as seperate services that operate in adjecency the the MiniTwit application component.

1.2.1 MiniTwit Application Component

The monolithic app is implemented as a ASP.NET MVC web server. It is delegated into two parts, or *areas*, as they are referred to in the .NET-universe:

- `Api`: Provides a RESTFUL API in accordance with the specifications provided by the course. It consists solely of controllers, that handle various endpoints.
- `FrontEnd`: Provides a web-interface for the human users of our system. It is structured as a classic MVC-webapp, and the views are virtually identical to the UI provided by the Flask-based Minitwit-application that was provided to us in the beginning of the course.

[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)

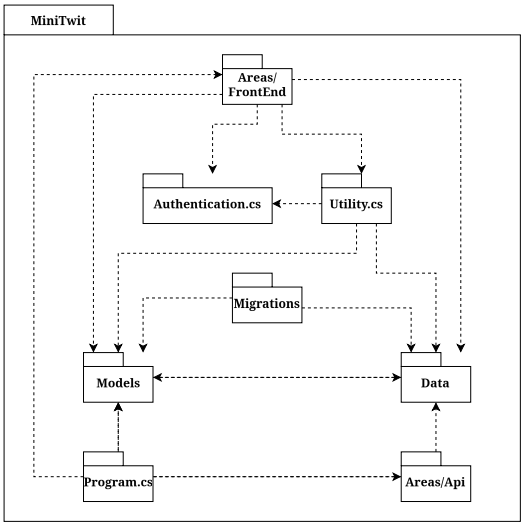

Figure 1.1: Module view of the top-level modules of the MiniTwit application component.

Both areas use the same object-relational mapping and data abstraction. As such, they are simply two different ways to interface with our system.

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: For basis web framework to handle, request, athentication,
- Dotnet API framework: For handling RESTFull API's
- Swagger: For dotnet API visualization
- Microsoft EntityFrameworkCore version 8.x: For general handling dependency injection to a database connection.
- Postgress v.8: Choosen database servere for this project.
- Docker / Docker-compose / docker-swarm: For containerized applications

- gravatar: For provide user profile images.
- Github / Github actions/ Github secrets: 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

This section the interations flow of request for the system, by user create a new request into the system. The request flow is create by providing sequence diagrams, of demostration for 4 important task for the system, namely, register a user, user login, user create a new tweet, and user follow another user.

