DevOps Final Report

Group o

Ahmet Talha Akgül – ahma@itu.dk
Alexandra-Afrodita Ghizdareanu – algh@itu.dk
Andrada Condor – acon@itu.dk
Giorgi Mchedlishvili – gimc@itu.dk
Oscar William Kankanranta – oska@itu.dk

2025-05-30

1 Introduction

This report includes a description of the architecture and implementation details of the ITU-MiniTwit system developed as part of the DevOps course at the IT University of Copenhagen.

The ITU-MiniTwit project is a microblogging service designed as part of the DevOps course. It is a simplified version of Twitter where users can register, follow others, and post short messages ("tweets"). The project provides a foundation to apply and demonstrate modern DevOps practices such as continuous integration and delivery, containerization, automated testing, infrastructure as code, monitoring, scalability, and teamwork practices. In order to apply these practices and to help us test the robustness and scalability of the system, we have been provided a simulator that generates realistic user behavior.

Together, these sections aim to provide a comprehensive view of our development process and the resulting system.

Throughout the term, our team incrementally developed and improved the system, focusing on architecture, automation, observability, and maintainability. This report outlines the system design and structure, the process by which we brought ideas into production, and reflections on the challenges and lessons learned along the way.

2 System's Perspective

2.1 Architecture and Design

The ITU-MiniTwit system follows a modular client-server architecture designed to simulate the behavior of a micro-blogging platform. The system is composed of three main components:

- **Web Application:** The frontend interface allows users to register, log in, follow other users, and post messages. This component communicates exclusively with the backend API.
- **API Server:** The API layer, serves as the core backend of the application. It handles user authentication, message posting, follower relationships, and provides all necessary endpoints for both the frontend and simulator.
- **Database:** A PostgreSQL database stores persistent data including user accounts, messages, and social connections. The API server interacts with the database through an ORM instead of direct SQL queries.

This separation of concerns enables a clean and maintainable architecture. It also supports scalability and facilitates the inclusion of automated testing and simulation tools. The API-first design ensures that the system can be extended or consumed by other services independently of the frontend implementation.

A simulation module is also integrated, which emulates realistic user behavior and interacts with the API to test the system's functionality and performance under load. Figure 1 shows high level module view of how the systems interact. Simulator and Web application access API while API does all read/writes on the DB.

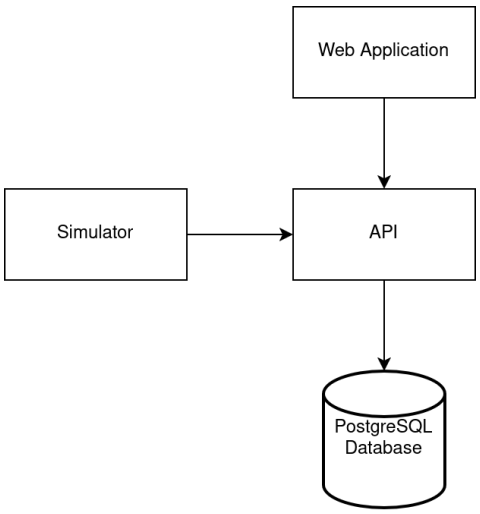

Figure 1: High-level architecture of the system

In our system, we also integrated monitoring tools. While details of which is explained in Section 3.2 in detail, Figure 2 shows how it is integrated into our system.

