DevOps, Software Evolution and Software Maintenance

Group D

Group Members:

Victor Memborg-Heinrichsen	vmem@itu.dk
Peter Bjørholm Hansen	pbjh@itu.dk
Lukas Vranic	luvr@itu.dk
Anne-Marie Rommerdahl	annro@itu.dk

BSc Software Development

IT University of Copenhagen

BSDSESM1KU

May 29, 2025

Contents

1 System's perspective

1.1 Architecture of MiniTwit

MiniTwit follows the server-client architecture, and the server is deployed via a Docker Swarm, which consists of virtual machines (aka 'droplets') on DigitalOcean. Data, such as user data, messages, and followers, is stored in a PostgreSQL database, which is also hosted on DigitalOcean.
In the swarm, we have manager-nodes and worker-nodes. The only services allowed to run on the managers are Prometheus and Dozzle (more about the swarm services in section 2.2). Figure 1 shows a specification-level deployment diagram showing how MiniTwit is deployed on a worker node.

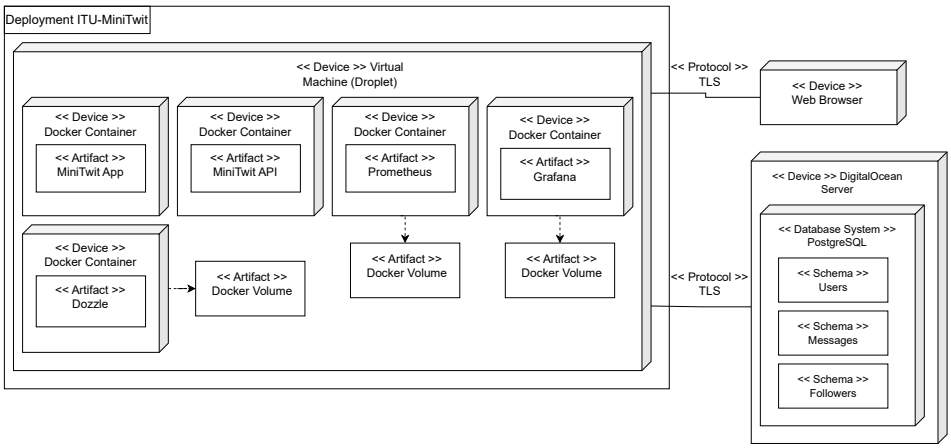

Figure 1: Deployment Diagram of MiniTwit on a worker node

1.2 Dependencies

This project relies on several dependencies across different project areas, including runtime, testing, and infrastructure. The dependencies have been kept up-to-date using Dependabot.

App and API

Our app and API are built in Go 1.23.0, and rely on the following direct dependencies:

- **github.com/gorilla/mux:** Router for HTTP request routing.
- **github.com/gorilla/sessions:** Session management for user authentication.

- **gorm.io/gorm** and **gorm.io/driver/postgres:** GORM ORM and PostgreSQL driver for database operations.
- **github.com/prometheus/client_golang:** Exposes Prometheus metrics endpoints for monitoring.

Testing

- **Go:**
  - **Go's built-in testing package:** Go's native testing framework
  - **Testify:** Used for more assertions and mocking
  - **github.com/mattn/go-sqlite3:** Creates a local testing database
- **Python:**
  - **Pytest:** Testing framework
  - **Selenium:** Browser automation for UI testing
  - **Requests:** HTTP client library for API testing

Infrastructure

- **Docker:** Used for containerizing both the API and the app. Docker compose was used during development to orchestrate services. Later in the project, Docker Swarm was adopted to deploy containers across multiple nodes in production.
- **Virtual Machines:** The swarm nodes are hosted on VMs (DigitalOcean Droplets in this project).
- **Docker Hub:** Used to distribute container images. The CI/CD pipeline pushes API and app images to Docker Hub for swarm nodes to pull.
- **PostgreSQL:** Main production database, managed as a hosted database in DigitalOcean.
- **Prometheus:** Scrapes and stores metrics from both the app and API.
- **Grafana:** Visualizes metrics collected by Prometheus.
- **Dozzle:** Provides a web UI for viewing container logs in Docker Swarm.

1.3 Interactions of subsystems

Depending on the type of request sent to MiniTwit, it will be handled either by the web application or the API. Both components share the same database, ensuring that data remains consistent between them. The sequence diagrams below illustrate how a "post message" request is processed. Figure 2 shows the handling of such a

request through the user interface, while Figure 3 shows how it is handled via the API.

