

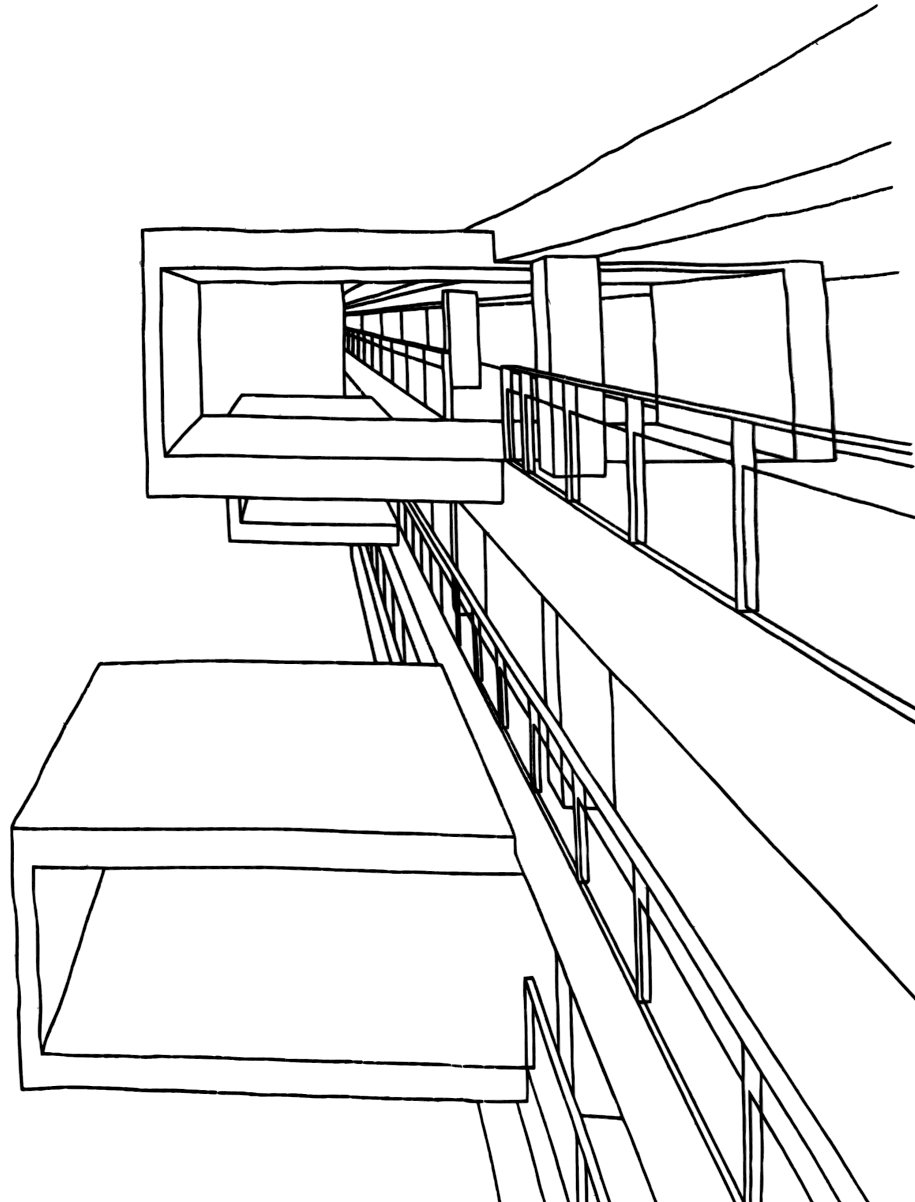
Exam Submission

# DevOps, Software Evolution and Software Maintenance

## Group S

Emiliano Gustavo Giusto (emgi@itu.dk),  
Ingrid Maria Christensen (inch@itu.dk),  
Matěj Basta (bact@itu.dk),  
Melissa Høegh Marcher (mhom@itu.dk),  
Nicholas Gioachini (ngio@itu.dk)

Course code: KSDSESM1KU  
May 2023



# 1 Introduction

The following report describes project work done during the course. The first part will go over the system architecture and current status. The process perspective will look into the group's ways of working, and the development of the system. The last part describes important lessons learned, and what the group hopes to bring to future work in the field of DevOps.

## 2 System Perspective

### 2.1 System Architecture

The following sections will go through various viewpoints to describe the architecture of our system. All views are created in accordance with the UML approach from Christensen et al. [2].

#### 2.1.1 Module Viewpoint

Figure 1 shows a code visualization of our repository. In this view, generated by Amelia Wattenberger's codebase visualizer [12], grey circles represent directories, while filled-in circles represent files, the color corresponding to a file format and its size to the file size. We include this view to give a broad overview of our repository and to give an idea of the size of the different files and directories, which is not possible to show with a package overview diagram. In addition, we point to the fact that the `grafana`, `filebeat`, and `terraform` directories lie outside the main application directory. Note that the `flag_tool` directory and several `.md` files have been omitted from this view.

Figure 2 shows a package overview diagram of the web app. Here we observe the dependencies between source files, the test directories, and the main `app.js` file. If we examine the `src` directory in more detail, as illustrated in Figure 3, it becomes apparent that the `routes` directory relies heavily on various other components of the code, as all the app's endpoints are located within this directory.