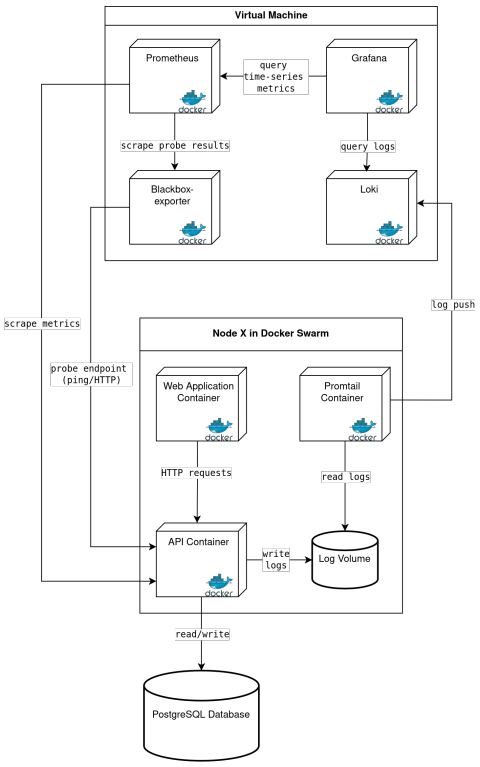

Figure 2: Monitoring and Logging Architecture

For container orchestration and deployment, we used Docker Swarm to manage the distribution and scaling of our services across multiple nodes. Our Swarm cluster consists of one manager node and two worker nodes, each of which runs containers for the web application, API, and Promtail for log collection. Logs are written to a local log volume on each node and collected by Promtail, which forwards them to Loki for centralized aggregation.

This deployment model ensures both high availability and horizontal scalability, as containers are distributed and can be replicated across nodes. The architecture also separates concerns clearly—allowing us to manage application services independently from monitoring components. The diagram below

illustrates the deployment structure across nodes in our cluster, highlighting the uniform setup of containers per node and the role of Promtail in the log pipeline.

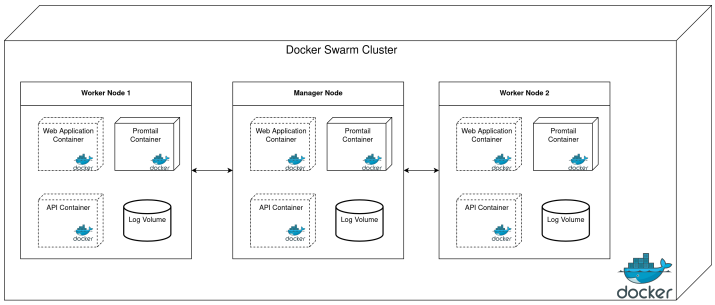

Figure 3: Deployment Diagram

The Figure 4 shows the high level system architecture with a focus on request routing. We deployed the application using Docker Swarm across three virtual machines: one manager and two worker nodes as it shown in Figure 3. We used NGINX as a reverse proxy on the manager node to route HTTPS requests to the appropriate services.

All external traffic for our domain is first routed through Cloudflare, which handles HTTPS termination and DNS-based routing to the correct virtual machine. From there, NGINX forwards requests to the correct service ports inside the Docker Swarm. Services communicate securely, and our PostgreSQL database is accessed using TCP/IP with SSL to ensure encrypted communication.

This architecture ensures scalability, service isolation, and a clear separation of concerns.

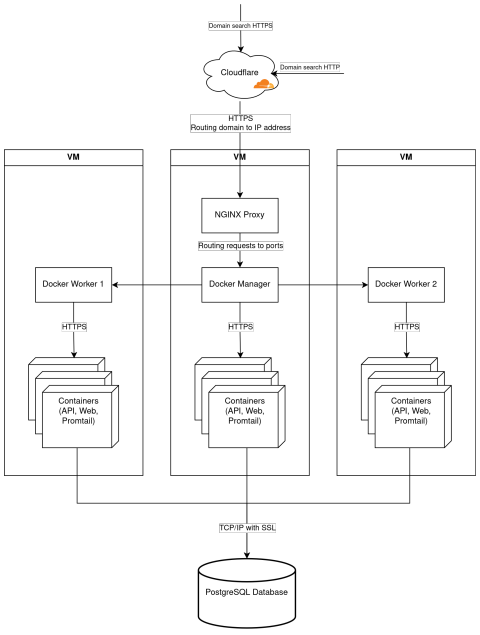

Figure 4: System Architecture Overview

2.2 Dependencies and Tooling

Frameworks

The application was refactored into Go to take advantage of its performance and built-in concurrency model. We mostly used Go's standard library, which provides robust support for HTTP servers and routing out of the box. For more advanced routing, we incorporated the routing library `Gorilla Mux`[1].

[1] `https://github.com/gorilla/mux`

Containers and orchestration
We used Docker to containerize our application components, ensuring consistency across development, testing, and production environments. As the project evolved, we adopted Docker Swarm for container orchestration. This allowed us to manage multi-container deployments, scale services easily, and perform rolling updates with minimal downtime.