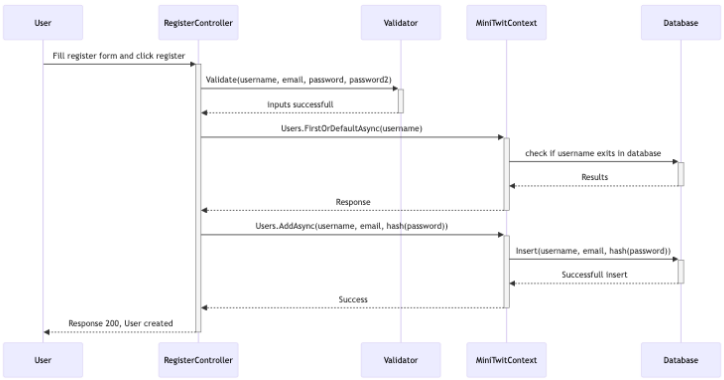

Figure 1.2: Sequence diagram demostrate the flow of registerings of a user

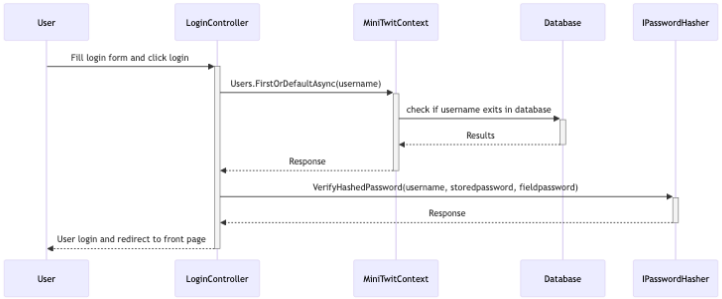

Figure 1.3: Sequence diagram demostrate the login procedure for a user

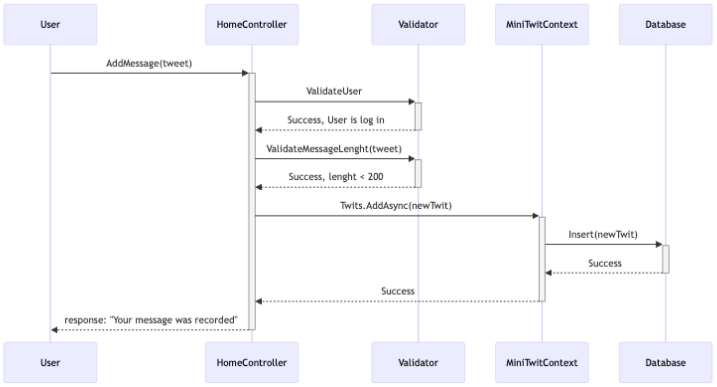

Figure 1.4: Sequence diagram demostrate the procedure for create a new twit

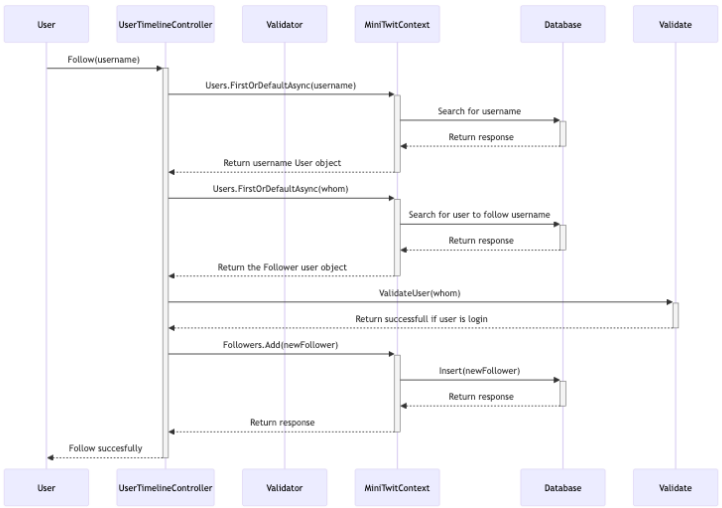

Figure 1.5: Sequence diagram of the procedure for a user follow another user

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

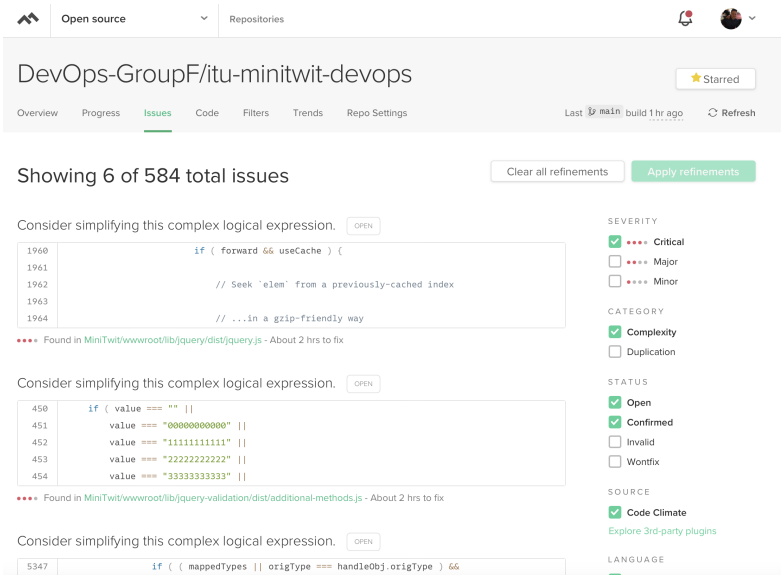

Figure 1.6: Code climate report

Codeclimate general reports 323 code smells, and 261 duplication. Mostly of these is from the javascript lib we are using, for some reason even try to filter code, so only `C#` code is shown, it still include it. Also it was also possible to filter critical issues, here were 6 found, but related to our used third party javascript library jquery. So nothing we can do here.

