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.

Together, these following 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 different aspects in development.

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. It is structured using a modular design, separating concerns into distinct packages such as handlers, dto, models, and others to improve maintainability and readability.
- **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.

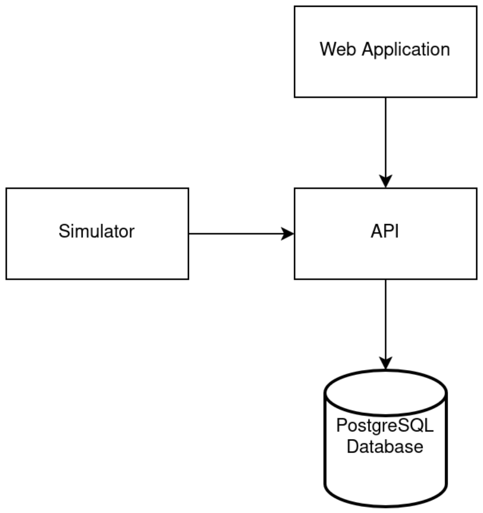

Figure 1: High-level architecture of the system

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. Figure 2 illustrates the deployment structure across nodes in our cluster, highlighting the uniform setup of containers per node. Dotted lines means that those containers do not necessarily have to work in those nodes, it depends on how many replicas we have. For instance, if we have 1 replica for API, then API container will work only one of the nodes.

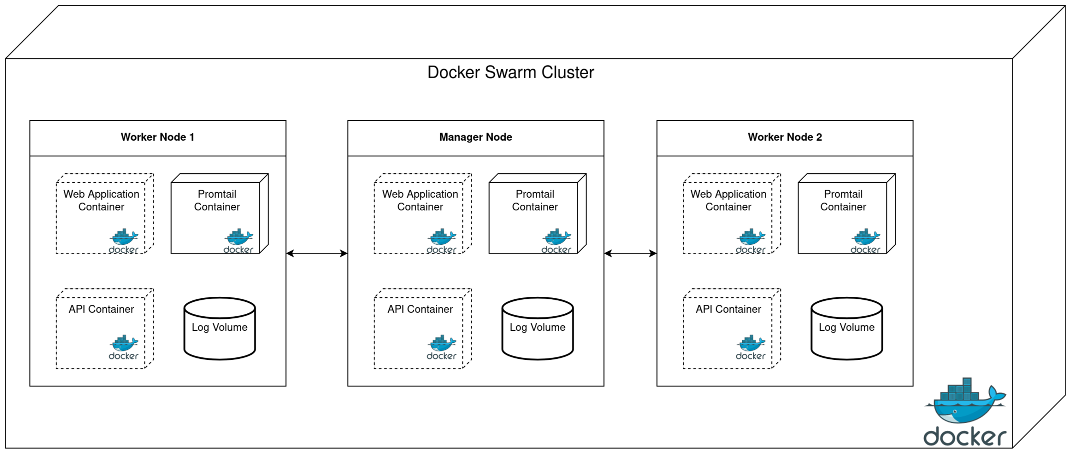

Figure 2: Deployment Diagram

In our system, we also integrated monitoring tools. While details of which is explained in Section 3.2 in detail, Figure 3 shows how it is integrated into our system.