Database
When refactoring away from SQLite, we migrated to a PostgreSQL database which is managed and hosted by DigitalOcean. This provided better scalability, reliability, and support for multiple connections. We used the `GORM`[2] library to introduce a Database abstraction layer so that we don't directly communicate with the database.

Others
We used a combination of Prometheus, Grafana, Loki, and Promtail. Prometheus was responsible for collecting metrics from our API. Grafana was used to visualize these metrics through their customizable dashboards. For centralized logging, we used Promtail to collect logs from our API and forward them to Loki, enabling us to query and visualize logs efficiently through Grafana's interface.

2.3 Subsystem Interactions

1st The user-flow (as shown in Figure 5) and simulator (as shown in Figure 6) show the interaction when trying to log in. There are other subsystem interactions, but we chose to only focus on the log in part, to not make the diagrams too big.

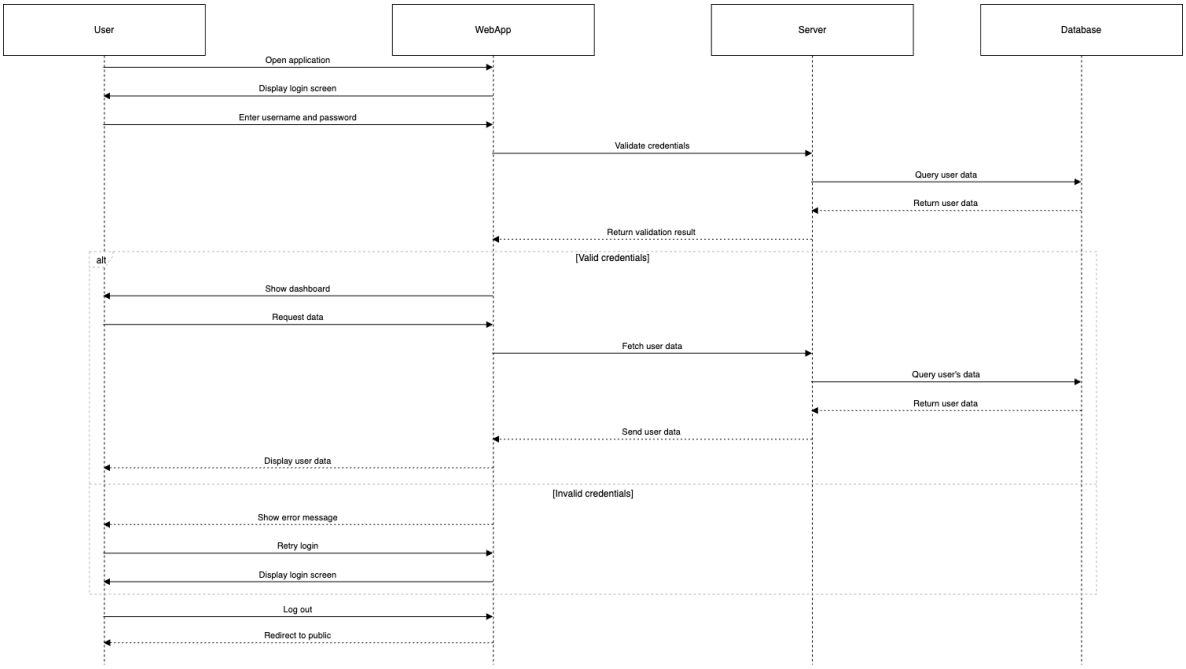

Figure 5: Sequence diagram for user interaction flow

2nd

[2] `https://gorm.io/`

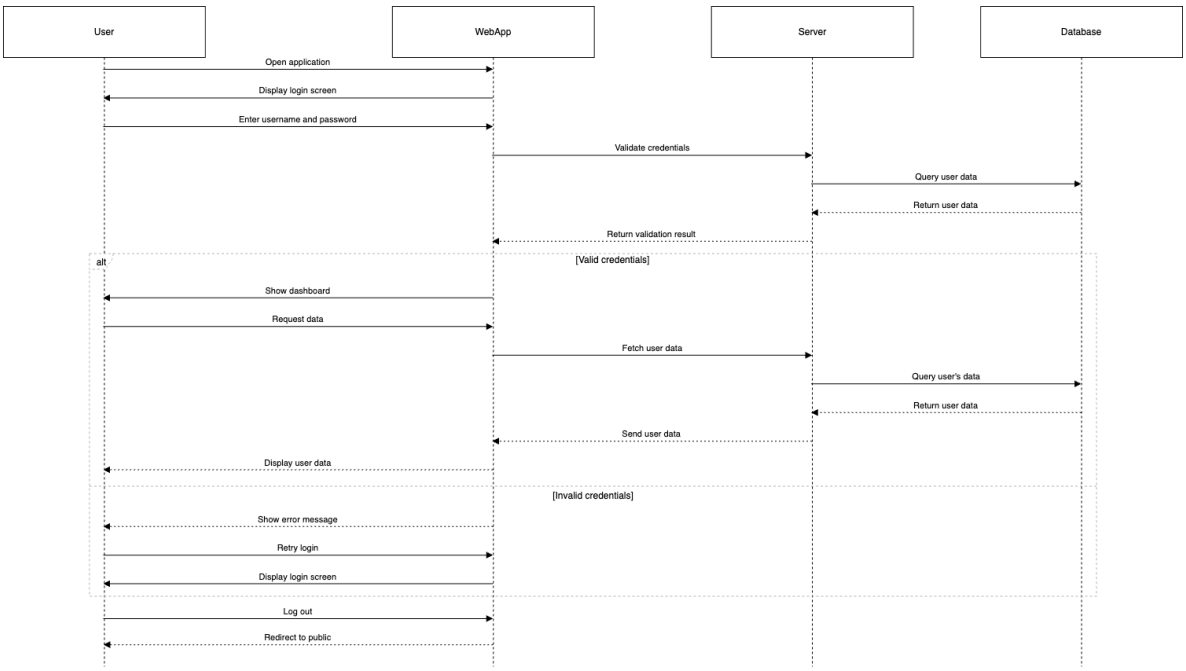

Figure 6: Sequence diagram for simulator

2.4 Static Analysis and System Quality

To ensure a maintainable and reliable system, we incorporated static analysis tools into our development workflow. These tools helped us improve code health and prevent technical debt accumulation.

We integrated three static analysis tools into our GitHub Actions CI/CD pipeline: **golangci-lint**, **Hadolint**, and **ShellCheck**. Each one of these tools targets a specific part of our codebase: Go code, Dockerfiles, and shell scripts respectively. The pipeline was configured in a dedicated workflow file, where each tool runs in a separate job.