Sonarqube

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

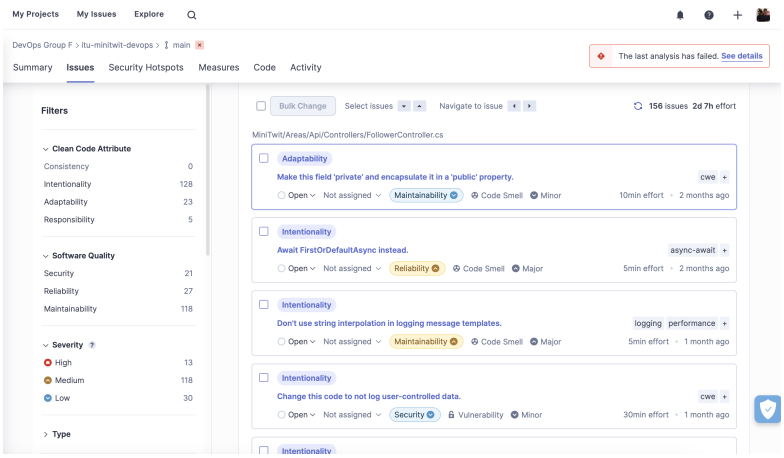

Figure 1.7: Sonarqube report

Chapter 2

Process Perspective

2.1 CI/CD pipelines

To configure and execute our CI/CD pipeline we use GitHub Actions. We chose to use GH Actions because we were already using GitHub for our repository, and GH Actions provided a seamless integration in our workflow in GitHub. Furthermore, it was easy to set up, and the free plan was more than enough for our needs.

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. We then instruct docker compose to use the exit code from the container executing the tests, which makes the pipeline fail if any of the tests has failed. This step is executed both when PR and opened or updated, and when a push is made to main.

2. **Linting**: In this step we execute linters for our code and Dockerfiles, with the use of the third-party GH action "super-linter". If problems are detected, this part of the pipeline reports an error. This step is only executed when a PR is opened or updated.

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. This step is only executed when a push is made to main.

4. **Create release**: In this step, a release is created on the GH repository, with the contents of the commit triggering the pipeline. This step is only executed when a push is made to main.

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. The person logging in cannot execute any other command other than the script, to prevent security issues should someone obtain the keys to the user. 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 script uses other files which are also stored on the same repository, such as a compose.yml, which specifies which images to use, and configuration files for f.e. grafana and prometheus.
    - The scripts also use secrets, f.e to clone the repository from GH, pull the image, or to configure the app to properly connect to our database (which is not located on the VM). These variables are stored on text files on the server. In order to maximize security, these files are owned by root, and are only readable by root or by members of the group deployers (of which deploy is a member), which prevents them from being read by unauthorized users. The bash scripts then access these files and export them as environment variables, which then can be used where they are required.

    In this step, we simply log to our server via SSH from GH Actions, which triggers the whole deployment procedure on the server. This step is only executed when a push is made to main.

During the execution of the pipeline we need to use some secrets, f.e. to SSH into our server. We use GitHub Secrets to store those.

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.

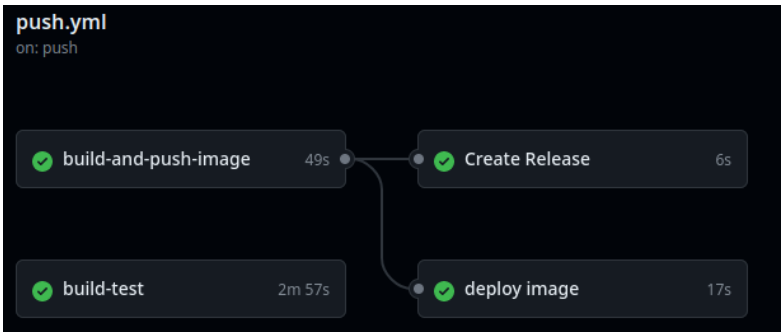

Figure 2.1: Diagram with the different steps executed by the pipeline on a push to main.

