# *minitwit* Project Report

ITU DevOps 2025 Group `E`

Magnus Thor Lessing Rolin thmr@itu.dk
Mathias Bindslev Hansen bimh@itu.dk
Nikolai Tilgreen Nielsen nitn@itu.dk
Rasmus Hassing Huge rahu@itu.dk
Rasmus Mygind Poulsen rpou@itu.dk

# 1 Table of Contents



---

Package Diagram

Figure 1: Package Diagram

Deployment Diagram

Figure 2: Deployment Diagram

# 2 System's Perspective

## 2.1 Design and Architecture

### 2.1.1 Static view

The application follows the Onion architecture and is split into three layers.

- The **Domain Layer** contains the domain model.
- The **Infrastructure Layer** contains the data manipulation and insertion logic.
- The **Application Layer** contains the entrypoint of the applications and defines the endpoints of the API. This layer also contains the UI.

The infrastructure is deployed to Digital Ocean.

- The minitwit application is hosted on two droplets - a primary and secondary.
- A nginx loadbalancer distributes load between the two minitwit servers.
- The Database is hosted as a PostgreSQL Database Cluster.
- Logging is hosted on its own seperate droplet.

### 2.1.2 Infrastruture as Code

The infrastructure above can be deployed with Terraform. The infrastructure as code is documented in the `/terraform` directory. This includes modules for provisioning the application servers and logging stack.

### 2.1.3 Dynamic view

## 2.2 Dependencies

Here is a list of all dependencies.

```
# Dependency List:
1. Microsoft.EntityFrameworkCore.Design - Version: 9.0.1
2. Microsoft.Extensions.Configuration - Version: 9.0.2
3. Microsoft.Extensions.Configuration.EnvironmentVariables - Version: 9.0.2
4. Microsoft.Extensions.Configuration.UserSecrets - Version: 9.0.2
5. Npgsql.EntityFrameworkCore.PostgreSQL - Version: 9.0.4
6. prometheus-net - Version: 8.2.1
7. Serilog.AspNetCore - Version: 9.0.0
```

8. Serilog.Sinks.Console - Version: 6.0.0
9. Microsoft.AspNetCore.Identity - Version: 2.3.1
10. Microsoft.EntityFrameworkCore.Sqlite - Version: 9.0.1
11. Microsoft.AspNetCore.Identity.EntityFrameworkCore - Version: 9.0.1
12. Microsoft.AspNetCore.Identity.UI - Version: 9.0.1
13. Microsoft.EntityFrameworkCore.Tools - Version: 9.0.0
14. Microsoft.VisualStudio.Web.CodeGeneration.Design - Version: 9.0.0
15. prometheus-net.AspNetCore - Version: 8.2.1
16. Serilog - Version: 4.2.0
17. Serilog.Formatting.Compact - Version: 3.0.0
18. Serilog.Sinks.Elasticsearch - Version: 10.0.0
19. Serilog.Sinks.Network - Version: 2.0.2.68
20. Serilog.Sinks.Async - Version: 1.5.0
21. coverlet.collector - Version: 6.0.4
22. Microsoft.AspNetCore.Mvc.Testing - Version: 9.0.2
23. Microsoft.NET.Test.Sdk - Version: 17.13.0
24. Microsoft.Playwright.NUnit - Version: 1.50.0
25. xunit - Version: 2.9.2
26. xunit.runner.visualstudio - Version: 3.0.0
27. Postgres - Version: 16.9
28. Kibana - Version: 8.12.1
29. logstash - Version: 8.12.1
30. elasticsearch - Version: 8.12.1
31. Nginx - Version: 1.27.0
32. Dotnet_SDK - Version: 9.0.0
33. org.Sonarcube - Version: 6.1.0

**2.2.0.1 Logging** For logging, our application uses Serilog to collect log data. This data is then transferred into the ELK Stack, which consists of Logstash, Elasticsearch, and Kibana — all used to process, query, and display the logging data. This setup is hidden behind Nginx, which acts as a reverse proxy and serves as an authentication layer between the user and Kibana.

For elasticsearch “209.38.112.21:8080” use “admin” “admin” to login and access logs.

**2.2.0.2 Monitoring** For monitoring, our application uses Prometheus as a real-time metrics storage server. On top of this, we use Grafana as a data visualization tool to display and analyze these metrics.

For Grafana “164.90.240.84:3000” you can use the given login to access the dashboard.