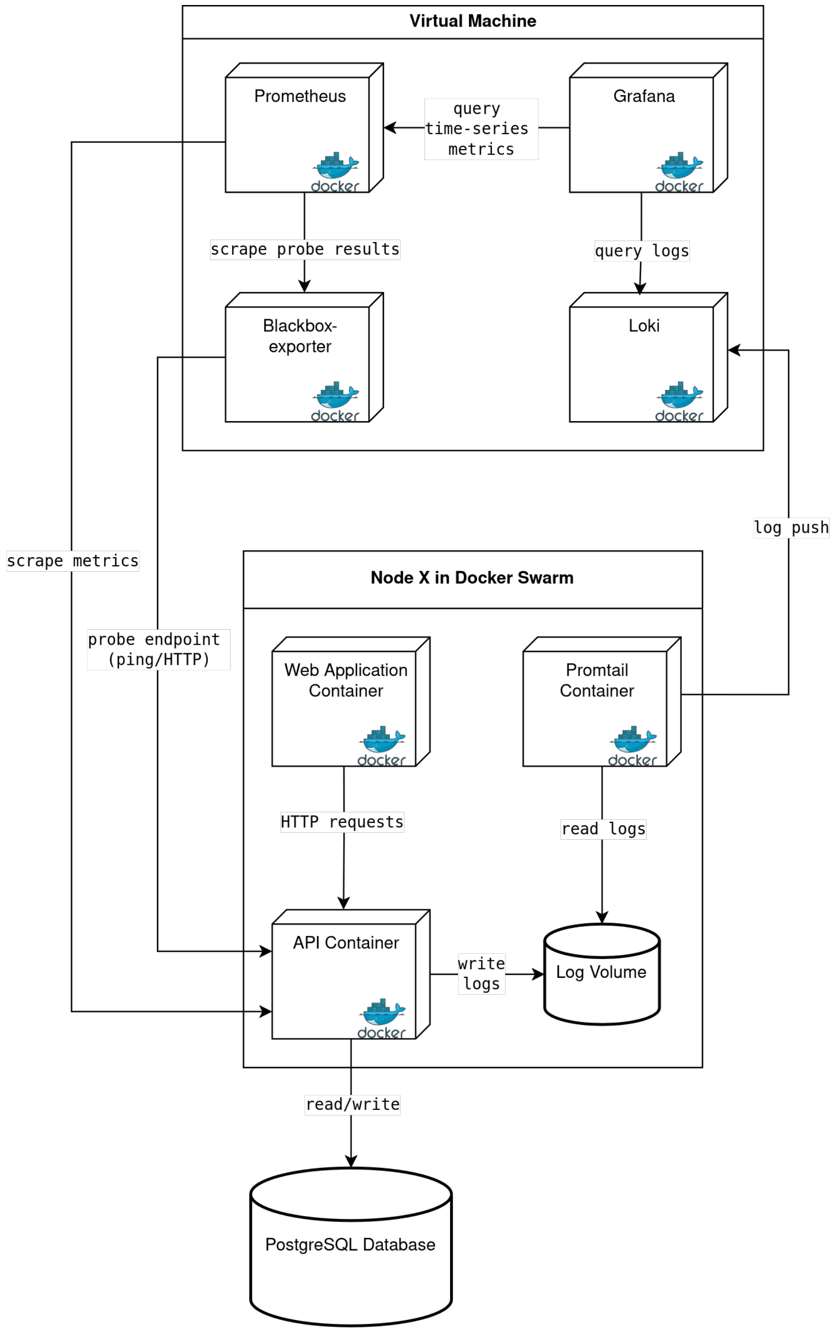

Figure 3: Monitoring and Logging Architecture

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 2. 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.