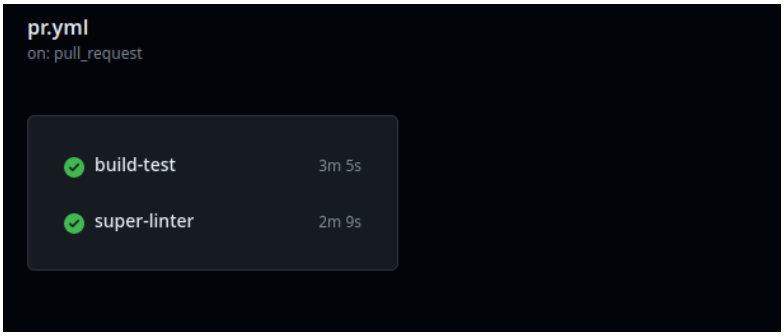

Figure 2.2: Diagram with the different steps executed by the pipeline when a PR is opened or updated.

2.2 Monitoring

2.2.1 Prometheus

Decision Log

Prometheus is a powerful and widely used open-source tool for systems monitoring. We chose this tool for its open-souce nature, ease of use, powerful

query language and simple interoperability with Grafana. Prometheus was used as a docker container running on our VMs and it was used for quering monitoring data about our running .NET minitwit application

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, including all visualization of the data itself. This ease of use was very evident and convenient.

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. In addition, multiple data sources can be easily set up for a single visualization dashboard, making the monitoring of our application easy.

Together, Prometheus and Grafana form a comprehensive monitoring solution that addresses both data collection and visualization needs. Prometheus's robust data collection, combined with Grafana's powerful visualization and dashboarding features, provide a seamless and efficient way to monitor and manage all our systems.

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 mertics on all minitwit instances in all worker nodes and display them on the manager nodes Grafana dashboard. Metrics would be gathered on {url}:8080/metrics and that is where Prometheus can find them. 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.