For Go code, we used **golangci-lint** that runs multiple Go linters in parallel in a single configuration. This tool helped us uncover common issues such as unused imports or formatting violations. One notable finding involved detecting repeated logic and varaible declarations, especially in our test files. By refactoring the duplicated code into reusable functions, we improved both clarity and maintainability.

To ensure that our Dockerfiles follow the best practices we used **Hadolint**. This tools analyzes Dockerfiles for issues such as inefficient layers, missing CMD instructions or unsafe calls. To note is that Hadolint integrated ShellCheck internally, allowing it to lint shell commands embedded within Dockerfile RUN instructions.

While Hadolint covers shell code inside Dockerfiles, we used **ShellCheck** directly to lint external shell scripts in our repository. Integrating it in our pipeline ensured that all scripts were checked automatically before merging, catching quote issues and other unnecesarry subshells.

As a security measure, we integrated **GitGuardian**, a tool specifically designed to scan Git commits, branches, and CI pipelines for exposed secrets such as API keys, passwords, and other sensitive credentials. During testing, GitGuardian successfully identified an exposed a *secret_key* in one of our commits. Thanks to the immediate feedback from the pipeline, we were able to respond quickly by relocating the credentials to a dedicated *.env* file, reducing the risk of leakage.

Lastly, we used **SonarQube** to gain insights into code quality metrics such as code duplication, complexity and any potential bugs. Unlike linters that enforce static rules, SonarQube focuses on maintainability and test coverage. One specific recommendation involved complex expressions muliple times in the same function, suggesting simplifying them to improve readability and testability.