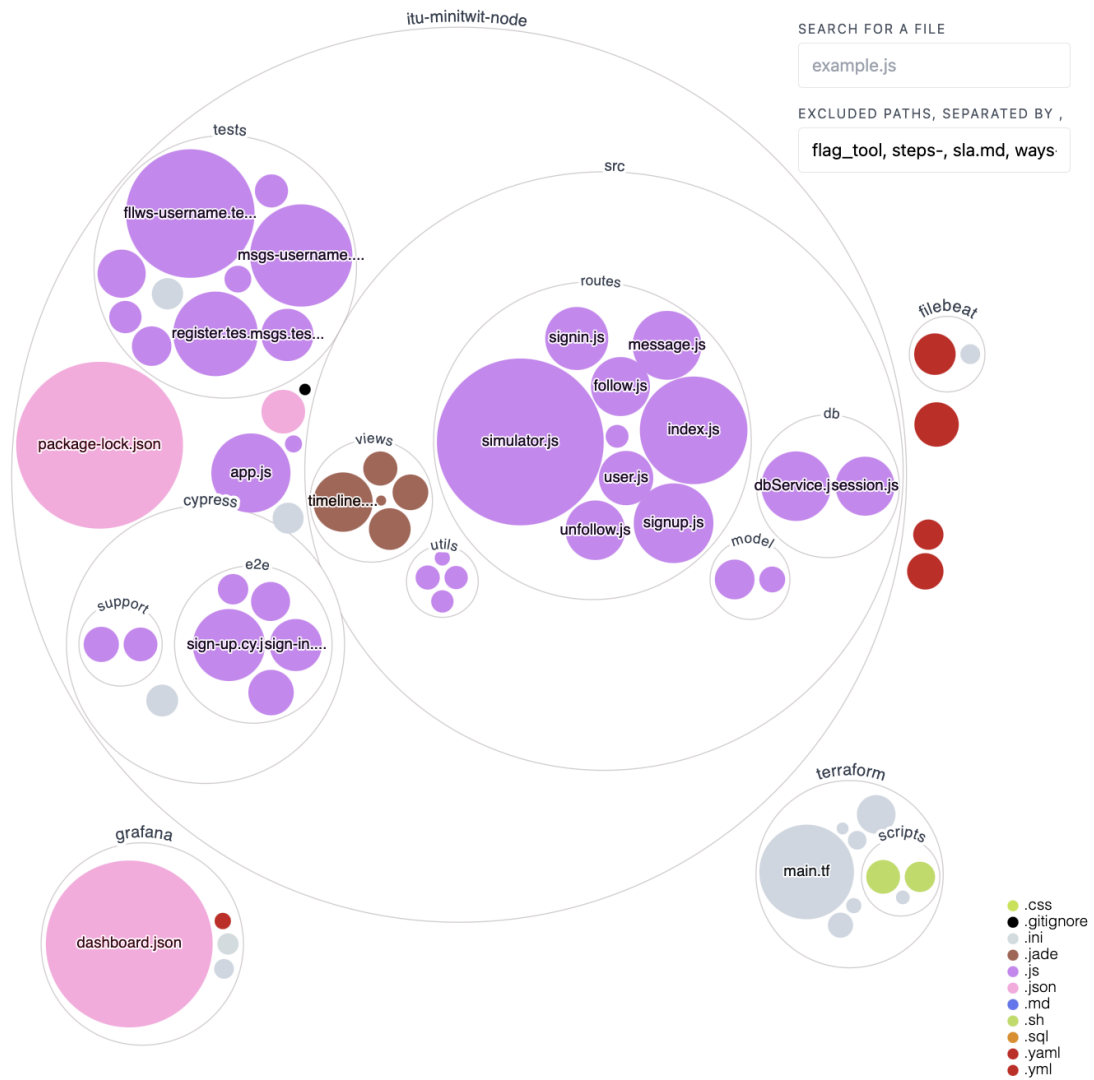


Figure 1: Visualization of our GitHub repository

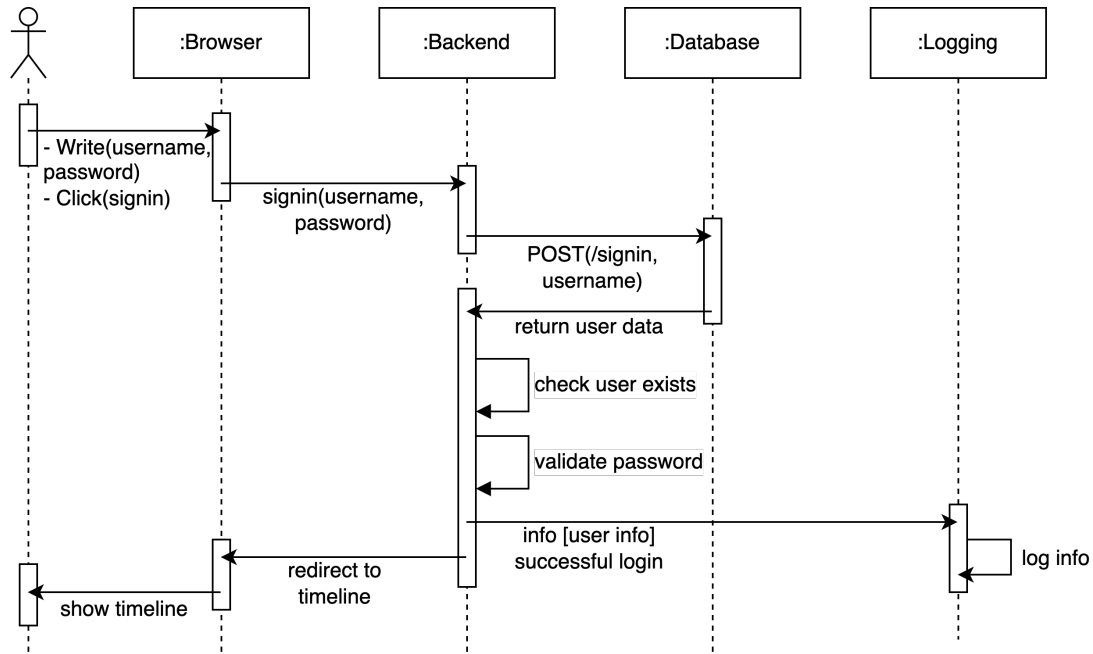


Figure 4: Sequence diagram of a successful login scenario

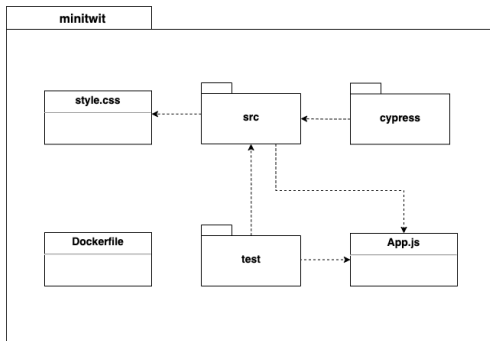


Figure 2: Package overview diagram for the minitwit directory

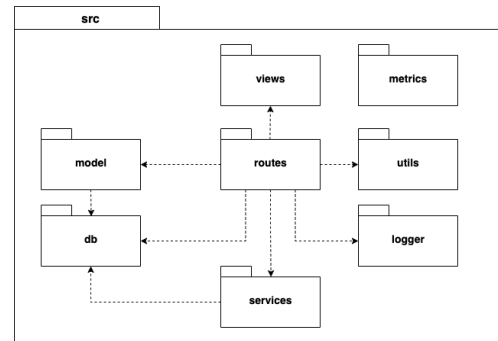


Figure 3: Decomposition of the `src` package

### 2.1.2 Component and Connector Viewpoint

The sequence diagram shown in Figure 4 shows the interaction between subsystems in the scenario where a user successfully logs into the system. Had the username or password been wrong, the error would have been logged instead.

### 2.1.3 Allocation Viewpoint

Figure 5 shows a deployment view of our system. Note that dependencies among software elements have been omitted from this view. This view shows how our software elements are allocated to (virtual) environmental elements at runtime. Software elements in blue can be allocated to any of the environmental elements in red, in accordance with constraints put on our Docker Swarm. The elements in blue are in several layers (ex. the Minitwit component) if specified to be in several replicas.

## 2.2 Arguments for key technology/tool choices

A list of all dependencies in the system can be found in Appendix 5.1.

### Programming language and frameworks

We chose JavaScript, as it is a popular and versatile language for web development that the team wanted to have more experience with. It also allows for a wide selection of front-end frameworks. We used the Pug library, allowing us to get started quickly and efficiently. While we initially considered transitioning to Vue.js or another modern framework for enhanced functionality, time constraints compelled us to prioritize the completion of mandatory assignments.