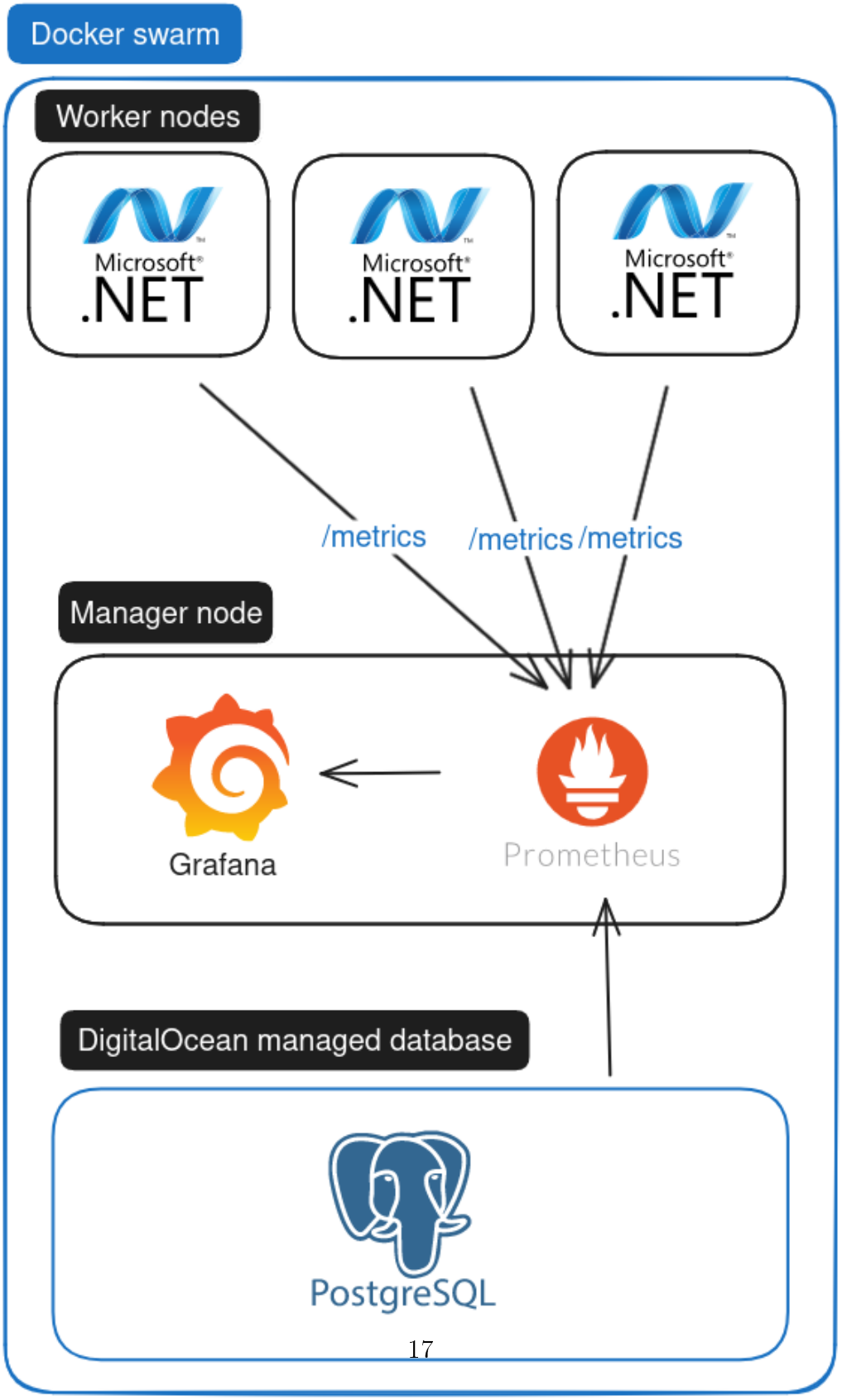

Figure 2.3: Diagram describing our monitoring implementation

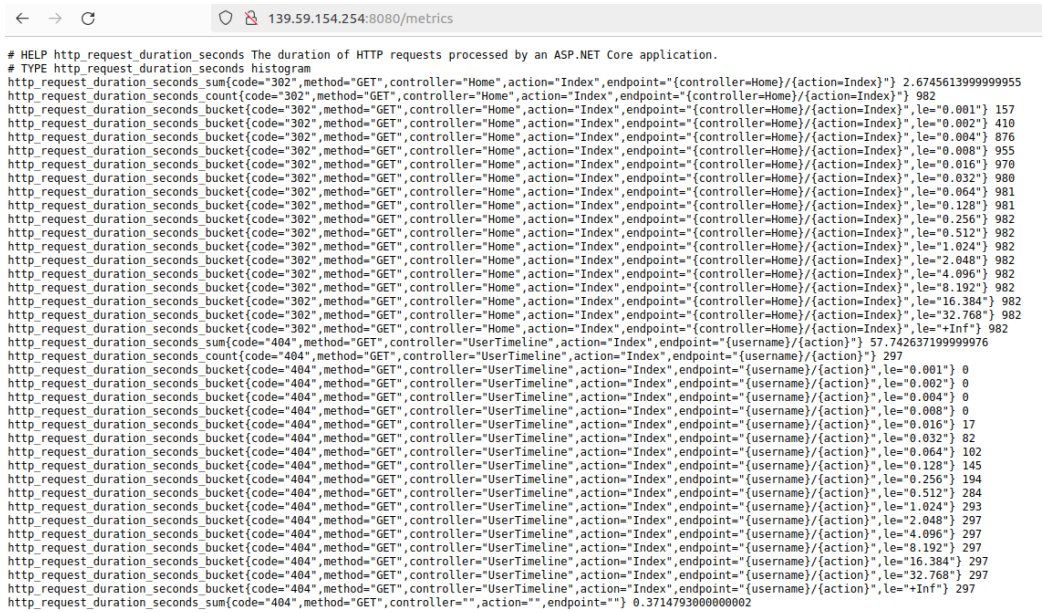

Figure 2.4: Example of a worker nodes metrics page

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. The data source yml file 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. They contain all the variables and setup parameters required to automatically set up the required dashboards.

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. The API call contains the users that are needed to set up automatically.

2.3 Logging

2.3.1 Fluentd

Decision Log

First we tried to implement our logging with kibana and elastic search, which worked on our local machines. The problem with this was that we only had 1GB RAM on our web server, which wasn't enough to run elastic search. To resolve that issue we looked for other technologies like Fluentd.

In order to manage difficulties like logs scattered across multiple sources or disparate log formats we decided to use Fluentd. 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:** Fluentd provides a unified platform for collecting, transforming, and routing logs from diverse sources, simplifying the log management process.
- **Data Agnosticism:** Regardless of the data source Fluentd seamlessly collects and processes data in a unified format.
- **Configurable Routing:** Effortlessly configure log routing based on application or service, directing logs to specific destinations for centralized storage and analysis.
- **Flexibility:** Fluentd's versatility enables us to send data from any source to any destination, empowering us to adapt to evolving logging requirements seamlessly.