3 Process' perspective

3.1 CI/CD pipeline

Our CI/CD pipeline was built using GitHub Actions and followed a build, test, deployment, and release structure. The pipeline was triggered automatically on every push to the `main` branch and could also be run manually via the GitHub interface.

In the **build stage**, Docker images for the API, web frontend, and Promtail are built with Docker Buildx and pushed to Docker Hub, using caching to speed up builds.

During the **test stage**, we provision a test server over SSH, deploy the latest images, and run API (pytest) and UI (selenium) tests to catch regressions early.

The **deploy stage** The deploy stage connects to the production server and runs a script that checks for updated images and redeploys the stack using `docker stack deploy`.

We also added a **release stage**, where we created a new Git tag and published a release on GitHub, using the version stored in a tracked `VERSION` file.

3.2 Monitoring

For monitoring our system, we have decided on using **Prometheus**, **Grafana** and **Blackbox Exporter**. Our type of monitoring is a **pull-based whitebox** and **blackbox** monitoring. For implementation, we used a Prometheus client for our Go application, which scrapes data from our endpoints and exposed them in a specific endpoint designed for that
metrics

3.3 Logging

To aggregate logs across our infrastructure, we decided to use Promtail and Loki. We set up the Promtail service to run in a container, where it checks for updates on any logs files inside a volume that our API writes to. This is then sent to our central logging/monitoring server, where Loki ingests the data and stores it. The logs can then be accessed and visualized through Grafana which is hosted on the same server.

The logs are structured by the help of the Go Library Logrus [3], for easy parsing. The API logs: All database errors relating to storing, fetching or updating, and Invalid http requests.

The reason for using Promtail and Loki for log aggregation was because the ELK stack required too many resources. While trying to deploy the ELK stack to one of our VMs, it slowed down the whole server because it used 100% of CPU when ElasticSerach was initializing. We then got recommended by another team to use Promtail and Loki, which proved to be great for our infrastructure, as Loki seamlessly integrates into Grafana, which we already used for monitoring.

3.4 Security assessment

We conducted a security assessment focusing on key components of our system, including the API, web application, infrastructure, and CI/CD pipeline. After the security assessment we figured out some problems and solved most of them. The following table outlines identified risks and the corresponding mitigation strategies implemented or planned to implement (Red ones shows they are still planned to implement):

[3] `https://github.com/sirupsen/logrus`

Asset/Area	Identified Risk	Mitigation Strategy
API Endpoints	SQL Injection via unsanitized inputs	Input validation and parameterized queries
Login System	Brute-force attacks on authentication	Rate limiting middleware on login endpoints
Docker Containers	Privilege escalation within containers	Containers run as non-root users
Secrets Management	Exposure of sensitive credentials	GitHub Secrets and environment variables used
SSH Access	Unauthorized server access	SSH key authentication; password login disabled
CI/CD Pipeline	Leakage of secrets during deployment	Encrypted GitHub Actions secrets; secure SSH deploy
Logging Mechanism	Sensitive data exposure in logs	Log filtering to exclude confidential data
External Traffic	Eavesdropping / MITM attacks	TLS termination via Cloudflare; HTTPS enforced through NGINX

Table 1: Summary of Security Risks and Mitigations

3.5 Scaling

In this project, most of our scaling and upgrades happened in our CI/CD pipeline. After the simulator was started, we needed a way to deploy code without having downtime. In the start, we adopted a blue/green deployment strategy, this was made possible by setting up a reverse proxy[4] for the incoming requests. When deploying new code, a new container was spawned with the changes, which we then redirected the requests to. This provided virtually zero-downtime, as it was only Ngnix's configuration that needed to be reloaded[5].

Further along in the project, we adopted a rolling update strategy using Docker Swarm. Our CI/CD pipeline uploaded new Docker images to Docker Hub, and a deployment script on the server checked for image updates. If changes were detected, docker stack deploy was triggered, which allowed Swarm to update services one task at a time. This ensured zero-downtime deployments by gradually replacing containers while keeping the application available.

Initially, we used Vagrant to manually provision droplets for our Docker Swarm cluster and test servers. Later in the project, we transitioned to Terraform to automate infrastructure setup. We successfully automated droplet creation and connecting the Swarm cluster. While we didn't fully automate starting the application, it highlighted Terraform's power and potential in managing IaC.