### Virtualization technique and deployment targets

We chose the containerization virtualization technique with cloud deployment. Containerization was chosen because of the apparent values it brings, such as being portable, ensuring that the application can run consistently in any environment, and being agile, since it is fast and easy to create, deploy, and destroy containers. We chose cloud deployment since none of the group members had the required setup for on-premise hosting and because of the convenience it brings, such as handling scaling and reduced infrastructure management. After shortly considering other opportunities, we selected DigitalOcean as our deployment platform, as suggested in the course. Among the most important benefits of the platform, we found an intuitive user experience and a 200 dollars credit included in accounts for academic purposes.

### DBMS and ORM framework

We opted to use MySQL as our chosen DBMS because it employs a relational data model, organizing data into tables and enforcing a fixed scheme. This structure aligns well with our application's needs, as it has a limited and well-defined set of data types. While PostgreSQL also supports this need and offers comparable features, the decision between MySQL and PostgreSQL was somewhat arbitrary due to their similar capabilities in this regard. As our ORM framework, we used Sequelize as it is widely used with Node.js applications and offers a comprehensive set of features for working with databases. Although

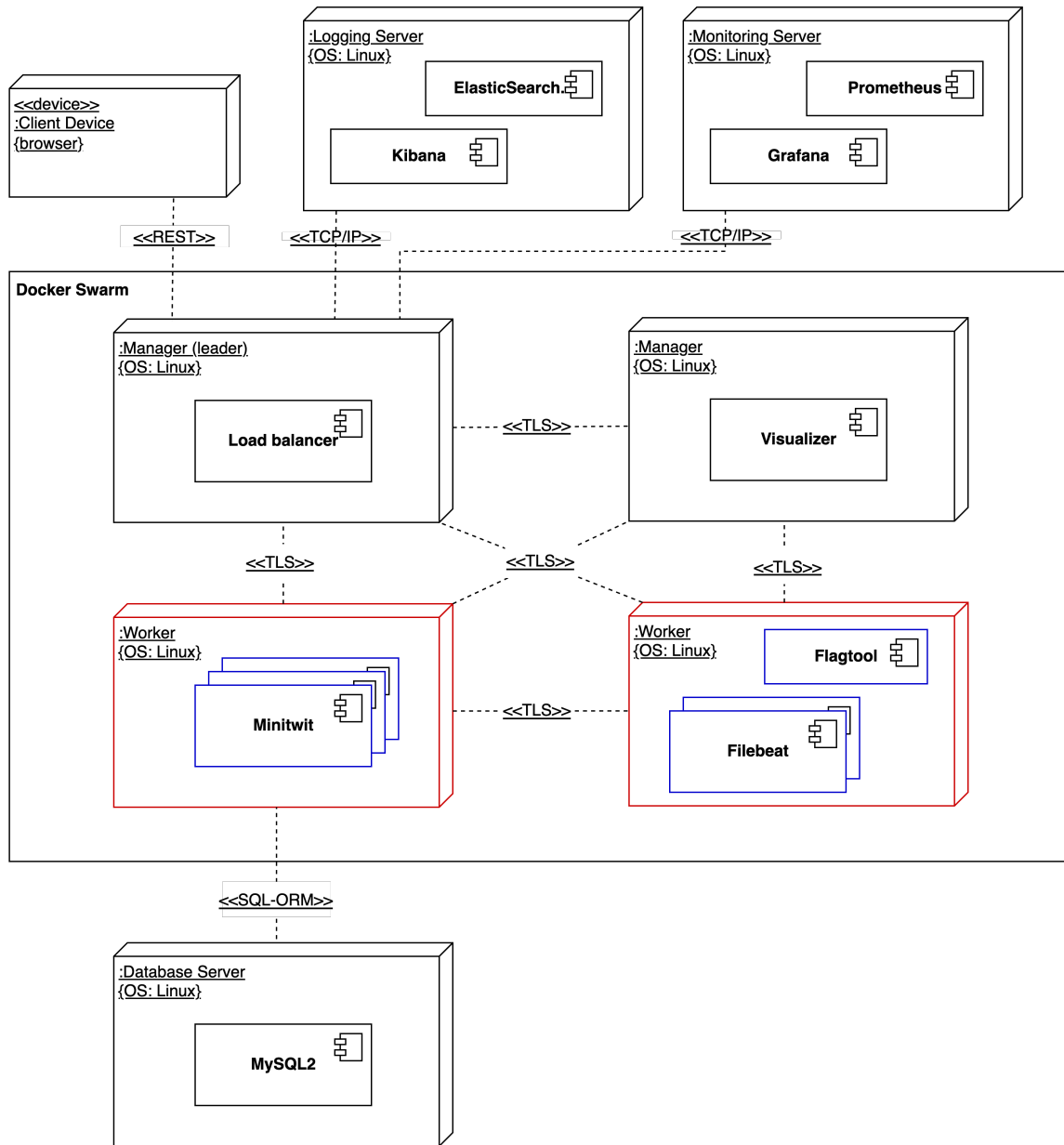


Figure 5: Deployment diagram of the Minitwit system

it was compelling to try a self-hosted database to learn how to deal with infrastructure management, maintenance, backups, and disaster recovery planning, we decided to go with a hosted database option provided by DigitalOcean for its convenience and ease of use.

### 2.3 Current state of the system

All the applications are fully functional. However, the system is currently shut down due to a lack of credits in DigitalOcean. We use SonarCloud as our quality management platform which runs automatic checks of our code after each pushed change. It detects potential vulnerabilities, bugs, code smells, and code duplication. Our codebase contains a significant number of code smells, primarily because our main focus was on addressing bugs and vulnerabilities rather than prioritizing their removal. Below, you can find a dashboard of the latest analysis in Figure 6.

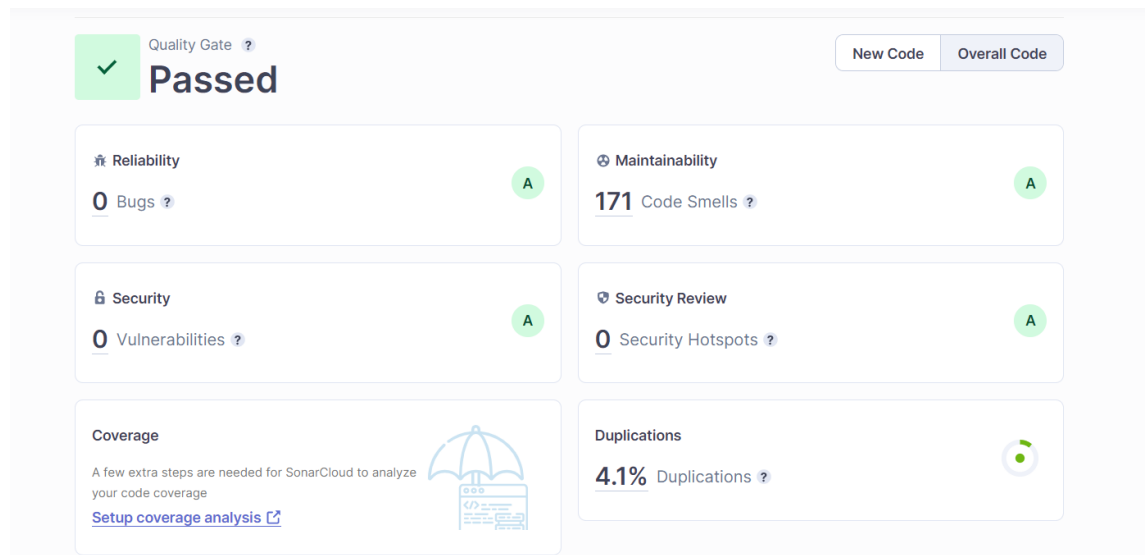


Figure 6: SonarCloud analysis of our repository.