Implementation of Fluentd

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

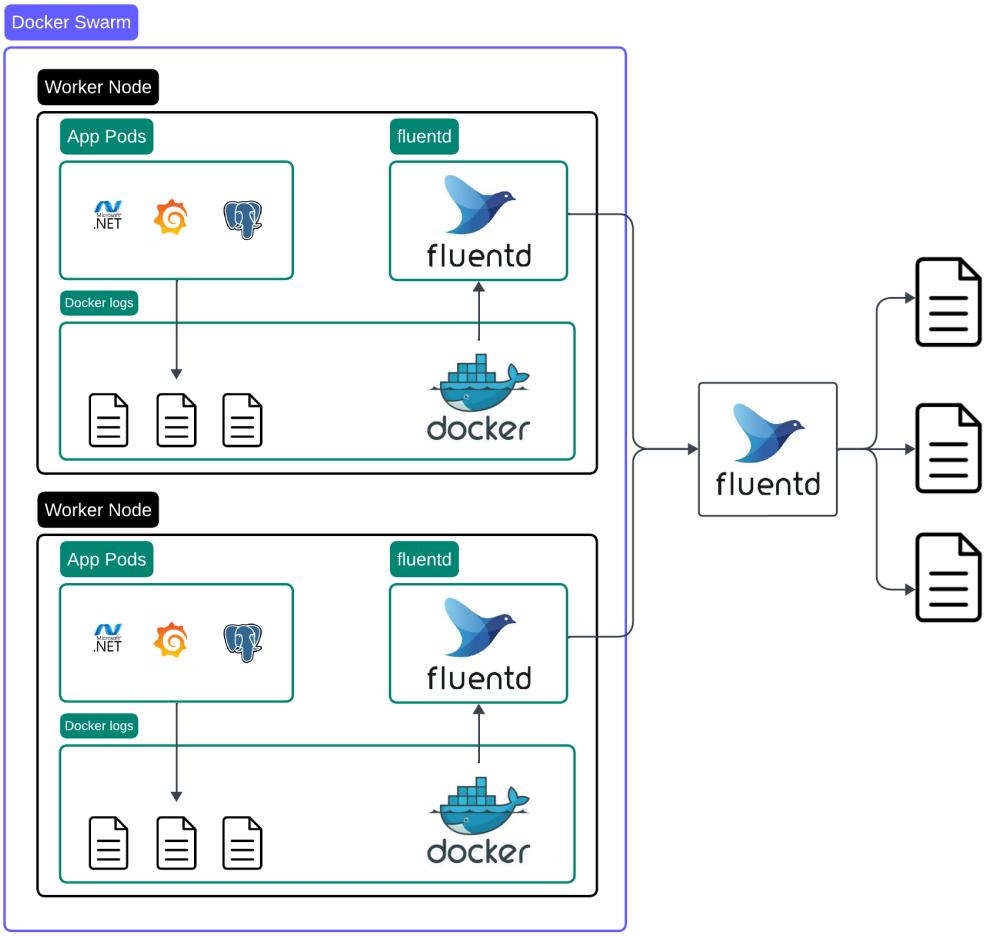

Figure 2.5: Diagram describing our implementation of Fluentd

Fluentd Configuration

Fluentd is configured to listen for incoming log data via the forward input plugin on port 24224. Log data from different sources is matched using the match directive which specifies the destination path and format for the corresponding log data. Log data is buffered using the buffer directive to handle data spikes or network outages, ensuring reliable log collection.

Docker Compose Configuration

A Fluentd container is defined using the fluent/fluentd:v1.12-debian image. Each service in the Docker Compose file depends on the Fluentd service for logging. Logging is configured to use Fluentd as the logging driver, with

specific options to ensure reliable log transmission. Each service specifies a unique tag to route log data to the appropriate destination, which in our case are simple log files.