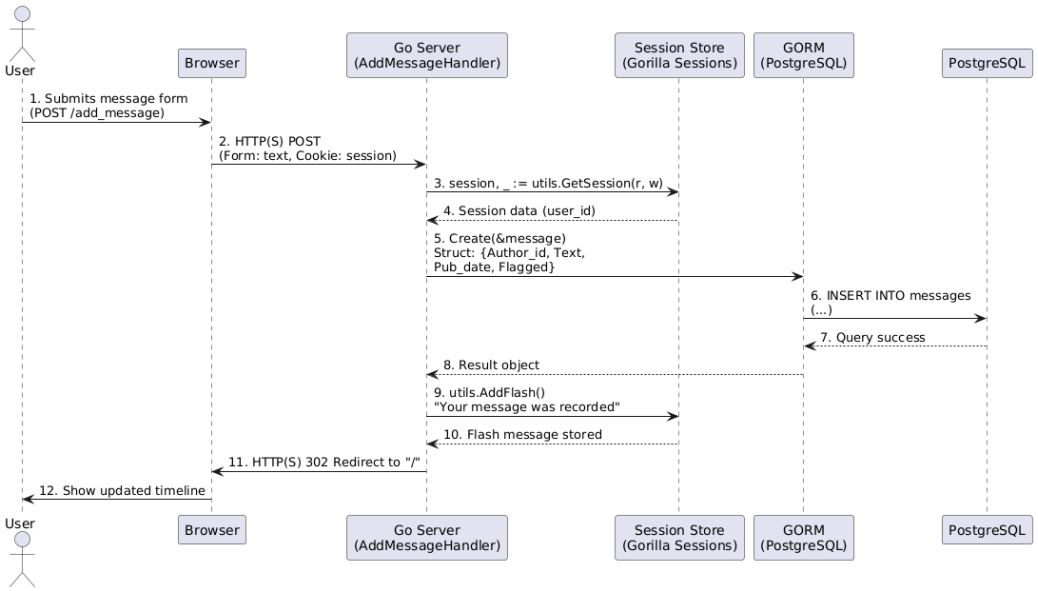

Figure 2: Sequence diagram of a user posting messages.

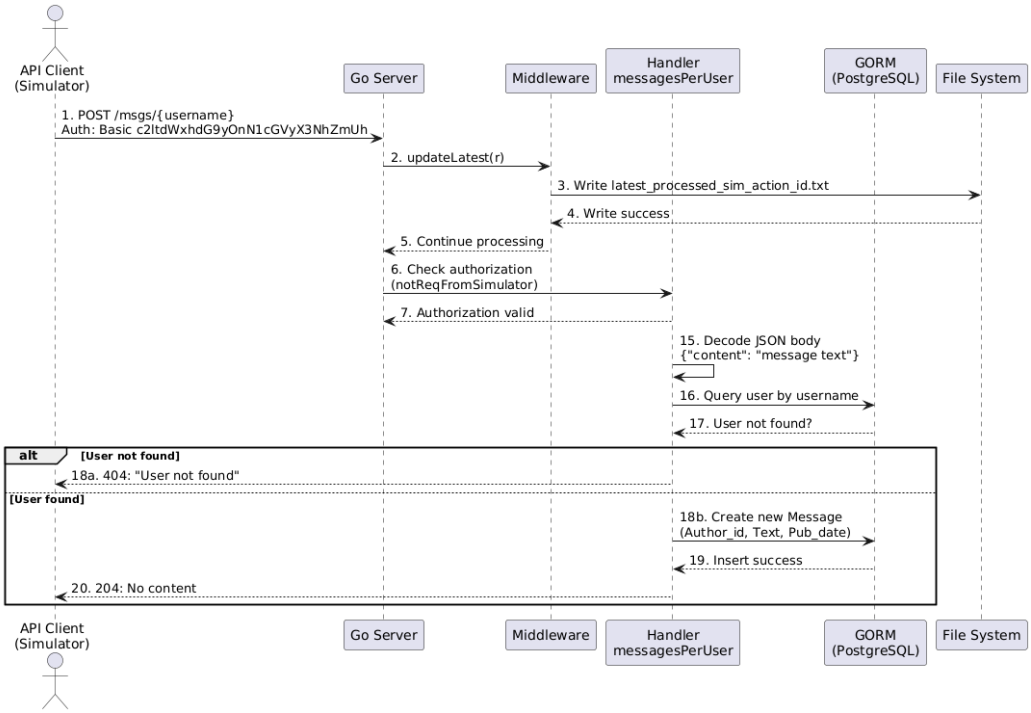

Figure 3: Sequence diagram of an API posting messages.

