IT UNIVERSITY IN COPENHAGEN

DEVOPS 2023

# DevOps dudes: MiniTwit

Nikolaj Sørensen (nikso@itu.dk) Daniel Kjellberg (dakj@itu.dk)
Jeppe Lindhard (jepli@itu.dk) Johan Flensmark (jokf@itu.dk)
Benjamin Thygesen (beth@itu.dk)

**Course Code:** KSDSESM1KU
**Course Manager:** Helge Pfeiffer

24. May 2023

# Contents

# 1 Introduction

This report documents the refactoring and maintenance of the application named *MiniTwit* which is made by the group named *dudes* for the DevOps course at IT-University of Copenhagen. The project source is located on GitHub at github.com/Lindharden/DevOps, and the application is running at 157.230.76.157:8080.

In section 2, we describe the system and architecture of the MiniTwit application. Here we go through the dependencies of the application and how they interact with each other. In section 3, we describe the process of maintaining MiniTwit with DevOps technologies. This includes the team organization, and the setups that we used in order to do DevOps. Finally, in section 4, we describe the issues we ran into underway, and the main lessons we have learned.

# 2 System's Perspective

This section documents the architecture of the MiniTwit application. First, the individual dependencies of the application are presented and the rationale behind them. Next, we describe the current state of the system.

## 2.1 Architecture & Design

The MiniTwit application relies on several dependencies, each interacting with each other in many ways. Figure 1 provides an overview of these interactions. In figure 1, it can be seen that a pipeline of actions is initiated every time a commit is pushed to the MiniTwit GitHub repository. When a commit is pushed, the GitHub Actions initializes our CI/CD setup, which is described later in section 3.3. This setup deploys the image of the application and all its dependencies, which are described in section 2.2, to the Docker Hub. From the Docker Hub, everything is deployed to DigitalOcean which hosts our application. On DigitalOcean we use a replication setup, described in section 3.6, which ensures availability of the application.

