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 28, 2025

Contents

1 System's perspective

Our MiniTwit is an 'X' (formerly known as Twitter) clone written in Golang using the Gorilla web framework. It is a project that continues development on the ITU-MiniTwit system presented in the course DevOps, and it consists of a web service and an API service.

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 are 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 4). Figure 1 is a Specification Level Deployment Diagram, which shows how ITU-MiniTwit is deployed on a worker-node.

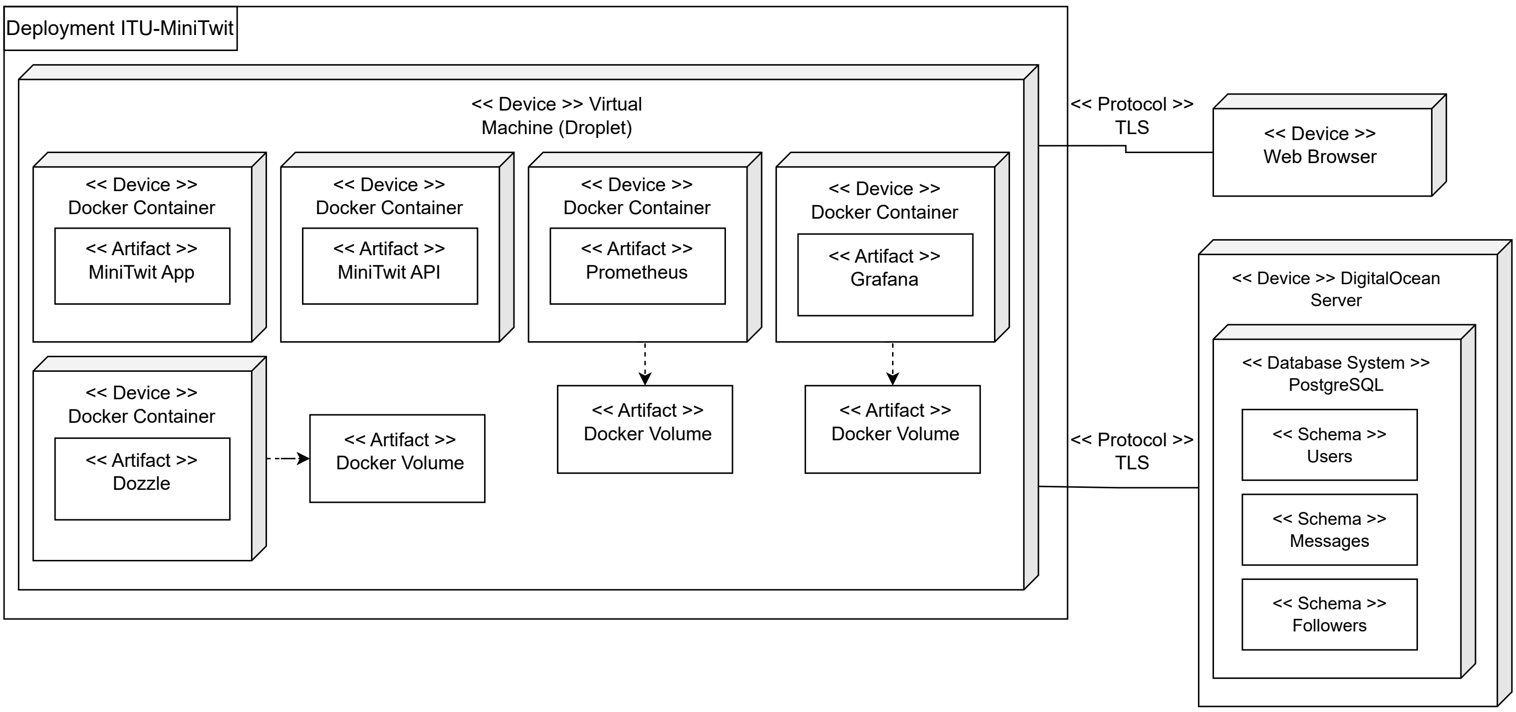

Figure 1: Deployment Diagram of MiniTwit on a worker node

1.2 Dependencies of MiniTwit

This project relies on several dependencies across different project areas, including runtime, testing, and infrastructure.