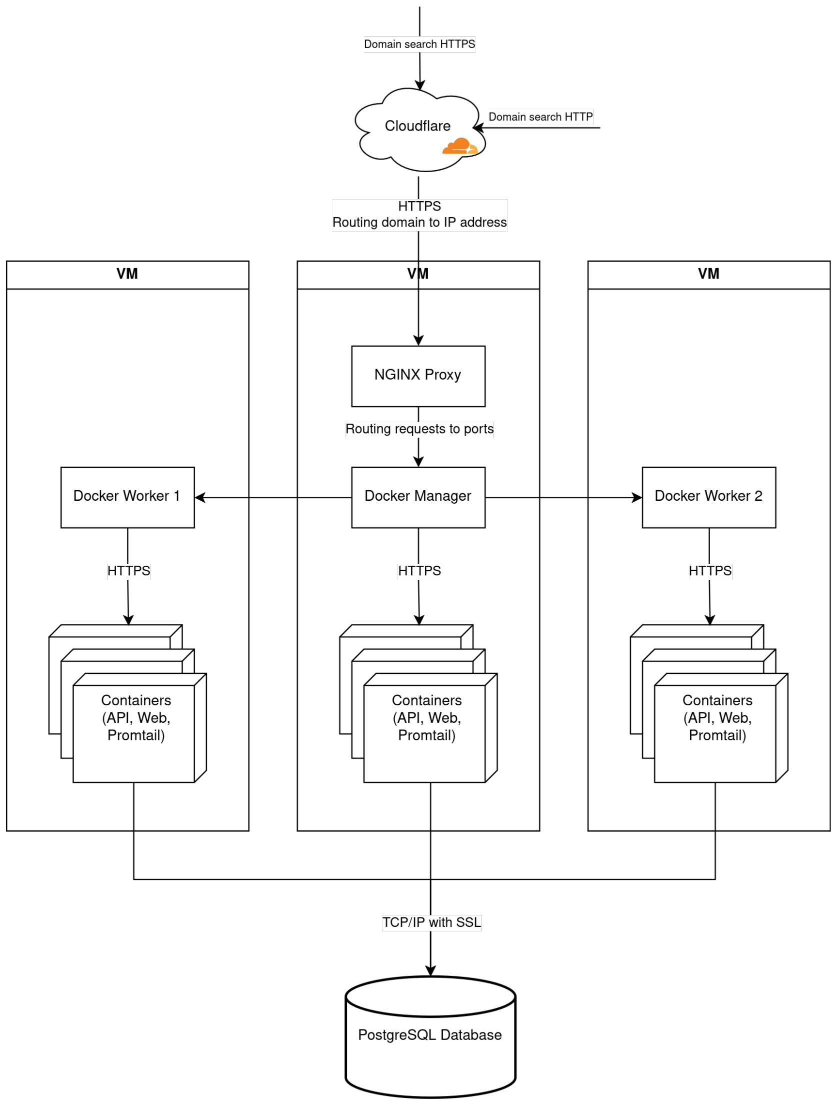

Figure 4: System Architecture Overview

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]. We chose Go over other languages due to its fast, ease-of-use, and security[2], making it well-suited for a project focused on DevOps and scalability.

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

[1]`https://github.com/gorilla/mux`
[2]`https://www.orientsoftware.com/blog/golang-performance/`

hosted by DigitalOcean. This provided better scalability, reliability, and support for multiple connections. `GORM`[3] library to introduce a Database abstraction layer so that we don't directly communicate with the database.

We decided to use PostgreSQL because of its ease-of-use and that it is well suited for applications that requires a lot of read and writes[4].

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

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.

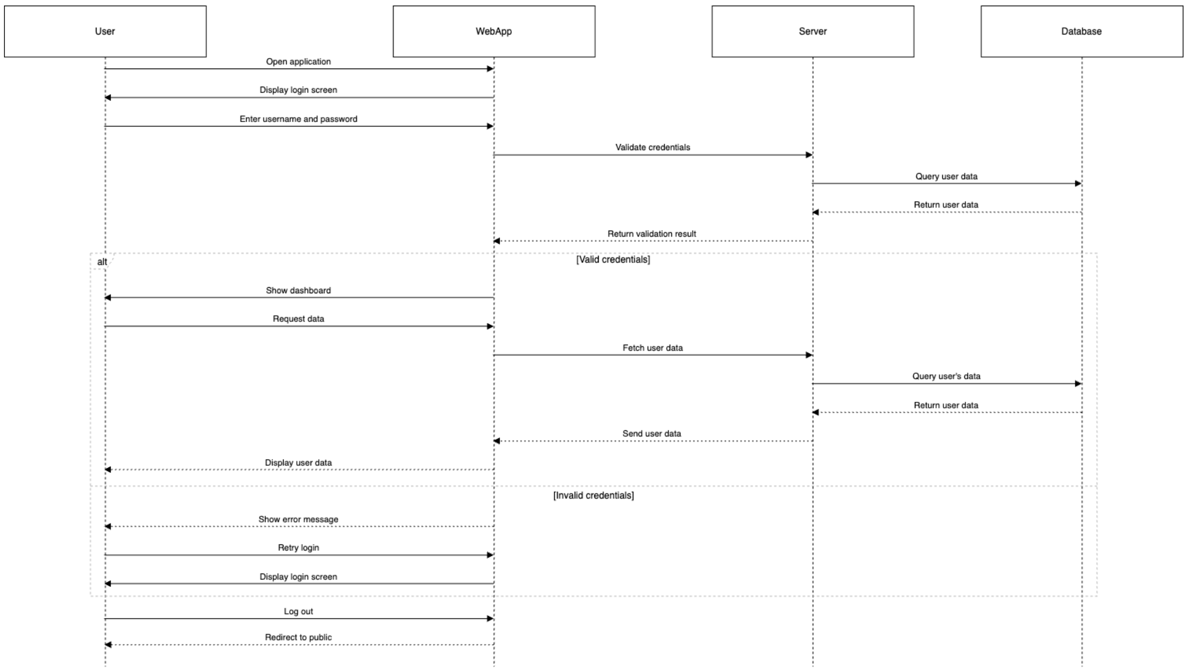

Figure 5: Sequence diagram for user interaction flow

[3]`https://gorm.io/`
[4]`https://cleancommit.io/blog/why-use-postgresql-for-your-next-project/`