### 2.4 License

We have chosen an MIT license since we using Node.js and npm packages are overwhelmingly using the MIT license or similar licenses [5]. The license is added in our `root` folder.

## 3 Process Perspective

### 3.1 Team work

#### 3.1.1 Team organization

Our project group has embraced a Scrum-like approach, working without a designated leader or specific areas of responsibility. We have integrated various elements from the Scrum framework into our week-to-week project work. We have prioritized meeting physically and working together, and have done so twice a week. During these meetings, we have conducted daily stand-up sessions to discuss progress, align our activities, and coordinate tasks. The remaining time was dedicated to project work, where team members collaborated either in pairs or individually. This flexible approach allowed us to adapt our working arrangements based on the tasks at hand and the preferences of team members.

#### 3.1.2 Collaboration and communication tools

We have used *Microsoft Teams* as the primary channel of day-to-day online communication. In addition, we used *Notion* as a shared space for note-taking and organizing links to relevant resources, see Figure 7.

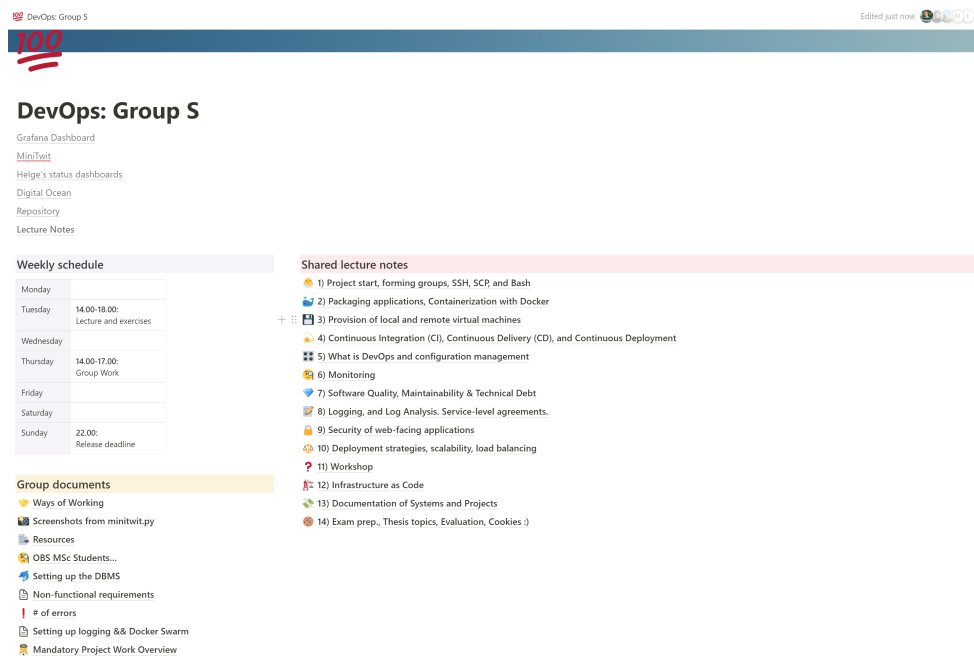


Figure 7: Shared Notion.



In addition to storing our MiniTwit repository, which will be further explained in the next section, we also used *Github* for organizing tasks. Every Tuesday, following the lecture, we divided the tasks for the current week into subtasks on a Kanban board in our GitHub project and prioritized the new tasks in comparison to uncompleted tasks from the previous weeks. The board was consistently updated throughout the project with new task assignments and overall task progress, see Figure 8. We used the Backlog-column for every unprioritized task, and the Todo column for the prioritized task the given week.

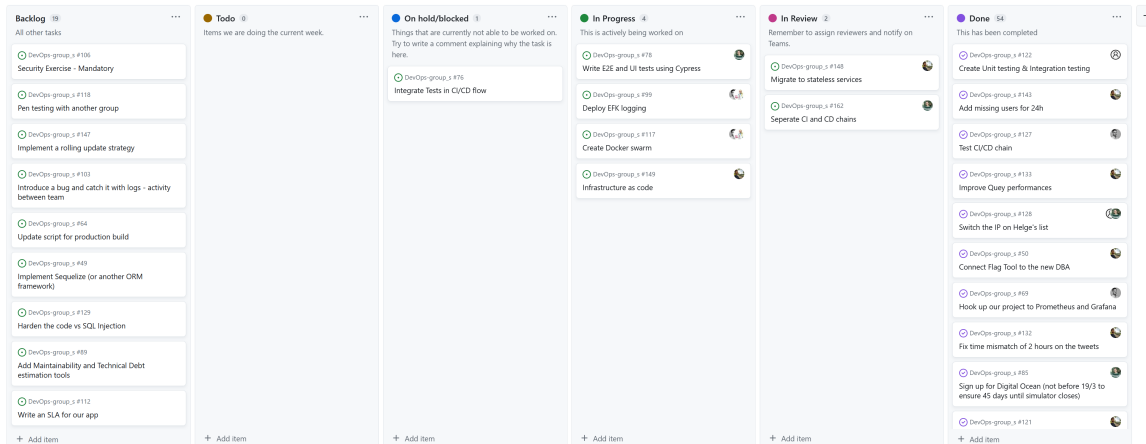


Figure 8: Our Kanban board

### 3.1.3 Branching strategy

Our team has implemented the Github Flow [6] branching strategy, which has one central **main** branch. To add new features, we start by checking out the **main** branch and push working changes to this local feature branch. Once the development work is completed, we create a pull request. Each pull request undergoes checks for merge conflicts, triggers our continuous integration (CI) pipeline, and requires review by at least one other team member.

The primary benefit of this approach lies in its simplicity. However, it also requires thorough checking and testing since it relies on a single central **main** branch. Our experience with this approach has highlighted certain challenges. Specifically, since we only added tests and separated our CI/CD chains relatively late in the process, we encountered issues with having to roll back **main**. This was due to only operating with a continuous deployment configuration which was triggered when changes were pushed to **main**.