**2.2.0.3 Application** We have built our application using the .NET software framework, following the onion architecture originally invented by Jeffrey Palermo. We use the ASP.NET Core Identity package as an authentication

Sonar Cube Quality assesment

Figure 3: Sonar Cube Quality assesment

system, allowing us to create and delete users. Initially, we used SQLite as our DBMS but later switched to Prometheus. In both cases, we utilized Entity Framework Core (EF Core) as our Object-Relational Mapper (ORM). For testing, we use NUnit as the primary testing framework, with Playwright layered on top for end-to-end testing. To handle API calls from the simulator, we use the ASP.NET Core MVC framework to create API controllers that process HTTP requests. As a software quality measure, we use SonarQube, specifically integrating their service via a GitHub workflow. SonarQube tracks security, reliability, maintainability, test coverage, and code duplications. As a further software quality measure, we use Hadolint on pushes to our main branch, enforcing warnings as errors to ensure proper Dockerfile syntax.

### 2.3 Interactions of Subsystems

Below you will see how our application handles an unfollow request from both a regular user and the simulator. The key difference is when the 204 status code is sent, as well as the simulator using a batch loader.

**Sequence Diagram for Simulator unfollow call** API Seq Diagram

**Sequence Diagram for User unfollow call** User Seq Diagram

### 2.4 Current State of the System

The current state of our system is generally good. At all levels of the application, we are observing the results we expect and want. The biggest weakness in our application is the lack of testing, which is currently close to zero. Below is the result of a quality check run by our SonarQube workflow.

## 3 Process' perspective – 1026 words

### 3.1 Deployment and Release

The following workflows are implemented to ensure a robust CI/CD pipeline:

1. **Build and Test Workflow**
   This workflow automates the build process and runs all unit and integration tests to ensure code quality.
2. **Build Release Workflow**
   Automatically creates a release when a new tag is pushed to the repository.
3. **Continuous Deployment Workflow**

Deploys the application to the production server upon successful completion of tests and builds.

4. **Lint and Format Check Workflow**
   Ensures that the code adheres to the project's linting and formatting standards.
5. **Scheduled Release Workflow**
   Automates weekly releases to ensure regular updates and maintenance.
6. **SonarQube Workflow**
   Performs static code analysis using SonarQube to identify potential bugs and vulnerabilities.

Each workflow is defined in the `.github/workflows` directory and is triggered based on specific events such as pushes, pull requests, or scheduled intervals.

### 3.1.1 Deployment Chain

The deployment process follows a structured chain format to ensure reliability and minimize downtime. The steps are as follows:

1. **Linting and Code Quality Checks**
   The code is first analyzed for adherence to linting and formatting standards. This ensures that the codebase remains clean and maintainable.
2. **Integration Testing**
   Once the linting checks pass, the commit undergoes rigorous integration testing to validate that all components work together as expected.
3. **Deployment with Rolling Updates**
   If the commit successfully passes all previous stages, the deployment process begins. Rolling updates are utilized to ensure a seamless transition. This approach guarantees that if the deployment encounters any issues, an unaffected backup server remains operational to handle the workload while the problem is resolved.

This deployment strategy ensures high availability and minimizes the risk of service disruption during updates.

## 3.2 Monitoring

The application utilizes Prometheus and Grafana for monitoring. Prometheus scrapes port 5000, the minitwit application, and sends the data to the /metrics endpoint. Then Grafana retrieves the data from /metrics, and uses this as its data source. The relevant information could however not be found in the default configs . In the `MetricsService.cs` file, there are custom metrics to our application, such as the "minitwit_follow_counter and"app_request_duration_seconds" The follow counter is implemented in the program by adding to the counter, every time a follow request is made.

The duration is measured by starting a timer when a request comes in, and stopping it when the request has been processed.

## 3.3 Logging of application

The application uses the ELK logging stack. In the beginning, the logs contained information from the information level and up. This resulted in a flood of logs, and it was impossible to see anything relevant. It was then configured to only show warnings and above. Here there were practically no logs. From here logging statements were added to the code, to log when problems occured. In the `ApiController.cs` there are custom creation of logs which are logged as warnings. These logs include system failures such as unsuccessful message post and failure to follow a user. This data is sent through Serialog to Logstash. Another important metric is logging of request times. If a request took longer than 300 ms to process, it will log it. This has been central in discovering the ReadTimeout issue, that has occured. To see all the logs for e.g. timeouts, the searchbar is used. Here the user can input "@m: slow", to get them all.

## 3.4 Security assessment

The Application consists of the following assets:

- Web application (Minitwit)
- Monitoring (Prometheus + Grafana)
- Logging (Logstash + Elasticsearch + Kibana)
- DigitalOcean droplets
- DigitalOcean database cluster

**Risk scenarios.**

- R0: DDoS attack overwhelms the server, making the application unavailable.

**General security:**

- R1: Attacker uses exposed secrets to gain access to sensitive data.
- R2: Attacker gains access to our API, and injects large amounts of data into our database, stressing the system.
- R3: An attacker exploits a known vulnerability in an outdated dependency.
- R4: Attacker extracts secrets from unprotected endpoints.