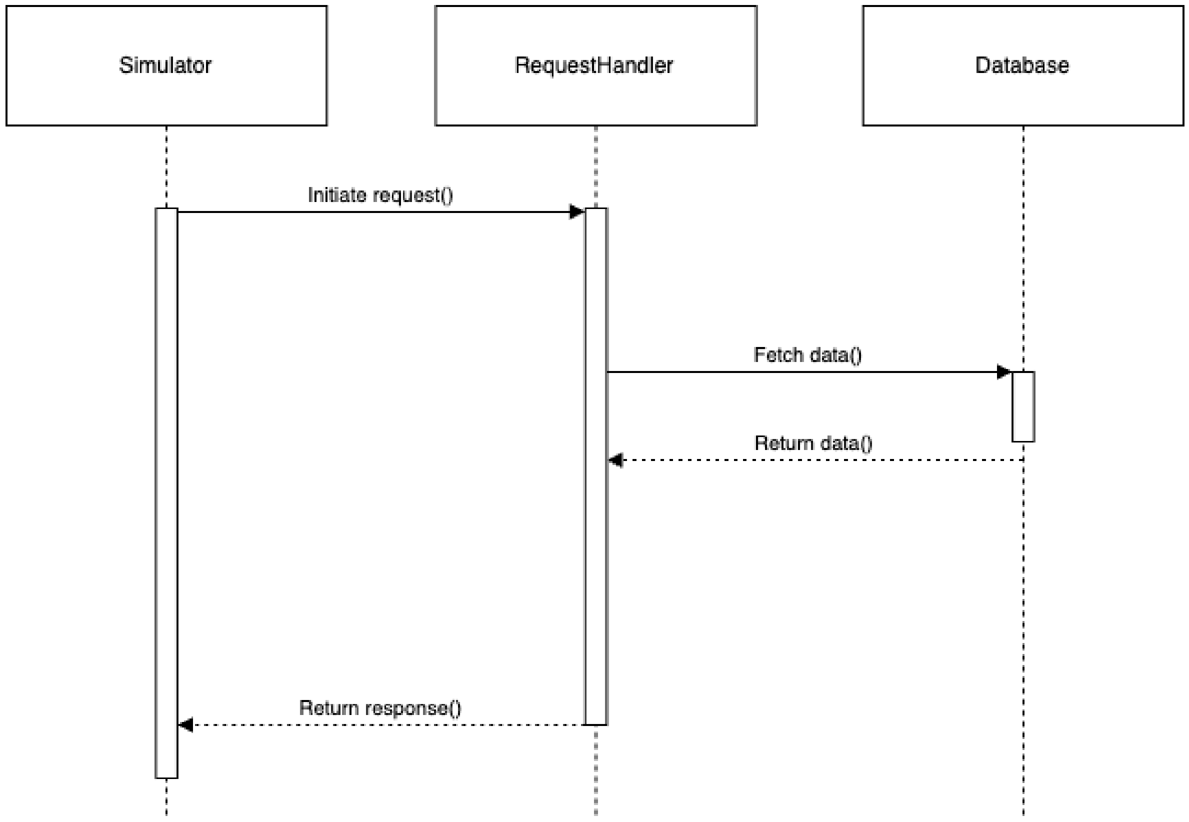

Figure 6: Sequence diagram for simulator

2.4 Static Analysis and System Quality

To ensure a maintainable and reliable system, we incorporated static analysis tools to 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.

For Go code, we used **golangci-lint**, which helped us uncover common issues such as unused imports or formatting violations. One notable finding involved detecting repeated logic and variable declarations, especially in our test files.

To ensure that our Dockerfiles follow the best practices we used **Hadolint**. This tool analyzes Dockerfiles for issues such as inefficient layers, missing CMD instructions or unsafe calls.