## 3.2 CI/CD chains

### 3.2.1 Continuous Integration

Our CI chain, see Figure 9, is triggered when a pull request is created. Note that the cypress testing step is grayed out since it was not thoroughly tested in the CI chain before we suspended our application. The diamond shapes are added to indicate where the chain can stop if a step fails. We intend to test and integrate this properly before the exam.

The CI chain has four main functionalities:

1. Check out code
2. Login to Docker Hub
3. Test the application
4. Build and push images

Due to time prioritization, we were unable to incorporate static checks into our CI chain. However, if we had implemented static checks, such as code analysis and linters, they would have been included before testing our application. By integrating these checks before testing, we avoid running unnecessary testing on an application that would fail either way. The same principle is the reasoning behind having testing before building and pushing our images.

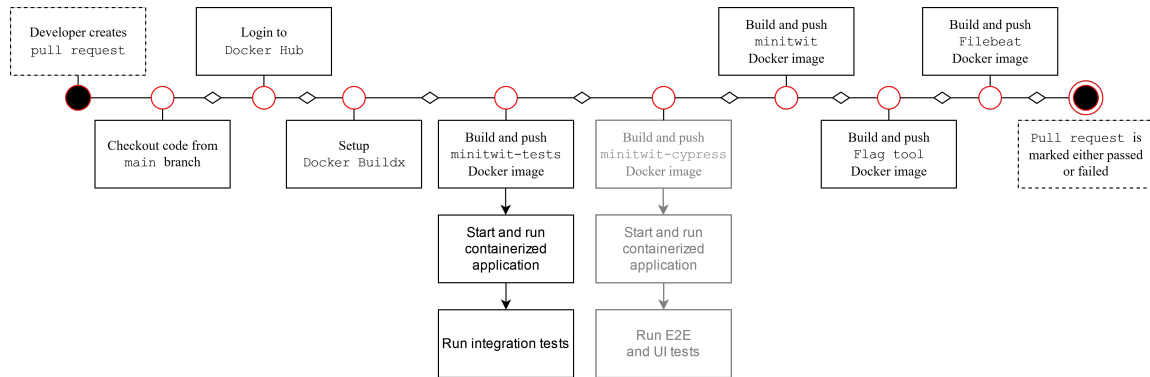


Figure 9: Continuous Integration flow diagram.

In summary, our approach is to complete all necessary steps before deploying the code to our DigitalOcean servers. As part of our CI chain, we have chosen to include the creation and pushing of Docker images instead of placing them in the CD chain. Although these steps often belong in the CD chain, this decision allows us to proactively identify any issues

associated with building or packaging the application into Docker images before merging to the main branch. Having limited experience working with Docker images, this approach favours a thorough understanding of Docker systems and documentation in case any issue arise within the images.

### 3.2.2 Continuous Deployment

Our CD chain, see figure 10, has two responsibilities:

1. Making the SSH connection available as an environment variable for the subsequent step in the deployment process
2. Establishing an SSH connection to the remote server and executing the `deploy.sh` script

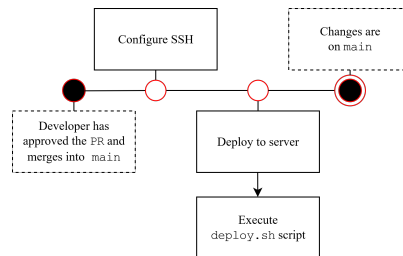



Figure 10: Continuous Deployment flow diagram.

We did not manage to implement continuous delivery in the form of automatic releases or rolling updates in our CD chain.


When we fully implement using Terraform and enforce infrastructure-as-code, our CD chain will be deprecated. Instead of connecting to our CI/CD droplet and executing the deploy script, we will connect to our DO account via their API. Once connected, Terraform will take the lead in provisioning the resources specified in our repository configuration files. This includes creating or removing VMs, networks, IPs, and other defined components.

## 3.3 Repository setup


We chose a mono-repository setup containing the whole project. In the root folder, we have all relevant configuration files as well as sub-folders for every aspect of the system: the application source code, monitoring, and CI/CD pipeline.



Melissamarcher Merge pull request #168 from ingrid-mc/fix-redirect ...



1687e9a 4 hours ago

 336 commits















 .github/workflows	update actions flow with testing	2 days ago
 filebeat	code cleanup	2 weeks ago
 flag_tool	using environmental variables	last month
 grafana	code cleanup	2 weeks ago
 itu-minitwit-node	redirecting to /api and sending status code 200	2 days ago
 prometheus	add monitoring configuration	last month
 remote_files	changes docker-compose removes esTransport	3 weeks ago
 .gitignore	Merge conflicts resolved	3 weeks ago
 Logging-Readme.md	add volumes	3 weeks ago
 docker-compose-logging.yml	adds depend	3 weeks ago
 docker-compose-monitoring.yml	add monitoring configuration	last month
 docker-compose.yml	Merge conflicts resolved	3 weeks ago
 steps-to-mount-monitoring.md	add monitoring configuration	last month
 ways-of-working	Update ways-of-working	3 months ago

Figure 11: root folder in our repository

In the `itu-minitwit-node` folder, see Figure 12, we have our source code, tests, and Docker files. The `src` folder is further split into relevant subsystems such as database and entity handling, frontend in `views`, and routing between pages, see Figure 13 for a full overview.

DevOps-group\_1 / itu-minitwit-node /

Add file

...

melissamarcher

Merge pull request #168 from ingrid-mc/terraform-workflow

a083db6 - yesterday

History

Name	Last commit message	Last commit date
..		
bin	Update www	2 months ago
cypress	Cypress Dockerfile added	4 days ago
public/stylesheets	timeline added and working with backend	3 months ago
src	redirecting to /api and sending status code 200	last week
tests	returning the error code	3 weeks ago
.gitignore	Merge conflicts resolved	3 weeks ago
Dockerfile	changes in port and local docker-compose	last month
Readme.md	modify readme	3 months ago
Vagrantfile	vagrantfile created	3 months ago
app.js	fix header vulnerability	5 days ago
cypress.config.js	cypress init	4 days ago
package-lock.json	code cleanup	3 weeks ago
package.json	cypress init	4 days ago

Figure 12: itu-minitwit-node folder

DevOps-group\_5 / itu-minitwit-node / src / 

Add file

...

ngarpedio redirecting to /api and sending status code 200 4b009f9 - last week 

History

	Last commit message	Last commit date
..		
db	migrates session	3 weeks ago
logger	code cleanup	3 weeks ago
metrics	add one more division to http request received	last month
model	refactor models for testing	last week
routes	redirecting to /api and sending status code 200	last week
services	async latest service	last month
utils	refactor utils	last week
views	add Nodetd to title	last week

Figure 13: src folder

## 3.4 Monitoring and logging

### 3.4.1 Monitoring

In order to monitor our system we implemented a white box pull-based monitoring. Our implementation is rather reactive, as most of our metrics focus on the availability of our services and the infrastructure. However, we also do measure a few metrics, which give us insights into the software quality that we offer to our users.