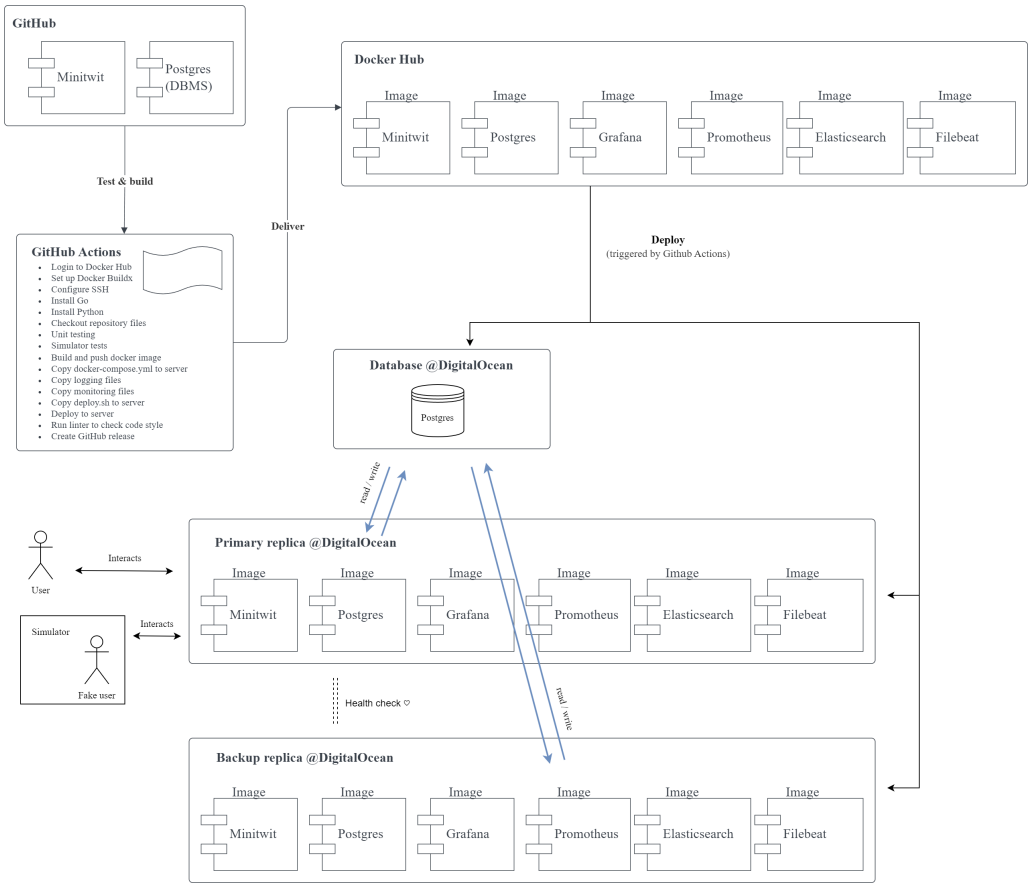


**Figure 1:** Architecture of the MiniTwit application. The diagram shows which dependencies are involved in the application, and how they interact with each other.

## 2.2 Dependencies

This section describes the different dependencies used in our application, and we argue for the choice of each technology.

**Docker** We utilize Docker[1] to containerize all our dependencies, making it easier to isolate and deploy our application. Without containerization, managing the various tools and their dependencies would be difficult, requiring multiple steps for each application run. Docker simplifies this process by packaging everything together and enabling us to execute the application with a single command. We chose Docker over alternatives like Kubernetes due to our prior experience with it and its ability to fulfill all our application requirements.

**Go & Gin** For our web-application, we opted to use Gin[2] as our web framework. Gin is a fast and stable web framework written for Go/Golang. We selected Gin due to its superior performance compared to other frameworks, including Flask, which was the original

[1] docker.com
[2] gin-gonic.com

framework used in the MiniTwit application. We specifically required a web framework that supports HTML rendering and relational databases, as we wanted to be maximize reuse of the original MiniTwit application. Gin fulfills these requirements while also offering scalability and type safety, which are useful for writing maintainable code. Moreover, this project provided us with an ideal opportunity to learn Go and utilize Gin.

**Gorm** In Go programming, one of the most popular ORM (Object–relational mapping) libraries is Gorm[3]. Our decision to employ Gorm was driven by its maturity and reputation within the Go community. Gorm encompasses all the essential ORM functionalities we required, such as database object creation, retrieval, updating, and deletion. With Gorm, we were able to effortlessly define Go objects with the necessary fields for smooth integration with our database.

**Vagrant** In our project, we employ Vagrant[4] to streamline the configuration and deployment of the virtual machine that hosts our application on the DigitalOcean server. Vagrant allows us to give an generic description of the virtual machine we need, which it utilizes to create the virtual machine. This eliminates the need for manual interaction with programs like VirtualBox, sparing us from navigating through UIs. By automating this process, we mitigate the risk of errors during each VM setup, ensuring consistency in our deployments. Vagrant was selected over alternative solutions due to its versatility and compatibility with various containerization platforms, seamlessly integrating with different development ecosystems. Furthermore, it provides cross-platform support for Windows, MacOS, and Linux, making it a suitable choice for our diverse operating system requirements.

**Prometheus & Grafana** We have adopted Prometheus[5] to extract valuable insights from our Minitwit application due to the following reasons: Prometheus offers extensive support for a wide range of metrics, making it highly versatile for monitoring needs. It seamlessly integrates with the Go programming language. By utilizing Prometheus, we can effortlessly integrate it with popular dashboard tools like Grafana[6], enabling us to create visually appealing and informative dashboards. In addition, Prometheus allows us to set up alerts that notify our team whenever certain metrics exceed predefined thresholds. In conjunction with Prometheus, we utilize Grafana to visualize the data collected. We specifically chose Grafana for its user-friendly interface and ability to create highly customizable dashboards that require minimal configuration efforts.

**Elasticsearch, Filebeat & Kibana** To facilitate logging in our Minitwit application, we implemented the EFK stack. This stack proves to be an ideal solution for our needs, offering

[3] gorm.io
[4] vagrantup.com
[5] prometheus.io
[6] grafana.com

several key advantages. Firstly, it incorporates Filebeat, a lightweight log shipper that collects, parses, and forwards logs from our application to a storage component. This log transportation ensures efficient and reliable log management. Secondly, the stack utilizes Elasticsearch as an efficient storage and analytics engine, enabling us to handle large volumes of data while providing excellent full-text search capabilities. Lastly, Kibana, the dashboard component of the EFK stack, simplifies setup and offers a user-friendly interface for filtering queries on the data stored in Elasticsearch. With Kibana, we can easily access and display the most valuable log data based on our requirements.