While Hadolint covers shell code inside Dockerfiles, we used **ShellCheck** to lint external shell scripts. Integrating it in our pipeline ensured that all scripts were checked automatically before merging, catching quote issues and other unnecessary 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. Unlike linters that enforce static rules, SonarQube focuses on maintainability and test coverage. One specific recommendation involved complex expressions muliple times, 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 automatically triggered on every push to the `main` branch and could also be run manually via the GitHub interface.

We chose GitHub Actions because it offers a seamless integration with our GitHub repository, requires no external CI/CD tooling, and provides a flexible YAML-based syntax that allowed us to define complex workflows with minimal setup. Additionally, GitHub Actions supports secure secret management, reusable actions, and tight permission control and managing sensitive credentials in our workflow.

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** 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, we chose Prometheus[5] and Grafana[6] for their complementary strengths. Prometheus handles pull-based whitebox metric collection, Grafana provides intuitive dashboards for visualization, and the Blackbox Exporter enables external uptime checks for effective blackbox monitoring. All are open-source, free, and widely adopted technologies with extensive documentation and libraries suited to our Go codebase. They are easy to integrate, making them a great fit for our project and providing a complete system overview.

We used a Prometheus client[7] in our Go application, exposing metrics via the dedicated **/metrics** endpoint. Prometheus scrapes this endpoint, collecting data such as request counts, response status codes, and endpoint-specific performance metrics. Blackbox Exporter performs external HTTP probes to verify service availability and responsiveness from the outside.

Grafana is used to visualize these metrics using a custom dashboard, enabling us to track both system-level behavior and user interaction. We monitor functional metrics such as route-specific request volumes, status code, and latency, but we try to gain business-relevant data insights, like total registrations, messages posted. Using this setup we have visibility on the system's health and usage, allowing us to detect anomalies early and assess the system's effectiveness in meeting user needs.

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 and stores the data. 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 [8], 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 that the ELK stack required too many resources. While trying to deploy the ELK stack to one of our VMs, it slowed down the whole

[5] `https://prometheus.io/`

[6] `https://grafana.com/`

[7] `https://github.com/prometheus/client_golang`

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

server because it used 100% of the 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 we implemented or planned to implement (Red ones show they are still planned to be implemented):

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[9] 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[10].

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. The reason for choosing docker swarm, is because it's better suited for the small application we have. We looked into how to set up Kubernetes, but we agreed that it would be an overkill, and would probably take too long to implement.

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

Initially, we used Vagrant to 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 Infrastructure as Code(IaC). We opted for **Terraform**[11], an open-source tool by HashiCorp, because it allows us to define and manage infrastructure using a simple, declarative language. It works across different cloud providers, making it highly flexible. This helps keep setups consistent, easy to reproduce, and much easier to share or collaborate on.

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.

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 means 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 redeployment 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[12]. 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[13].

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.

[11]`https://developer.hashicorp.com/terraform`
[12]`https://github.com/Niceness-2-0/itu-minitwit/commit/d97e6233156af6ce7d8cac76d87aef19836021a2`
[13]`https://github.com/Niceness-2-0/itu-minitwit/commit/f00d91373c2a3b79bf03663b2cee3ad58113c219`

- **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.

4.3 Maintenance

Most of our performance issues came from using the smallest DigitalOcean droplets, which offered limited computing power. Resource heavy services like the ELK stack quickly maxed out CPU usage, ElasticSearch alone would push CPU to 100%, rendering the VM unresponsive until restarted. We observed similar spikes on the Grafana server. As a result, we upgraded the logging server to a more expensive VM for better performance.

While our production server ran mostly without issues, the simulator reported around 30,000 read timeout requests. The cause is unclear, but we suspect either slow database connections or insufficient server 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. We also prioritized creating scripts and IaC files to automate the process, instead of having to manually create servers through the GUI.

6 Links

- GitHub repository: `https://github.com/Niceness-2-0/itu-minitwit`
- Web: `https://minitwit.anemotech.com`
- API: `https://minitwit-api.anemotech.com/latest`
- Monitoring and Logging service(Grafana): `http://209.38.43.0:3000/`
- Prometheus: `http://209.38.43.0:9090/query`
- Terraform visualizer: `http://157.245.21.149:8888/`