1.4 Current state of MiniTwit

We have implemented the two tools Sonarqube and Code Climate, which can assist in estimating the maintainability and technical debt of our project.

Code Climate

Code Climate reports 10 code smells, 0 duplications and 0 other issues. This leaves us with an A-rank (see figure 4).

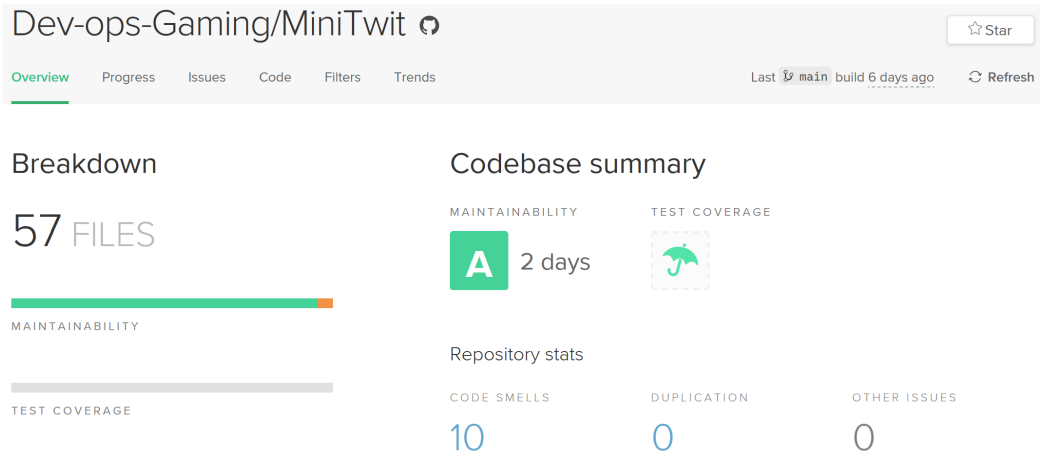

Figure 4: Overview of MiniTwit on Code Climate

SonarQube

SonarQube reports 18 issues - all in the category 'maintainability' - and 0.9% code duplication (see figure 5). A majority of these issues are found in the tests, and many of the issues are in regards to string duplicates. Overall, we've been given an A score for security, reliability, and maintainability.

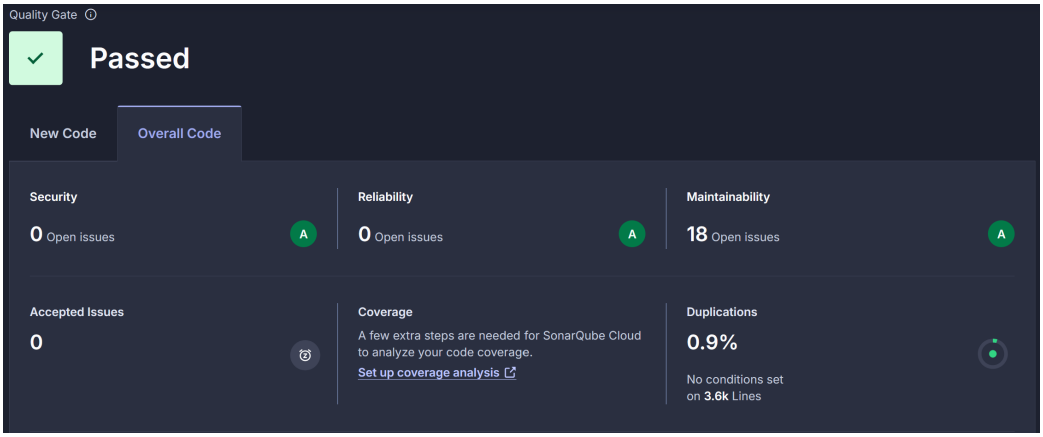

Figure 5: Overview of MiniTwit on SonarQube

2 Process' perspective

2.1 Security assessment

Risk identification

Our assets:

- Web app and API
- PostgreSQL database (Hosted through DigitalOcean)
- Monitoring (Prometheus and Grafana)
- Logging (Dozzle)
- DigitalOcean VMs

Threat Sources:

- SQL Injection
- Exposed ports
- Vulnerable password hashing
- No requirements for user password (length, symbols, etc.)
- Vulnerable dependencies
- No password on Dozzle (logging)
- Cross-Site Scripting (XSS)

Risk Scenarios:

A Attacker performs SQL injection to download sensitive user data

B Attacker performs SQL injection to delete information from the database

C Attacker performs SQL injection to log in to a legitimate user's account

D Attacker brute-forces the hash of a password, letting them log in to a legitimate user's account

E Attacker connects externally to our database, either stealing, modifying or deleting data

F Attacker intercepts messages sent over the network, as they are not encrypted

G Attacker exploits a vulnerable dependency, which lets them perform some malicious attack. As the actual threat level depends on the specific vulnerability, we've chosen to be pessimistic and assume that the consequences will be disastrous

H Attacker accesses our logs on Dozzle, which lets them see all logs

I Attacker uses Cross-Site Scripting to run malicious code

J Attacker performs a DDoS attack

Risk Analysis

Here are the scenarios presented on the risk matrix:

	Rare	Unlikely	Likely	Certain
Catastrophic	B			G
Critical	A			
Marginal			F	E, I
Negligible	C		D, J	H

Discussion of risks

Scenarios A, B, and C have a 'Rare' probability, as we already utilize the ORM-library 'GORM'. GORM uses prepared statements and automatically escapes arguments to avoid SQL injection. Without GORM, the probability would be much higher. Specifically for scenario B, it would also have been a good idea to have a backup of the database. The severity is also not a high risk, as the passwords are encrypted and no other sensitive user data is stored.
Scenario F was dealt with using the TLS protocol. Our Minitwit application using TLS is hosted at https://lukv.dk.
For scenario G, we have added Dependabot, which will automatically search for outdated dependencies, and create pull requests to update such dependencies.

For Scenario E, we should have set up some sort of firewall (which we could do on DigitalOcean), such that only specific services are allowed to establish a connection to the database. Scenario I can never truly be prevented, but there are ways to mitigate the risk, such as sanitizing inputs.
Scenario H can be prevented by locking our Dozzle interface with a password. However, no sensitive data is logged in Dozzle, and thus it is considered low priority.

2.2 Scaling and upgrades

Scaling

To scale our Minitwit application, we performed horizontal scaling using Docker Swarm. We created three manager nodes and four worker nodes.
We have replicas of the following services:

Service	Replicated/Global	No. of replicas
Minitwit App	Replicated	3
Minitwit API	Replicated	4
Prometheus	Global	N/A
Dozzle	Global	N/A

Prometheus and Dozzle are set to be global services, meaning that one instance of each runs on every node in the swarm. This ensures that monitoring and logging are consistently performed on all nodes, preventing any loss of critical information, whether it is logging or monitoring, due to missing coverage.

Upgrades

We use a rolling update strategy with a start-first-order to update the services in our swarm. This means that one replica at a time is updated by starting a new container before stopping the old one, ensuring minimal downtime. Updates are monitored for 30 seconds, and if any failures are detected, our swarm will automatically roll back to the previous working version [1].

2.3 CI/CD chain

We are using GitHub Actions to automate the testing and deployment of Minitwit. There are 5 GitHub Actions workflows:

1. Deploy services to DigitalOcean
   (a) Runs when there is a push to `main`.
   (b) This workflow builds and pushes Docker images to Docker Hub, and then pulls them to the server on DigitalOcean.
2. Run tests and linter
   (a) Runs on any push or when a PR is updated/created.

   (b) Runs linters and tests on new commit or PR.

3. Release MiniTwit

   (a) Runs every thursday at 23:30 UTC.

   (b) Creates a weekly release of MiniTwit.

4. Run linter

   (a) Runs when called by other workflows.

   (b) Lints Go files using golangci-lint.

5. Hadolint on dockerfiles

   (a) Runs when called by other workflows.

   (b) Lints Docker files using Hadolint.