**DigitalOcean** Our chosen cloud infrastructure provider is DigitalOcean[7]. DigitalOcean stands ouut for its user-friendly interface, specifically designed for setting up and deploying Droplets (VMs) and Volumes (storage). In general, the platform is known for its simplicity and affordability, which aligns perfectly with the requirements for our Minitwit application. DigitalOcean's reputation for simplicity and affordability, combined with a student benefit they offer, makes it an ideal choice for our Minitwit application. It allows us to effectively manage our infrastructure within our budget while taking advantage of the student benefit.

**GitHub & GH Actions** To streamline our CI/CD workflow, we utilize GitHub Actions due to it being integrated in GitHub, which is our chosen version control system. By utilizing GitHub Actions, we consolidate our workflow within a single platform, eliminating the need for multiple tools. Additionally, it being integrated in GitHub allows us to conveniently track the status of individual commits as we push them. This visibility enables us to promptly identify build errors or failures in tests associated with specific commits. Furthermore, GitHub Actions allows for Continuous Delivery and Deployment capabilities, automating the process of delivering and deploying changes seamlessly using the same system.

## 2.3 Current state of the system

At the time of writing (20-05-2023) our static code analysis tools, which are explained in section 3.3, report the following:

- **Sonarcloud**: Latest pull request passed with 0 bugs, 0 vulnerabilities, 0 security hotspots and 10 code smells. We have assessed the code smells and decided to ignore them as they are insignificant.
- **Snyk**: It is currently disabled as it caused issues that blocked our CI/CD chain. We have not prioritized to fix the underlying problem as we believe the other code analysis tools provide a satisfying coverage of potential security vulnerabilities.
- **CodeQL**: Latest deployment passed CodeQL in the CI/CD chain.
- **Dependabot**: No active pull requests for vulnerable dependencies.

[7] digitalocean.com

- **Lint for Go**: Latest deployment passed Lint in the CI/CD chain.

### 2.4 License compatibility

Our project is under GPL 3.0 which is compatible[8] with all dependencies, namely: *MIT License*, *GNU Affero General Public License v3.0*, *Apache License 2.0*, *Server Side Public License*, *ISC*, *BSD-3-Clause*.

## 3 Process's Perspective

In this section we present how we organized the development process.

### 3.1 The team

Dudes is a *close-knit team* consisting of five Master students. We value an in-person working environment when possible, which also means that a majority of project work and general interactions have taken place in-person at school during the course hours. With our collective expertise, we have tackled most challenges collaboratively, working together to find solutions. This approach has fostered a seamless workflow, enabling efficient problem-solving and encouraging a sense of shared accomplishment within our team. For work intensive weeks where we did not manage to finish within the course hours, we usually continued on Fridays or Saturdays either at school or remote, but still collaboratively.

### 3.2 Version control setup

Our version control system setup revolves around a monorepo approach, providing a centralized repository to manage our codebase. We have chosen this approach to consolidate all related code in a single repository, promoting the ease of code reuse and streamlining collaboration. To enhance our workflow and issue tracking, we utilize GitHub's built-in issues extensively. The issues created in GitHub follow predefined templates[9], defined with GitHub issue templates, in order to ensure consistent issue management and to establish guidelines for issue creation, assignment, and resolution. These templates provide a structured framework for issue creation, ensuring all necessary information is captured for the respective type of issue, whether it be bug reports, feature requests, or security vulnerabilities. Additionally, we have implemented a well-defined branching strategy, as shown in figure 2, consisting of a main branch as the primary branch, and subbranches designated for the development of single features or bug. The subbranches are named according to a set of naming schemas in order to provide clarity and organization. The naming schemas consists of a predefined tag followed by a relevant name, such as "feature/feature-name" or "bug/bug-name". Before

[8] gnu.org/licenses/license-list.en.html

[9] github.com/Lindharden/DevOps/tree/main/.github/ISSUE_TEMPLATE.

merging any changes into the main branch, a set of checks is performed to ensure code quality and stability.