2.4 Security

2.4.1 Risk Identification

Assets

We have different assets that need to be protected:

- The web application: The web app serves and receives content from users.
- The API: The API allows for software developed by third-parties (f.i. the simulator) to obtain content from our application and sent their own.
- The database: The database stores all user information.
- Development, debugging, and deployment tools:
  - SSH: We use SSH to connect to and configure our Ubuntu Server VM(s).
  - Graphana/Prometheus: We use Graphana and Prometheus to monitor our system.
  - GitHub: We use GitHub to store our code, as well as the deployment logic for our application.
  - GitHub Actions: We use GitHub Actions for our CI/CD pipeline.
  - GitHub Secrets: Secrets needed by the CI/CD pipeline are stored in GitHub Secrets.
  - DigitalOcean: Our VM and database are provided by DigitalOcean and managed via their web interface or API.

Threat sources

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

- Vulnerabilities in our code: Vulnerabilities in our code could allow attackers to access user data or send malware to users. Among the vulnerabilities that we **could** have, these are the most relevant ones:
  - SQL injection: SQL injection would allow attackers to access and modify user data.
  - Cross Site Scripting (XSS): Cross Site Scripting would allow attackers to potentially access and modify user data, as well as to sent malicious information to users.

  We need to emphasise that we have checked our code for such vulnerabilities, and we haven't found any.

- Missing HTTPS and firewall: The use of basic HTTP could result on attackers snooping on users and obtaining password, or on man-in-the-middle attacks.

- Misconfigured firewall: A misconfigured firewall could give attackers direct access to, amongst others, the DB.

- Stolen SSH keys for the VM: Stolen SSH keys for the VM would allow attackers to execute arbitrary code in our VM.

- Stolen credentials for Graphana: Stolen credentials for Graphana would allow attackers to obtain access to the monitoring data of our system.

- Public Prometheus metrics: Metrics from the application for Prometheus to scrap are publicly available at */metrics*. Attackers could use this information to gain information about the operations being executed on the system.

- Stolen credentials for GitHub: Stolen credentials or tokens for GitHub would allow attackers to view, modify, and delete our code, CI/CD pipeline and/or deployment logic, essentially allowing them to execute arbitrary code in our VM.

- Stolen credentials for DigitalOcean: Stolen credentials or API keys for DigitalOcean would allow attackers to delete and/or access our VM and database.

- Vulnerabilities in the OS or Docker Images used: Vulnerabilities in the OS or Docker images used could allow attackers to take control over our system.

- Supply chain attacks: Supply chain attacks could cause vulnerabilities to be introduced in the dependencies we use.

Risk scenarios

Some possible scenarios of attacks are as follows:

1. Attacker performs SQL injection to download sensitive user data.
2. Attacker performs XSS on web application to download sensitive user data or deliver malware.
3. Attacker snoops on user traffic and obtains a user's password, which then uses to impersonate them.
4. Attacker performs a man-in-the-middle attack on a user and provides them with false information.
5. Attacker steals SSH keys, connects to the VM, from there to the DB, and downloads sensitive user data or deliver malware.
6. Attacker steals Graphana credentials, connects to it, and learns more information about the usage of our system.
7. Attacker connects to */metrics* and learns more information about the usage of our system.
8. Attacker steals GitHub credentials and modifies our code or CI/CD to introduce vulnerabilities, or execute arbitrary code.
9. Attacker steals DigitalOcean credentials and downloads sensitive user data from the DB, or use our VM to deliver malware.
10. Attacker steals DB credentials and connects directly to it, due to a firewall misconfiguration.
11. Attacker uses vulnerabilities our the OS or Docker images to connect to the DB and download user data, or to the VM and deliver malware.

2.4.2 Risk Analysis

Risk Matrix

A risk matrix helps visualize the possible risks in terms of impact and probability. We classified the above scenarios according to their likehood

and probability on the following matrix. Note that the numbers on the matrix correspond to the numbers on the list of risk scenarions above.

	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 in order to make it more difficult to steal.
- Hide Graphana behind a firewall.
- Limit the access to */metrics*.
- Use two-factor-authentication in all GitHub and DigitalOcean accounts of the team members.
- 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, which teams members keep separate from the key. 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*. We chose Docker Swarm specifically because installation of Docker was already a part of our automated provisioning procedure, so it would reuire no extra installations. As such, we could readily migrate from our initial Docker Compose-setup to Docker Swarm.