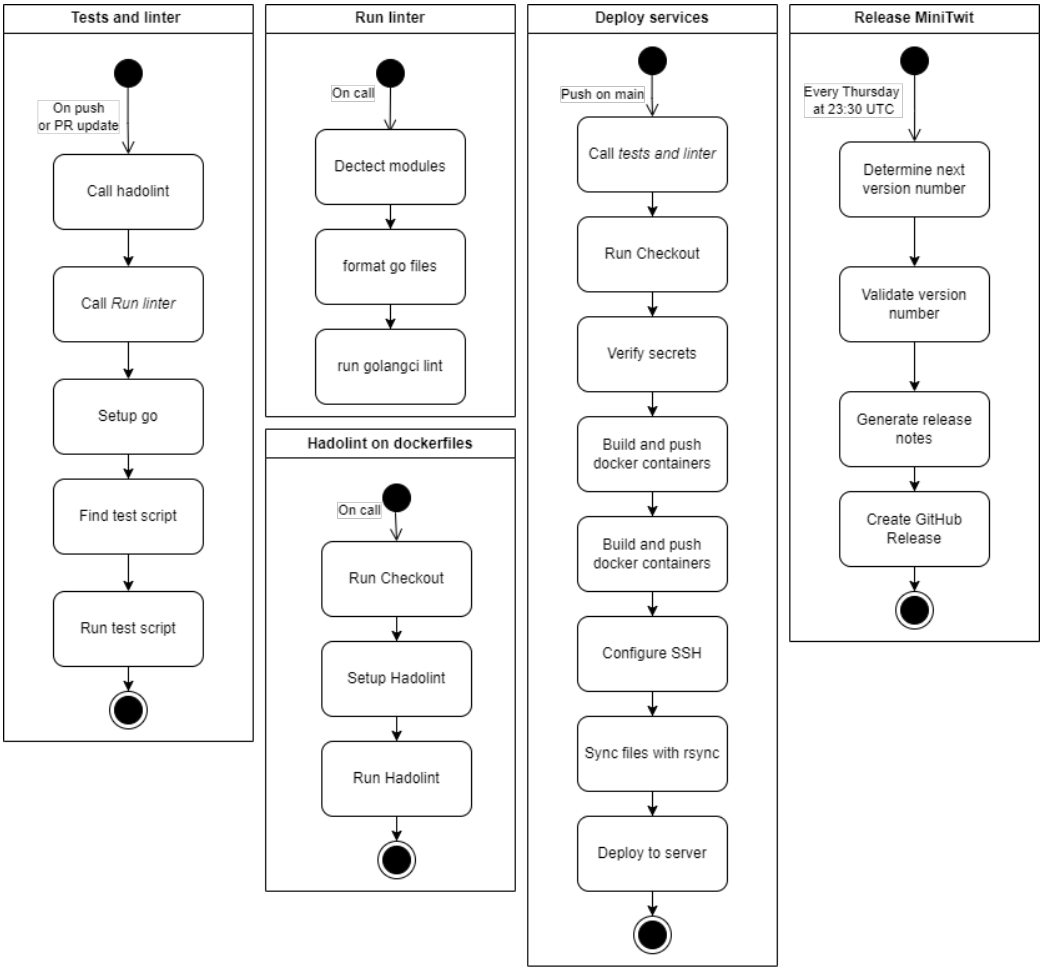

Figure 6: Activity diagram of the workflows.

2.4 Monitoring

Prometheus

Prometheus was integrated into both the API and the application through a custom middleware that intercepts HTTP requests to gather metrics. The metrics collected includes:

- `http_requests_total`: A counter that counts the total number of HTTP requests. It includes labels for request path, HTTP method, and status code. This metric helped us monitor request load per endpoint, track response statuses (in general or just on an endpoint basis).
- `http_request_duration_seconds`: A histogram measuring the time taken

to process HTTP requests, in seconds. It is labeled by request path and method, enabling performance benchmarks for endpoints.

- `http_response_messages_total`: A counter that logs the total number of HTTP responses, categorized by status code and message type. Before logging was fully implemented, this metric was particularly helpful in identifying which endpoints triggered specific status messages and understanding the reasons behind them. A bit deprecated as soon as logging was implemented.