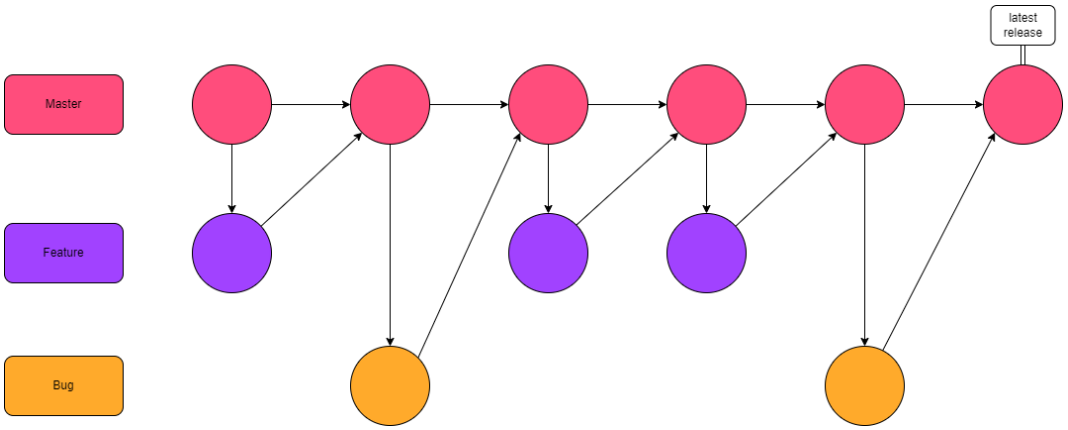


**Figure 2:** Illustration of our GitHub branching flow.

This branching strategy enables seamless feature development, bug fixing, and code maintenance, ensuring a streamlined development process and promoting collaboration within our team. To track releases and provide visibility into the version history, we utilize the GitHub Releases feature. This feature is configured to automatically create a new release for each deployment, ensuring the latest release is always up-to-date with the deployed version. In addition to the automatic releases, we also implemented a weekly release schedule. This manual process allows us to consolidate any significant and relevant features or updates that have been developed during the week. By creating weekly releases, we provide a clear and structured approach to showcasing our progress and informing about the latest developments.

## 3.3 CI/CD

For our CI/CD setup, we leveraged the capabilities of GitHub Actions to automate the verification process when code is merged into the main branch. GitHub Actions allows us to define custom workflows that encompasses various stages of our pipeline. Within these workflows, we incorporated essential steps to ensure code quality and reliability.

Firstly, the actions execute unit tests to validate the functionality of our codebase, ensuring that it meets the expected behavior. Secondly, we integrated static analysis tools into the workflow, specifically Lint for Go, Snyk, CodeQL, and SonarCloud. These third-party services thoroughly analyze our code to detect potential security vulnerabilities, code smells, and many other potential issues, assisting in maintaining a high level of code quality. Lastly, once all the checks pass successfully, GitHub automatically builds a Docker image of our project and deploys it to our hosting service. This ensures that the latest version of the application is available to the users.

To further enhance our development workflow, we integrated GitHub Actions with the pull request feature. This integration enables us to run tests and analysis on proposed changes before merging them into the main branch, minimizing the risk of introducing faulty code into the main branch.

The integration capabilities and the support for automated tests provided by GitHub Actions were key factors in our decisions to choose this CI/CD solution. Our setup streamlines the development process, automates quality checks, and ensures our code is validated and ready for deployment upon introducing changes.

## 3.4 Monitoring & logging

The monitoring system we have implemented for Minitwit is Prometheus. We have implemented application monitoring and infrastructure monitoring with the following metrics:

- **Number of tweet per hour** - This helps us determine the load our application is under. With this we will be able to see whether the system is overloading or not.
- **Error codes** - This metric shows us the total amount of errors which the system reports. This metric can be used to determine whether everything runs as expected. If the dashboard displays an unusual amount of errors, this might be a hint that something is going on with the system.
- **Response time** - This metric tracks how many milliseconds a user has to wait for a response. This is useful to see if any network issues arise and also to ensure that we uphold the expected average response time of <50ms included in our API SLA.[10]
- **Hardware load** - This metric reports system load, CPU and memory usage, disk I/O and amount of network transmit and receive requests. This can be useful for determining whether everything runs as expected and whether the hosting server has sufficient resources to handle the incoming load.