App and API

Our app and API are built using Golang. It introduces the following dependencies:

Testing

Infrastructure

- To maintain a swarm
- Postgres database

1.3 System Interactions

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 2)

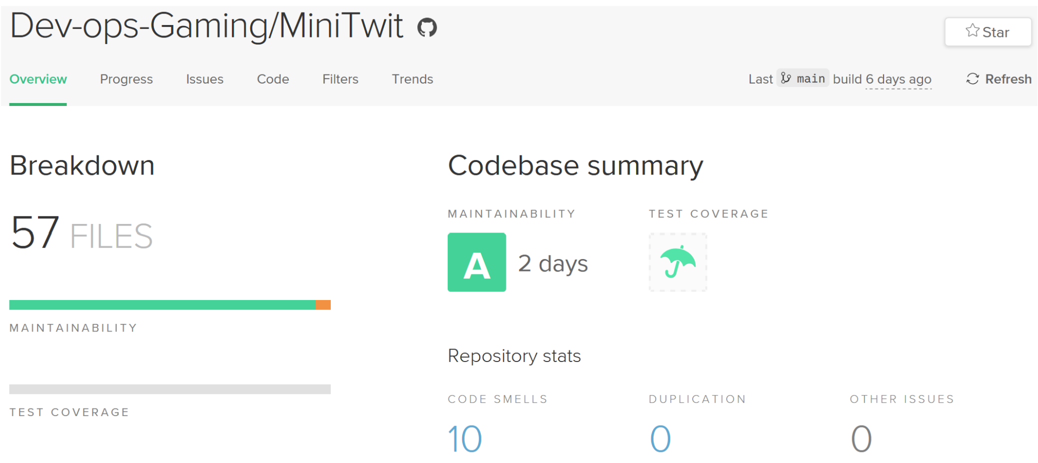

Figure 2: Overview of MiniTwit on Code Climate

Sonarqube

Sonarqube reports 18 issues - all in the category 'maintainability' - and 0.9% code duplication (see figure 3). A majority of these issues are found in the tests, and many of the issues are in regards to string duplicates.

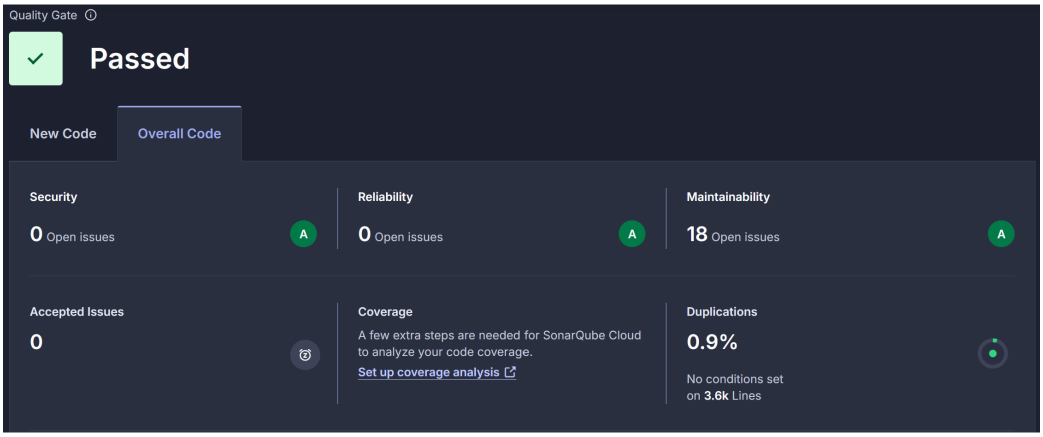

Figure 3: FIX POSITION OF THIS WHEN REPORT ALMOST DONE!! Overview of MiniTwit on SonarQube

2 Process' perspective

2.1 Security Assessment

To protect our system against adversaries, we first identify the assets that must be protected. We then identify which threats we could face and how. Finally, we analyze these scenarios to figure out which scenarios pose the biggest threats.

Risk Identification

Our assets:

- Codebase
- Database
- Logs

Threat Sources:

- SQL Injection
- Exposed ports (Prometheus)

- Vulnerable password hashing
- No requirements for user password (length, symbols, etc.)
- Vulnerable dependencies
- No password on Dozzle (logging)

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 uses the open ports to overload our Prometheus with queries
- F - Attacker intersects messages sent over the network, as they are not encrypted
- G - Attacker exploits a vulnerable dependency, which lets them perform some malicious attack
- H - Attacker accesses our logs on Dozzle, which lets them see all outputs, like usernames

Risk Analysis

Here are the scenarios presented on the risk matrix:

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

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.
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 would have to perform a workaround, as the exposed ports are caused by Docker and UFW being incompatible, an issue which has not yet been patched. However, Prometheus is the only service that is exposed, and the only threat is Prometheus crashing due to an overload of queries. Therefore, it is not considered a high-priority threat.

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, it then pulls them onto 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.