Seen in retrospect, it would have been a good idea also to collect metrics on database queries, but this will be described more thoroughly in section 3.3

Grafana

While Prometheus was important for collecting data, Grafana was utilized to visualize the data. Grafana connects to Prometheus as a data source, enabling the creation of dashboards. As mentioned in section 2.4, we monitored both our API and app, which resulted in us setting up two dashboards: one for the app and one for the API (see figure 7).

For instance, the API dashboard converted the metrics collected by Prometheus into a clear visual representation of the API's health and performance. Panels were set up to showcase:

- Indicators from `http_requests_total`, such as the rate of requests per second, overall request load distributed by endpoint, and the ratio of successful and client-error status codes.
- Performance monitored by `http_request_duration_seconds`, including average response times and 99th percentile latency, allowing for quick identification of performance degradation.

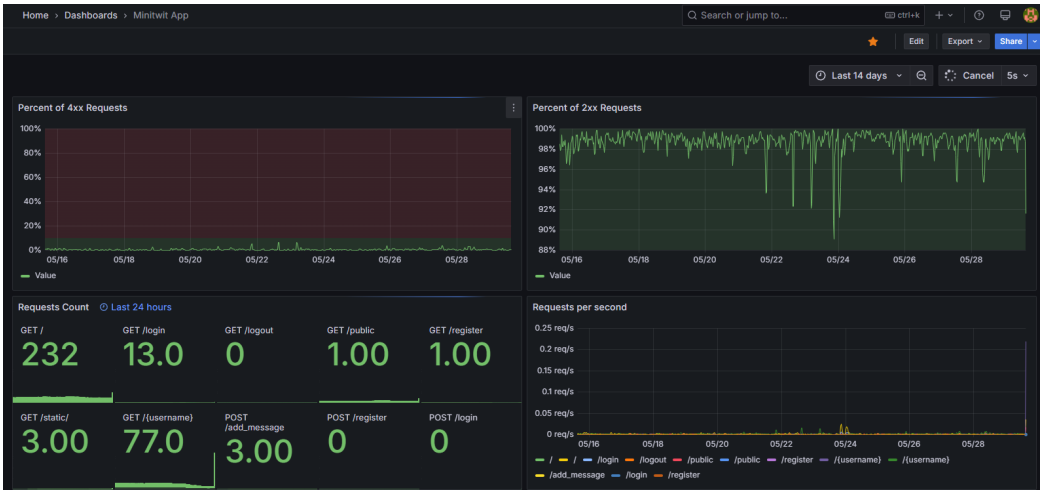

Figure 7: A snippet of our App dashboard in Grafana

2.5 Logging

Logging in MiniTwit was implemented using Dozzle. When we initially wanted to implement logging, we implemented an ELK stack in our standard Docker Compose. This was a very basic implementation with no parsing or user setup. However, this never made it to production because while it was being worked on, we implemented Docker Swarm, and suddenly, a lot changed for our system. Meanwhile, another group member noticed a TA proposing a look at Dozzle if you had problems with ELK. Dozzle was implemented with great success in no time and worked great for us.

What do we log in Minitwit?

In Minitwit, we log application events using Go's standard logging functions, including error messages from our main application and API services. Our database layer (GORM) is configured to log at a warning level. We also log API error responses and critical application failures.

How do we aggregate logs?

All logs in Minitwit are aggregated at a container level using Docker's built-in logging, with every service writing to `stdout` and `stderr`. We use Dozzle, deployed in global mode in our swarm, to view and search logs from all running services in real time through its web interface.

Why we choose Dozzle

We ended up using Dozzle because it enabled us to filter logs by container and log level. We can transmit logs from our services in real time and filter them by service and different logging levels. A sidebar is available, which is equipped with a fuzzy search ability and provides a comprehensive overview of each service, node, replica, and swarm manager. It offers a standard search within the logs for manual searches and the ability to conduct Regex searches within the logs. Dozzle is a simple and effective method of logging, as it is not infrastructure-intensive and is extremely lightweight. Implementation within a Docker swarm is effortless with Dozzle. Dozzle was the optimal choice for us due to its user-friendly interface and all of the aforementioned.

ELK stack features we missed out on

We cannot implement a role-based access control or authentication system with Dozzle, which is unsuitable for production. As of now, unauthorized users can see our logs, and that is, as mentioned, potentially a security breach. We cannot save our logs indefinitely, and cannot distinguish between what should be parsed and what should not. Additionally, we are forfeiting the performance that Elasticsearch would have offered through its robust indexing and search capabilities.