We use a Grafana dashboard to display the data collected for each metric through Prometheus, which can be seen in figure 3. The dashboard is configured through a JSON file[11]. Both Prometheus and Grafana have been integrated into the application through the docker-compose.yml configuration file.[12]

[10] github.com/Lindharden/DevOps/blob/main/SLA.md
[11] github.com/Lindharden/DevOps/blob/main/monitoring/grafana/dashboards/main.json.
[12] github.com/Lindharden/DevOps/blob/main/docker-compose.yml.

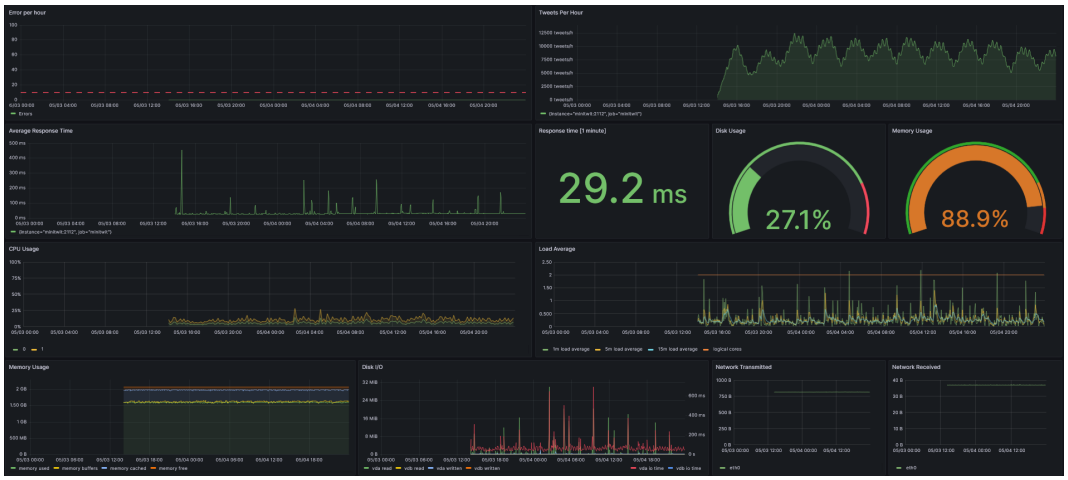


**Figure 3:** Grafana dashboard displaying the data collected with Prometheus.

The logging system we have implemented is an EFK stack consisting of Elasticsearch, Filebeat and Kibana. We used the *Zap* package for Go to nicely format and emit log messages that Filebeat collects. We have two types of log messages, the first being the native Ginlog messages that happen for every request. The second one is the log messages that we have implemented in the code, which are formatted by *Zap* in JSON as follows (values are examples):

```
{
    "level":"error",
    "timestamp":"2023-03-25T13:09:38Z",
    "caller":"controllers/loginController.go:130",
    "msg":"Could not find session",
    "user":"John Doe",
    "stacktrace":"..."
}
```

The Kibana dashboard can be seen in figure 4. The EFK stack is integrated into the application through the docker-compose.yml configuration file.

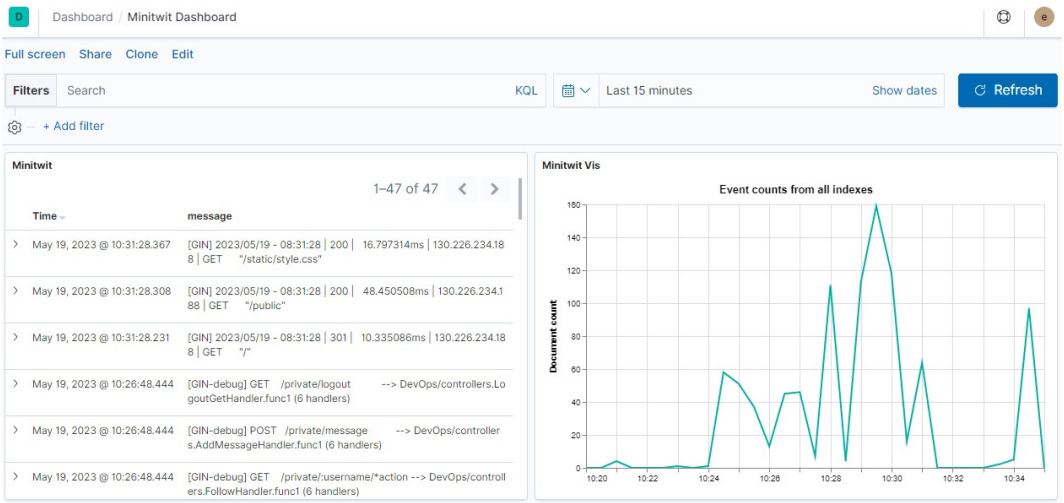


**Figure 4:** Kibana dashboard displaying the logs that are collected, organised and stored by Filebeat and Elastic search.