**Web application threat sources:**

- R5: Attacker performs SQL injection on our web application to download sensitive user data.
- R6: Attacker exploits a cross-site scripting vulnerability to hijack a user sessions.

Risk matrix

Figure 4: Risk matrix

- R7: Attacker forces or tricks an authenticated user to do unwanted request to the web application. A malicious site sends a request to the trusted website using the user's cookies and session.
- R8: Attacker can interrupt unencrypted HTTP request and modifies requests.

**Infrastructure threat sources:**

- R9: An attacker scans for open ports and discovers multiple exposed services. this can lead to data exposure and disruption of service.
- R10: An attacker scans for open ports and identifies an exposed Elasticsearch instance listening on port 9200. Since the service lacks authentication, the attacker is able to gain access to vulnerable data.
- R11: An attacker gains SSH access to our droplet and interacts with our running containers.

**Monitoring/logging threat sources:**

- R12: An attacker gains unauthorized access to Elasticsearch logs and backs up sensitive data.

The application is secure against SQL injections. There is no public secrets and dependencies are up to date. Access monitoring and loggin is resticted via login and droplets are protected using DigitalOcean's default security.

the biggest vulnerability is the lack of protection against request spamming and application overloading.

To mitigate DDoS attacks a possible solution is to temporarily shut down the server when the number of requests per minute exceeds a defined threshold. To secure HTTP traffic, HTTPS could be added. To protect open ports, authentication should be required for all exposed services

## 3.5 Strategy for scaling and upgrade

Our project is scalable. You can scale the system vertically by investing more in the hosting provider, or, in our case, we leverage the infrastructure we have built to scale horizontally.

To scale horizontally, we first need to deploy a new application server. After that, we add the IP address of the server to our "load balancer's" upstream server list in the configuration file of Nginx. At this point, the server should be up and running, with the load balancer utilizing the new server to distribute incoming requests. The only remaining steps are to add the new server to the "rolling update" workflow in ".github/workflows/continuous-deployment.yml,"

which involves adding new secrets to the project's GitHub secrets and implementing a safety check in the workflow. Once these steps are completed, the server will be fully integrated into the architecture of the application.

### 3.6 The use of AI

AI tools such as Chat GPT was used for idea generation. When problems occured, and no one knew how to fix it, the AI were asked to see, if an easy fix existed. AI were also used for finding errors in e.g. docker files. This approach often speeded up development, as it often had great suggestions for common issues. Sometimes it was also useless, as it was a somewhat "unique" problem, and it did not know how to fix it. In these cases, the TA's were useful.

## 4 Reflection Perspective – 445 words

### 4.1 Evolution and Refactoring

After implementing SonarQube quality assessment, a lot of code was refactored and renamed. Most of the codes issue was maintainability, where names did not align in different classes. This was quickly changed everywhere. Next issue was long functions, that did many different things. This was refactored out, so that one function has one job. This improved maintainability and readability of the code base.

### 4.2 Operation

The biggest issue this project was the ReadTimeout issues. That meant some requests were lost, and caused problems. E.g. when a user tried to register, but failed. Then all following message- and follow requests failed. Many approaches to this was tried. First the Database calls were minimized. Sometimes the database would be called, when not strictly necessary. This dropped response time by about 30%, but was not enough to actually remove the issue. Then the sql queries were combined, so only one to two trips were needed pr. request. Then a batch service was introduced, so requests would only be added to the database, when 10 requests were gathered. Both of these improvements significantly decreased response time, but on the 10th user, it would insert 10 requests at once, so that 10th users request would often get lost. In the end it was decided that a status code 204 would be sent back immediately after receiving a request. The request would be processed, and handled in the background, while the "user" would think everything went fine. The response time dropped to almost 0, and no requests from that point were lost.

### 4.3 Maintenance

At one point the Grafana and Prometheus data were gone. No one knew why it was gone, and it happened at a time, where everyone was working with

something different. This made it difficult to trace why the error occured. Many hours were spent trying to find the issue, and it was finally discovered, that the snap version of Docker had been installed, over the official version. The snap version saved its volumes somewhere different, and caused it to look in the wrong place, when looking for the previous saved volumes. The snap version was removed, and all the data was back. This resulted in an immediate backup of the volumes, so if the volumes were ever removed, there still was a backup. After this issue was resolved, it was coincidentally discovered that the droplet only had 1 gb left of storage. It was therefore upgraded and disaster was avoided.

## 4.4 DevOps

There was an automatic linter and quality assessment tool, which together gave insights into, what needed to be changed. This removed unnecessary human intervention, that saved a lot of development time.