In order to monitor our application and infrastructure we used Prometheus and Grafana. Prometheus is a monitoring system that scrapes our monitoring metrics every 15 seconds. The infrastructure monitoring metrics include the status of our Minitwit application and database, the uptime of the deployed containers, and memory usage. The application monitoring metrics consist of tracking the total count of HTTP errors and the error counts per individual endpoints, the number of active requests over time, and the response time of each endpoint. To collect those metrics, we instrumented our code with the `prom-client`, a Prometheus client library for Node.js applications. Grafana is a visualization web application, that was used to visualize gathered metrics in organized dashboards.

### 3.4.2 Logging

We chose to implement an EFK stack and use the Winston logging library [13] in combination with an ECSFormat module to log every HTTP request. We sort the messages based on their error level and write them into corresponding files. The log files are then continuously checked by Filebeat, which takes new entries and ships them to Elasticsearch where they are stored. Lastly, we use Kibana to query and visualize the data.

To test that our logging implementation works, we simulated a faulty sign-up request. We proceeded by sending a `/POST request` without providing any password value in the body. An `[ERR_INVALID_ARG_TYPE]` error was logged, indicating that the implementation was working. A postmortem report of the incident has been written and added to the repository. The error can also be seen in Kibana, see Figure 14.

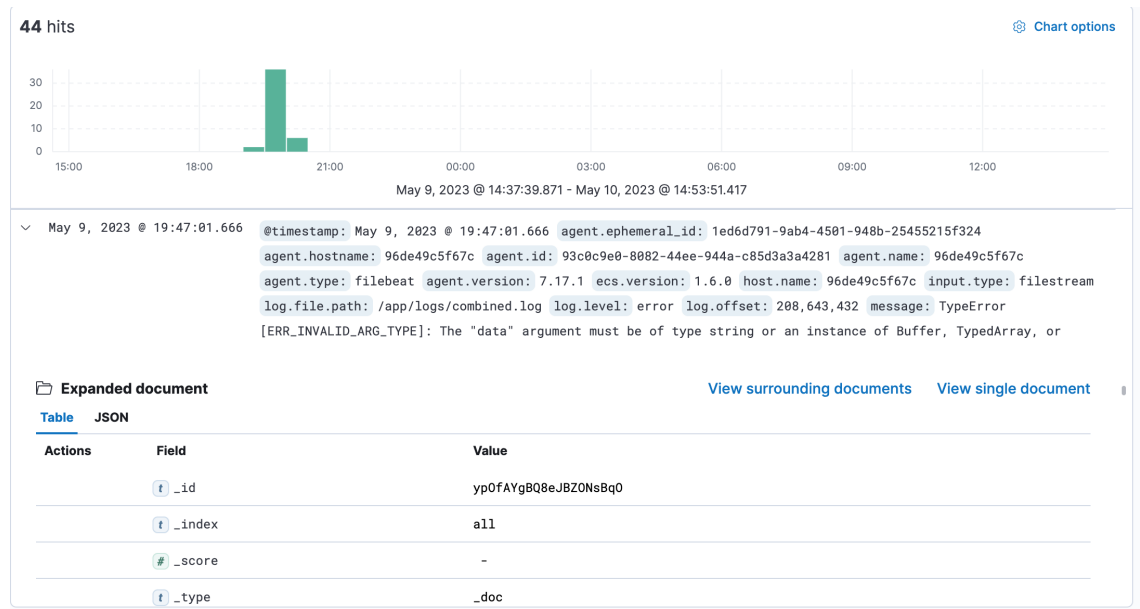


Figure 14: Error display in Kibana

## 3.5 Security assessment

### 3.5.1 Risk identification

The following assets have been identified and considered when making the security assessment:

- User data: We store sensitive user information, such as emails and passwords
- Source code: The entire codebase of MiniTwit
- Production VMs: The servers utilized for running the MiniTwit application
- Logging data: Logs could potentially be misused to exploit the system
- Employees: Employee knowledge regarding the system and access credentials to various platforms

Based on the assets, we have composed seven security breach scenarios. For each scenario, the potential impact has been described and they have been categorized in terms of the three characteristics of security as defined in the CIA triad: Confidentiality, Integrity, and Availability[1].

- |                                                                                                                                                                            |                                                                                                                                                                                                     |
|----------------------------------------------------------------------------------------------------------------------------------------------------------------------------|-----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <p>1. <b>Third-party software contains malicious code or vulnerabilities</b><br/> <i>Impact:</i> Software vulnerabilities, data leaks<br/> <i>Principle:</i> Integrity</p> | <p>5. <b>Man-in-the-middle attack (non-encrypted traffic)</b><br/> <i>Impact:</i> Data leaks<br/> <i>Principle:</i> Integrity</p>                                                                   |
| <p>2. <b>Denial-of-service/distributed-denial-of-service attack</b><br/> <i>Impact:</i> Slowing down or bringing down systems<br/> <i>Principle:</i> Availability</p>      | <p>6. <b>Social engineering, ex. phishing e-mails</b> <i>Impact:</i> Leak of company secrets, Leak of source code<br/> <i>Principle:</i> Integrity</p>                                              |
| <p>3. <b>Brute-force attack</b><br/> <i>Impact:</i> Unauthorized access to user accounts, data leaks, manipulation of data<br/> <i>Principle:</i> Confidentiality</p>      | <p>7. <b>Attackers gain access to production VMs</b><br/> <i>Impact:</i> Unauthorized access, data leaks, bringing down systems<br/> <i>Principle:</i> Integrity, Confidentiality, Availability</p> |
| <p>4. <b>SQL-injections</b><br/> <i>Impact:</i> Altering and/or destruction of data<br/> <i>Principle:</i> Integrity</p>                                                   |                                                                                                                                                                                                     |

### 3.5.2 Risk analysis and mitigation

To determine the severity of each of the threats, we have placed them in a risk matrix to prioritize appropriate mitigation strategies. The severity categories are defined as:

- *Insignificant:* Little to no impact
- *Marginal:* Manageable impact
- *Critical:* Difficult to recover




	Insignificant	Marginal	Critical
Likely		6	
Possible		5	1, 4
Unlikely	3		2, 7
	 Low risk	 Medium risk	 High risk

Figure 15: Risk matrix with ids of threats.

### High risk

1) The impact of a third-party vulnerability can vary broadly. Fortunately, this risk is also easy to mitigate, as static checks such as Snyk [10] are able to identify which third-party libraries should be updated or removed.

4) In the worst case, SQL injections can destroy whole databases. the way we mitigated this risk was to implement an ORM framework, which abstracts away pure SQL from the source code. Further, it is possible to get automatic regular backups on your managed database in DigitalOcean.

6) This type of attack has many forms, is one of the most common, and can be almost impossible to mitigate fully. However, awareness of the potential dangers of phishing e-mails, unknown USB keys, etc. among employees is vital.