3.6 AI assistant

The project do not contain any AI assistants in the pipeline. But we have made use of LLMs throughout the project for different tasks. They have been a great resource to help understand, and help with various tasks. Especially in the beginning in the refactor stage, where the team had limited knowledge of Go. The AI assistants were able to explain and produce code that we weren't familiar with in the start.
AI assistants are however not the best when it comes to infrastructure. Often it would produce outdated or just wrong information. It was able to produce okay results when you prompted it with

[4] `https://nginx.org/`
[5] `https://immersedincode.io.vn/blog/zero-downtime-deployment-with-docker-compose-nginx/`

a specific task that you provided context to. If you understood the concept, and had some knowledge about the subject, you could use it to produce IaC.

4 Reflection Perspective

4.1 Evolution and Refactoring

Go was an entirely new language for most of the group, which created some difficulties in the refactoring part in the beginning. It mostly because we required some time to understand the libraries and syntax. We also made a mistake early on, which was that we made the user facing functions access the database directly, instead of going through the functions in the API. This mean that our separation refactoring more extensive.

4.2 Operation

Some of the major challenges we faced were upgrading to a deployment strategy and migrating to a new database. We wanted to do this while ensuring *zero-downtime* and *no data loss*.

Deployment Strategy

Initially, our database was stored inside the docker container file system, meaning that any re-deployment or container recreation would result in data loss. This setup made it incompatible with strategies like blue/green deployment, where multiple containers might be started and stopped dynamically. Our first Docker Compose setup did not attach a volume to persist data[6]. To resolve this, we manually copied the SQLite database from the container to a mounted volume, ensuring data persistence across container restarts. The updated Docker Compose file reflects this fix[7].

To implement blue/green deployment, we set up NGINX as a reverse proxy, listening on port 5001 and forwarding traffic to the active API container. However, since NGINX can't bind to a port already in use, we wrote a shell script to stop the existing container on port 5001, reload NGINX, and begin redirecting traffic to the new container immediately. This minimized downtime to just the time it took for NGINX to reload, which was effectively near-zero.

Database Migration

As part of system upgrades, we transitioned from SQLite to a managed PostgreSQL database hosted on DigitalOcean. To avoid downtime and data inconsistency, we developed a staged migration process:

- **Initial Copy:** We copied the SQLite database (25 MB) from the production server to a secondary VM and used a Python script to insert the data into PostgreSQL. This process took nearly an hour.
- **Second Copy:** During the initial transfer, new data continued to be written to the production SQLite database. To handle this, we used `sqldiff` to generate a differential SQLite database containing only the changes since the first copy.
- **Final Copy and Switch:** We copied one extra time, applied the remaining data to PostgreSQL, and then used a shell script to reload NGINX to point to the version connected to the new database.

This approach allowed us to complete the database migration with almost no data loss and near-zero service interruption.

[6] `https://github.com/Niceness-2-0/itu-minitwit/commit/d97e6233156af6ce7d8cac76d87aef19836021a2`
[7] `https://github.com/Niceness-2-0/itu-minitwit/commit/f00d91373c2a3b79bf03663b2cee3ad58113c219`

4.3 Maintenance

Most of our performance issues can be related to the droplets on digital ocean. All of our VMs were from the cheapest tier, which do not have a lot of computing power. This was most apparent when we needed to start a service that was a bit more resource heavy. For example, when we tried to initialize the ELK stack, ElasticSearch took all the resources and used 100% of the CPU processes, basically making the VM useless until you restart it. The same happened on our Grafana server, where periodically, the CPU usage would skyrocket to 100% until we shut down the service. This made up upscale to a bit more expensive tier on the logging server.

On our production server, we didn't see much issue, however we did receive around 30.000 Read Time Out Requests which was reported by the simulator. We are not entirely sure what caused this, but we are speculating it could either be a slow database connection, or that the server didn't have enough performance.

5 Concluding Remarks

In this project, we followed a DevOps type approach. This meant we weren't just writing code, but also constantly thinking on how the system was built, deployed, and maintained. Compared to previous projects, we focused more on automating the work flow from code to production with CI/CD pipelines, instead of having it as a back thought.