## 3.5 Security assessment

In our security assessment we have considered which risks the infrastructure and application are vulnerable to. The risks we have assessed are mapped in a risk assessment matrix which can be seen in figure 5. More concretely they are:

- **Compromised GitHub repository** - The GitHub repository owner uses two-factor authentication. Furthermore, all users must make a pull request and have it approved to change the code on the main branch. This makes it unlikely that the GitHub repository is compromised. However, a compromised GitHub repository would mean that the attacker would gain access to one of our GitHub admin accounts, which would mean they have the ability to change our code, making it a significant impact risk.
- **White-box exploits** - As our GitHub repository is public, anyone can see our code and create exploits for it. Therefore if the attacker finds a reason and an opportunity to create a white-box exploit, it would likely happen. However, as we continuously update our code and update vulnerable packages, the chance of an exploit being possible is very low. As the attacker knows what our code base looks like they can plan their attacks towards having as large of an impact as possible. In addition, the attacker can see all the dependencies we use, meaning they can utilize any exploits found in our dependencies, making it an extensive impact risk.
- **DDoS attack** - Since our application is not that popular it is not likely that it will get DDoS'ed, but it is still possible since our application is public. A DDoS attack would mean that our application would be unreachable for a period of time. This would be a moderate inconvenience for us and our users, making it a moderate impact risk.

- **SQL injection** - Since our database abstraction layer GORM sanitizes all keywords, an SQL injection attack is very rare, basically impossible. SQL injection would mean that an attacker performs attacks directly on our database, by sending arbitrary SQL queries, making it an extensive impact risk.
- **DB leak** - The only ways our database can be leaked, is by someone compromising our code repository and performing some kind of white-box exploit, or by performing SQL injection. As we deemed these risks as "rare", our database leaking must also be a rare event. If our database leaked, the attacker would gain access to usernames, email's and hashed passwords of our users. As none of this is sensitive information the impact would be negligible.

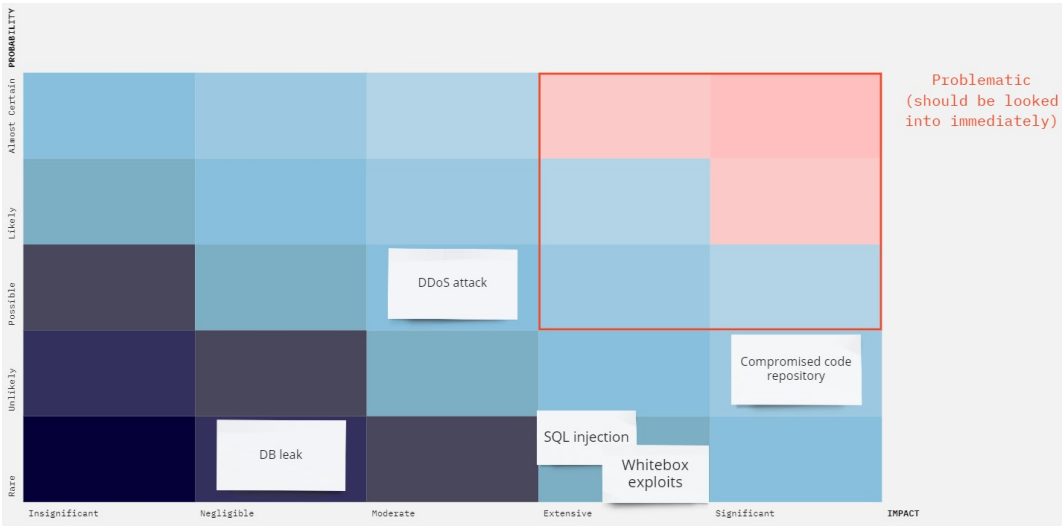


**Figure 5:** Risk assessment matrix. The vertical axis indicates the probability of a risk while the horizontal axis indicates the impact of a risk. Risks in the top-right corner are considered problematic and should be handled accordingly.