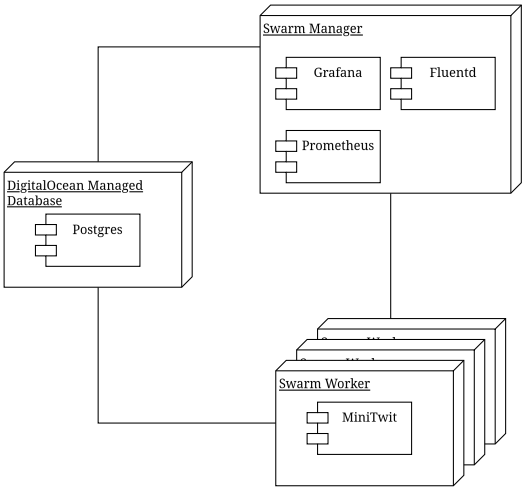

Figure 2.6: Deployment view of our system.

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. This setup works well, because we want to centralize our logging and monitoring, as they are not under variable load from the public internet: It's only the system owners and administrators that need to access those. Furthermore, it is trivial to store the logs on the disk of the single manager-node, which is otherwise a challenge in Docker Swarm environments.

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 addtion of the few lines below in the `deploy`-section of the *Docker Stack* configuraiton achieves it:

```
update_config:
  parallelism: 1
```

```
rollback_config:
  parallelism: 1
```

Here, we specify that at most one worker-node must be under update. This ensures a maximum peformance under a system update. Furthermore, we also specifiy that, in the case of failure of a new update, rollbacks should also only be performed on one node at the time Furthermore, we also specifiy that, in the case of a need to revert a new update, rollbacks should also only be performed on one node at the time. While both these choices have the consequence of out-of-date versions of the system being online for a non-minimal amount of time, we prioritize system performances over the speed of version changes.

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, like f.i. 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". For example they tend to make a first guess and then try to find solutions within the area of the first suggestion, which can lead to losing a lot of time because the fix of the problem lies somewhere else completely. When implementing our E2E tests with Cypress ChatGPT tried to find the solution in the docker compose file, but basically it was our test file naming what caused problem.

As a conclusion, the use of AI-systems can save time and make development efficient, if used in the right places. When those systems do not give the correct answer within the first tries it is good advice to stop using it and try to resolve the issue by yourself.

Chapter 3

Lessons Learned

3.0.1 Evolution and Refactoring

We have followed the recommendation to implement the changes in small steps, which made the whole project more manageable. Nevertheless, we ran into some issues, especially with our logging. We tried to implement elastic search with Kibana, which worked perfectly on our local machines, but deploying it crashed the server due to limited memory. We spent a lot of time on how we could reduce the memory usage so our server keeps up and running before 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 major hurdle occurred when we unexexpectedly depleted our credits on Azure. This was caused directly by adding a managed database instance to our infrastructure, as credits were deducted based on the computational and I/O cost of persisting the high-frequency changes incurred from the simulator.

Since we only discovered this issue once our credits were depleted, our application was inoperable, and we were forced to find a new hosting solution for our project.

While we previously were prevented from creating accounts on DigitalOcean, for some reason we were to create accounts at that later point. Getting deployed on DigitalOcean was trivial, but 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, which was also unexpected by us. While it was easy to prune the logs and redeploy, the surprising nature of the issue did affect our operation. Another issue related to our frugal mentality came in the form of memory constraints, were some of the auxillairy services used so much memory, that the VM ran out of RAM and resorted to swap memory, which made the performance of the MiniTwit applicaiton grind to a halt.

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. A review of our DigitalOcean credits show that we still have a significant left.

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. The right approach seems to conduct an initial analysis of the optimal allocation of the resources available, implement the findings of that analysis, and then consistently monitor the actual usage so that the scale of the infrastructure can be adjusted to the real-life cost, a cost that most likely depends on a dynamic factors such as user traffic.

3.0.3 Maintenance

The first lesson we learned was maintenance of web application requires time and efforts, 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 challenge. Ex. Could be for UI testing, Selenium was the first choice, but after some research we change and choose Playwright for UI testing because is newer ad maintain by Microsoft.

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.