While there is a lot more we miss out on not using an ELK stack, these are the ones we feel were relevant to us. An ELK stack would require much more management, and you could almost endlessly improve it. We feel that Dozzle provided us with what we needed. Combined with our monitoring, we have the opportunity as developers to quickly gain an overview of bottlenecks or bugs in the application and fix them accordingly.

2.6 Usage of AI-assistants

We utilized two large language models (LLMs) throughout the project to support our development process: GitHub Copilot and ChatGPT. GitHub Copilot was primarily used for code completion and was constantly active during development. ChatGPT, on the other hand, was used to quickly gain an overview of how to attack an issue, e.g., a database index issue, which is described in section 3.2, or conveying the essential documentation for e.g., our web framework ”Gorilla”.
We found that the LLMs significantly improved our workflow by accelerating learning and implementation. However, we suspect Copilot of introducing the API issue also described in 3.2, so it is important to be critical when utilizing these tools.

3 Reflection Perspective

3.1 Evolution and refactoring

Introduction of technical debt

One of the challenges we faced during the refactoring phase of the MiniTwit project was the accumulation of technical debt. Refactoring often took longer than anticipated, and tasks that weren't tracked tended to be forgotten. To address this, we created some informal meetings after our lecture, where we could review progress, create new tasks on GitHub issues, and assign the responsible person for the tasks. Although this helped, in hindsight, we should have enforced the creation of issues a bit more, especially during the later stages of the project. We should probably have used some Kanban board, e.g., GitHub projects, to align with DevOps practices [2] and improve collaboration. However, we found that tools such as SonarQube and Code Climate helped us remove some of the technical debt we had previously introduced.
Such practices are quite different from other development projects we've faced, where there is often a tendency to focus entirely on providing the required functionality - with little thought given to the quality of the code, or the structure of the work, as long as the finished product fulfills the requirements.

3.2 Operation

Adding indexes

As we monitored the performance of our API and web application throughout the project, we noticed a degradation in response times. At one point, the `/public` endpoint was particularly affected, with average response times reaching up to 70 seconds. This was unacceptable, and we began investigating the cause.

Although we had not implemented database query monitoring through prometheus, we suspected the bottleneck might lie in our SQL queries. Looking at the performance metrics provided by DigitalOcean, where our PostgreSQL database is hosted, we identified that the `queryMessages` method was the cause of this issue, and it became clear that we needed to index on frequently queried columns. The indexation resolved the issue, and our `/public` endpoint has since had a response time of 20-50ms. In future projects, it will be optimal to monitor the database execution time and visualize it as a `Time series` in Grafana to see if queries degrade over time.

API terminating

The very first issue we encountered was the API becoming unavailable due to crashing when the database encountered an error. We had a hard time debugging this, as the crash did not provide any valuable information in the Docker logs (a logging tool had not been introduced yet), only to discover that we used `panic`, which causes the program to terminate in Go. This took a bit of time to realize, but when we finally traced the issue, we replaced the panic call with returning a proper status code and tried to enforce more thorough code reviews, better logging and although implemented a bit late, we integrated our testing suite into our CI flow as shown in section 2.3, which also aligns well with DevOps principles of automating tasks [2].

3.3 Maintenance

Disk usage

One valuable lesson we have learned during the course was the importance of managing disk space in production. At one point, we lost the ability to SSH into our main droplet. We used the recovery console to investigate the issue and discovered that the disk was full. After analyzing the disk usage, we found that whenever we deployed a new version of minitwit, we pulled the latest image, but didn't delete the old one, which slowly consumed all the storage. To resolve this, we removed the old images, scaled the storage vertically, and have since been more observant of disk usage. Although not yet implemented, it would also be a good idea to prune old unused images automatically when pulling new ones and visualize the disk space on Grafana.

4 Artifacts

- MiniTwit repository
- Report repository
- Issue tracking
- App
- API
- Monitoring
- Logging

References

[1] Docker, Inc. *Compose Deploy Specification*. n.d. URL: https://docs.docker.com/reference/compose-file/deploy/ (visited on 05/25/2025).

[2] Lauren Morley. *How to Create a DevOps Culture In Your Workplace*. 2025-05-20. URL: https://openmetal.io/resources/blog/how-to-create-a-devops-culture-in-your-workplace/ (visited on 05/27/2025).