### Medium risk

2) A distributed denial-of-service attack (DDoS) can bring down a system. Since it is distributed, it can be hard to detect. DigitalOcean takes some precautions in protecting against reflective DDoS attacks [3]. However, we have not implemented any additional precautions.

5) This risk can be mitigated through the simple use of HTTPS encryption. Due to lack of time, however, this risk has not been mitigated in Minitwit.

7) DigitalOcean allows firewall rules that protect access to virtual machines. However, we have not enforced any additional rules that protect our VMs.

### Low risk

3) Currently, it is possible to guess a user's password through brute force. If we were to mitigate this risk, two-factor authentication could be implemented.



### 3.5.3 Automatic vulnerability test

To test for vulnerabilities in our system, we used the tool OWASP ZAP [14]. It takes a URL as input and scans the web app for vulnerabilities. ZAP found seven vulnerabilities in our system - four marked as "medium" risk, and three marked as "low" risk. The alerts can be seen in Figure 16.

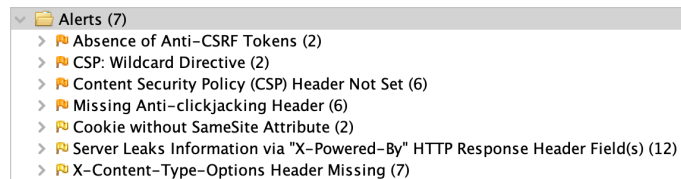


Figure 16: The security alerts identified by OWASP ZAP.

We decided to focus on the alert *"Server Leaks Information via 'X-Powered-By' [...]"*. Here, ZAP tells us that frameworks used by our application will be visible in HTTP response headers. It means that attackers could potentially check for vulnerabilities in these frameworks and use them to their advantage. The 'X-Powered-By' field can easily be omitted by including the line `"app.disable('x-powered-by');"` in our `app.js` file. However, we decided to include the Helmet dependency [7] instead, as it includes an array of response header protection, not just on the 'X-powered-by' field. We were able to see traces of the ZAP scan in our EFK stack, seen in Figure 17, which shows a spike in activity for a few seconds (the scan was done after the simulator was shut down).

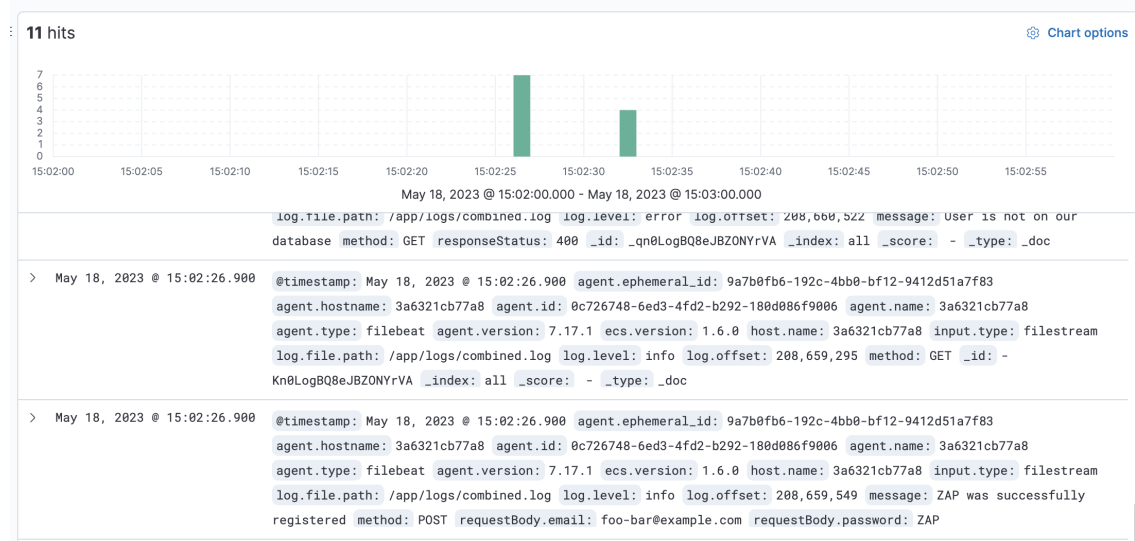


Figure 17: Activity in the logs after running a security scan with OWASP ZAP.

We would have liked to run the OWASP ZAP check once again after including the Helmet dependency. However, we had to shut down the app before running ZAP again to avoid receiving a large bill from DigitalOcean.

## 3.6 Scaling and load balancing strategy

### 3.6.1 Scaling and Docker Swarm

Although DigitalOcean makes it easy to scale applications vertically by increasing droplet size, we decided to scale our system horizontally using Docker Swarm [4]. We decided on Docker Swarm mode as it allows for container orchestration, which is something we wanted to get experience with. In addition, Docker Swarm includes capabilities like self-healing (restarting services when they are down) and built-in load balancing. The cluster is built in code using Terraform[11]. The `main.tf` file sets up one leader (manager) droplet, one manager droplet, and two worker droplets in DigitalOcean. The `minitwit_stack.yml` file sets up services for the cluster. All services are in replicated mode, with three replicas of `minitwit`, two replicas of `filebeat`, and a single `flagtool`. Here, we also add additional services: a Docker Swarm Visualizer and an NGINX load balancer. If we need to scale up the capacity of our application in the future, we can simply rewrite the Terraform file and redeploy it.

### 3.6.2 Load balancing

Adding an NGINX load balancer in our stack file means that we not only have load balancing between nodes but also "in front of" the whole cluster. While the NGINX load balancer distributes task load to an appropriate node, the ingress load balancer in each node makes sure that a request to a specific service will be rerouted to the node running that service. The current system does not use HTTPS for secure client-server communication. However, if we were to implement that in the future, an added bonus of having an extra load balancer is that only traffic to and from that load balancer would need to be secure, as all other communication would run within a closed system.

## 3.7 Use of AI-assistants

We used two different AI assistants during the project: GitHub Copilot and ChatGPT. A few of the team members used GitHub Copilot as a code assistant. This speeds up the coding process, particularly during the initial phase of rewriting the application from Flask to Express.js. ChatGPT was used by all team members and used more as an occasional helper when stuck with a code or a configuration problem. Its effectiveness varied, occasionally leading to confusion. We observed that ChatGPT is more adept at providing general code structures rather than specific configurations. If ChatGPT was used for any sections of the report, we have mentioned it in the relevant section.

## 4 Lessons Learned Perspective

### 4.1 Lack of holistic view of the system

Only very late in the project did we learn the advantage of the first way of DevOps: flow [9]. When the simulator was started, our system was lacking behind in various ways. However, instead of doing little by little and bettering the system as a whole, we did everything we could to adhere to the simulator's API. As a result, we ended up migrating the database and implementing the ORM framework a lot later in the project than intended. Had we simply accepted that the simulator endpoints would not be perfect from the start, the system could have gradually gotten better.