None of the assessed risks are placed inside the problematic area in the risk assessment matrix in figure 5. Therefore, we have not taken any actions against these risks, we have simply decided to accept them for now. Additionally, we generated a vulnerability report using Zap which can be found here: github.com/Lindharden/DevOps/blob/main/docs/OWASP_ZAP/report_290323/2023-03-29-ZAP-Report-.html. This report didn't show any major vulnerabilities.

## 3.6 Scaling & redundancy

To ensure fault tolerance we introduced redundancy in the form of a replica. We created an additional droplet on DigitalOcean and changed our Vagrantfile accordingly to have identical instances running on each of the droplets. One of the instances act as the primary one, for which all traffic is directed to. The other instance serves as a backup. With the use of

heartbeats the server can keep track of the primary replicas state, and should it fail to show signs of life, the backup replica will take over as primary.

In addition to fault tolerance, having a primary and backup replica also enables rolling updates which eliminates the inevitable downtime caused from deploying updates. We did not get around to do it for this project, but we had a plan of how we wanted to do it. In order to increase our availability, we will first be updating the backup replica and make it our primary server, then we can shut down the replica which was previously the primary one and update it as well. We facilitate this process by using DigitalOcean's API to automatically switch the primary server to the running replica.

# 4 Lessons Learned Perspective

During the development of the project, we had some issues while setting up new services in Docker. We did not have prior experience with many of the services such as Grafana, Prometheus, and Kibana, and therefore we found that it took way longer than expected to get these services to run properly. There was not a simple solution, other than trying until we got it right, and this was partly because of how hard it was to narrow down problems as the Docker logs are not always that elaborate. From this we learned to dedicate more time to tasks that involve the implementation of new services.

Our project utilize many different tools for code quality, and security assessment, hereunder Snyk. We implemented Snyk into our CI/CD chain, where it's supposed to stop the deployment if it finds any major problems with the commit. This ran fine for a while until something suddenly went wrong in the setup. From this point on Snyk failed our CI/CD pipeline, not because of any vulnerabilities, but because of some other unidentified problems[13]. We didn't find any solution to this, so we solved the problem by disabling Snyk entirely. We deemed this fine for now, as we have other tools which also check for vulnerabilities in our application.

## 4.1 DevOps style of work

For a majority of other school related development projects we have not utilized a concrete tool to organise tasks and issues. For this project we took great advantage of GitHub Issues to have a centralized backlog that was also streamlined using issue templates. GitHub Issues provided a much needed overview of features and bugs and made it easy to manage and distribute responsibilities.

The DevOps handbook [Kim+21] mentions the Three Ways. This is our thoughts on how we adhered to each of the principles:

- **Flow**: We adhere to the Flow principle which talks about implementing a fast left-to-right flow, which is the time it takes from when requests are put on the backlog, till

[13] Problem with Snyk can be seen in GitHub Actions: github.com/Lindharden/DevOps/actions/runs/4660817522.

they are implemented and is in production. Our setup has a fast Flow as we have a CI/CD setup which automatically builds, tests and deploys changes from the main branch to our remote server at DigitalOcean. This means that the time it takes from when changes are committed, till they are in production, is just as long as it takes for our team to approve and merge pull-requests. We also try to split tasks into as small as possible backlog items (GitHub Issues), such that they are easier to delegate to different group members, and thus pass through the pipeline faster.

- **Feedback**: We adhere to the Feedback principle, which focuses on continuous problem solving when they occur. Our application don't have real users, but we simulate the feedback process by receiving error messages from the user simulation run by the teaching team. When we see that errors occur, we create an Issue/ticket which goes in our backlog. We then try to fix the error as fast as possible.
- **Continual Learning and Experimentation**: We try to adhere to Continual Learning and Experimentation where we encourage risk taking and see mistakes as opportunities to learn. We are in a special situation as we are not implementing a real application, and therefore we can afford to make big mistakes without any bigger impacts. We relieve the pressure of having to manually deploy the changes we make, by having an CI/CD setup which automatically deploys all the changes we make.

# Literature

[Kim+21] Gene Kim et al. *The DevOps handbook: How to create world-class agility, reliability, & security in technology organizations*. IT Revolution, 2021. Chap. 1-4.