### 4.2 Better refinement of tasks

We maintained an updated Kanban board by transferring tasks from the weekly project work assignments. However, we frequently encountered situations where seemingly straightforward tasks ended up taking more time than anticipated. We believe that this could have been alleviated by collaboratively refining the tasks. By utilizing the collective knowledge of the team, we could have cooperatively written solution keywords that potentially could have optimized our work. Additionally, this approach would have facilitated continuous learning, which is the third way of DevOps[9], as everyone would have gained a better overview and a basic understanding of all tasks, rather than just the ones they were individually responsible for.

### 4.3 Earlier implementation of proper CI/CD pipelines

We have learned how properly implemented CI/CD pipelines significantly improve the flow of development in projects. As mentioned, we did not manage to integrate tests and separate our pipelines until late in the project. This resulted in multiple incidents of pushing faulty code to our `main` branch and it made it difficult to test additions to the pipelines since it would not be triggered until pushing to `main`. Since succeeding in separate pipelines, we have experienced the value of increased branch protection and a shorter feedback loop.

### 4.4 Single owner of our GitHub repository

During the course of the project, our repository was set up with a single owner, which means only a single team member could change code secrets, repository settings, and permissions. Not only did this lead to bottlenecks a few times in the project, but it also made the project feel slightly less agile. Alternatively, we could have created a company GitHub repository with equal permissions between members.

## 4.5 Good practices even with tight deadlines

Towards the end of the project, we ended up slacking on a few of the practices we had originally committed to, for example, our pull request practice. At certain points when we needed to do a lot of changes in a short amount of time, we ended up committing small fixes directly to `main` instead of going through a pull request of a feature branch. Although it did not end up breaking anything, it is bad practice and something we brought back into internal feedback, following the second way of DevOps [9]. Here at the end, we are back to our original branching strategy and pull request practices.

## 4.6 Individual learning on DevOps in general

Evaluating the lessons learned from a higher perspective than simply the project, the DevOps course positively affected our individual learning of what DevOps means to a project or an organization. Working with both theory and practice on IT infrastructure and development has challenged all of us. It truly led to new perspectives and enhanced our problem-solving abilities. Most importantly, it truly helped us “bridge the gap between Development and Operations, emphasizing communication and collaboration” [8].

## References

- [1] Len Bass, Paul Clements, and Rick Kazman. *Software architecture in practice*. Addison-Wesley Professional, 2003.
- [2] Henrik Bærbak Christensen, Aino Corry, and Klaus Marius Hansen. The 3+ 1 approach to software architecture description using uml revision 2.4. *Workingpaper*, Århus Universitetsforlag, 2016.
- [3] Digital Ocean. Ddos.
- [4] Docker Swarm. Swarm mode overview.
- [5] Github. Existing projects and communities.
- [6] GitHub. Github flow.
- [7] Helmet. Get started.
- [8] Ramtin Jabbari, Nauman bin Ali, Kai Petersen, and Binish Tanveer. What is devops? a systematic mapping study on definitions and practices. In *Proceedings of the Scientific Workshop Proceedings of XP2016*, pages 1–11, 2016.
- [9] Gene Kim, Jez Humble, Patrick Debois, John Willis, and Nicole Forsgren. *The DevOps handbook: How to create world-class agility, reliability, & security in technology organizations*. IT Revolution, 2021.
- [10] Snyk. Open source security management.
- [11] Terraform. Automate infrastructure on any cloud.
- [12] Amelia Wattenberger. Visualizing a codebase.
- [13] Winston. winstonjs.
- [14] ZAP. Owasp zed attack proxy (zap).

## 5 Appendix

### 5.1 List of dependencies

The descriptions of the list of dependencies have been partly written by ChatGPT.

#### Frontend

- **jade 1.11.0:** (now known as Pug) template engine for NodeJS

#### Backend

- **cookie-parser 1.4.4:** a middleware for Express that parses HTTP request cookies and makes them available in your application's request object
- **crypto 1.0.1:** a Node.js module that provides cryptographic functionality - used with gravatar
- **dotenv 16.0.3:** a zero-dependency module that allows you to store configuration variables in a file and easily access them in your application
- **express 4.16.1:** web application framework for Node.js that provides a set of features and middleware to build web servers and APIs
  - **express-mysql-session 3.0.0:** a session store for Express applications that uses MySQL to store session data
  - **express-session 1.17.3:** a middleware for session management in Express applications that provides session handling capabilities
- **helmet 7.0.0:** a middleware for securing Express applications by setting various HTTP headers to prevent common security vulnerabilities
- **mysql2 3.3.0:** a MySQL client for Node.js
- **Node.js 18.14.0:** an open-source, server-side runtime environment that allows you to execute JavaScript code outside of a web browser
- **nodemon 2.0.20:** a utility that monitors changes in your Node.js application's files and automatically restarts the application when changes occur
- **os 0.1.2:** a Node.js module that provides operating system-related functionality
- **path 0.12.7:** a Node.js module that provides utilities for working with file paths
- **url 0.11.0:** a Node.js module that provides utilities for URL handling and manipulation

- **winston 3.8.2**: a versatile logging library for Node.js.
- **sequelize 6.31.1**: a JavaScript ORM framework

## Testing

- **chai 4.3.7**: assertion library for JavaScript testing
- **cypress 12.10.0**: JavaScript testing framework for E2E and UI testing
- **debug 2.6.9**: debugging utility for Node.js applications
- **mocha 10.2.0**: a JavaScript testing framework for Node.js applications
- **sinon 15.0.4**: a JavaScript testing library that provides test spies, stubs, and mocks
- **supertest 6.3.3**: library for HTTP request testing

## Other development and collaboration technologies

- **@elastic/ecs-winston-format 1.3.1**: library that provides a log formatter that logs entries according to the Elastic Common Schema (ECS) format
- **npm 9.4.2**: a Node.js package manager
- **prom-client 14.2.0**: a Prometheus client library for Node.js. It provides an API to instrument your applications and expose metrics that can be scraped by Prometheus for monitoring and alerting
- **docker**: A platform for containerization that allows developers to package and distribute applications with their dependencies.
- **docker-compose**: A tool for defining and running multi-container Docker applications, simplifying the management of complex, interconnected services.
- **git 2.40.1**: A distributed version control system that enables developers to track and manage changes to source code efficiently.
- **grafana 8.1.5**: A data visualization and monitoring tool that helps users analyze and display metrics from various data sources.
- **kibana**: A web interface that allows users to explore, visualize, and analyze data stored in Elasticsearch using charts, graphs, and dashboards.
- **filebeat 7.17.1**: A lightweight log shipper that forwards log files from various sources to Logstash or Elasticsearch for centralized log management.

- **terraform**: An infrastructure-as-code tool that enables users to define and provision infrastructure resources across various cloud providers and services.
- **snyk**: A security tool that identifies vulnerabilities and provides remediation guidance for open-source libraries and container images.
- **sonarcloud**: A cloud-based code quality and security platform that analyzes source code for bugs, vulnerabilities, and code smells.
- **owasp-zap 2.12.0**: An open-source web application security scanner that helps identify security vulnerabilities during the development and testing phases.