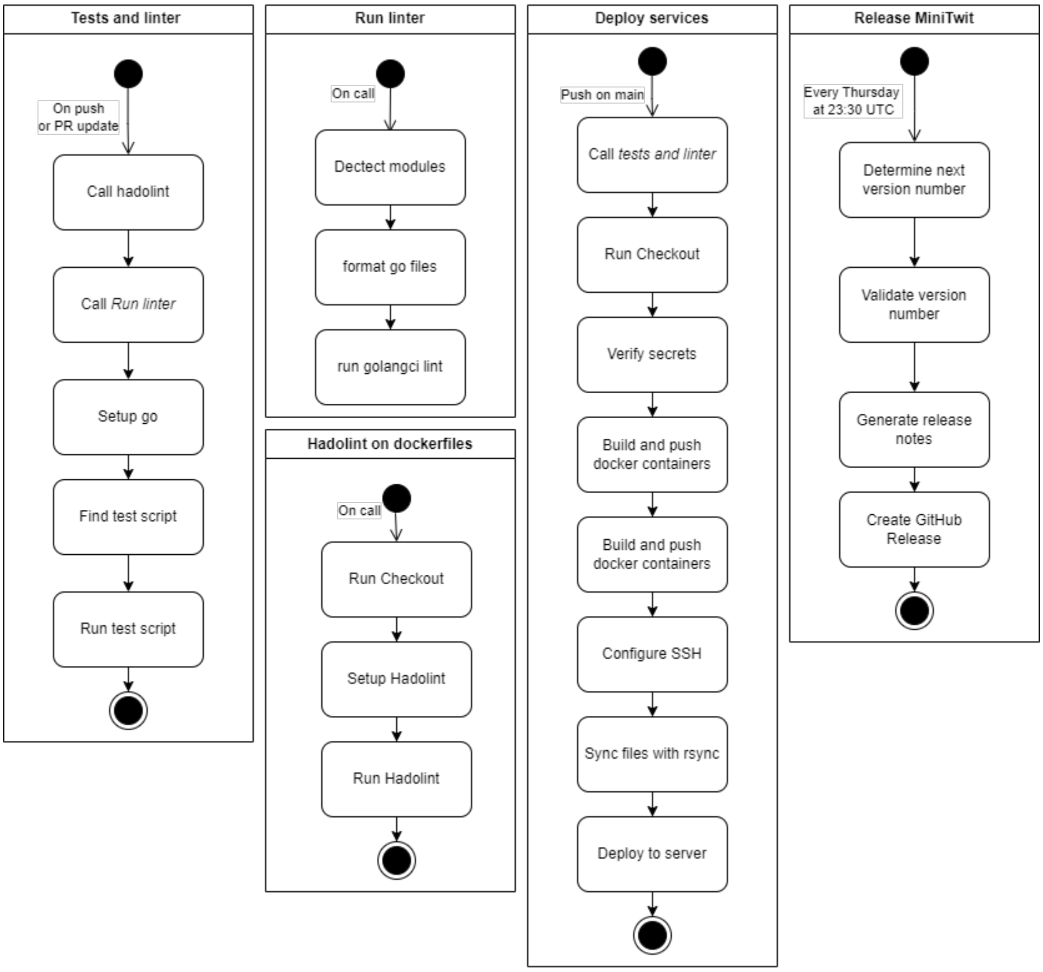

Figure 4: Activity diagram of the workflows.

2.4 Monitoring

We monitored MiniTwit using Prometheus for collecting metrics and Grafana for visualizing them. This setup provided great insights into the application's performance and behavior.

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

Althoughugh Prometheus was important for collecting data, Grafana is utilizto 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 two dashboard up; one for the app and one for the API.

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.

2.5 Logging

We implemented logging in MiniTwit using Dozzle. We were recommended to implement an ELK or ELFK stack, but for our group, Dozzle was a better lightweight alternative; I will explain why this is the case later on.

Dozzle is a lightweight web-based log viewer for Docker containers that also easily supports Docker Swarm. When we first wanted to implement logging, we implemented ELK in our standard Docker Compose. This was a very basic implementation with no parsing or user setup, so naturally, the logs were almost unreadable. This never made it to production as we had several people working on it; while being worked on, we implemented Docker Swarm, and suddenly, a lot changed for our system. We switched to Docker Stack instead of Docker Compose, so changes were required. Meanwhile, another group member saw 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?

How do we aggregate logs?

Why we choose Dozzle

1. **Instant, real-time streaming:** Dozzle shows every stdout/stderr line as it happens, with no batching or indexing delay. When we create new replicas (e.g. our three `app` or four `api` containers), their logs show up live in the UI.

2. **Filtering by service with Docker filters:** Dozzle respects Docker's own `--filter` flags (or the `DOZZLE_FILTER` env var), which means that we can, for example, show only logs from containers labeled `service=api` or that are running on a particular node.

3. **Fuzzy sidebar search:** The sidebar shows all the containers in the minitwit-network overlay. If you type "api," it will immediately narrow it down to your four api replicas. You will not have to scroll through a lot of IDs anymore. You also have the possibility of checking your logs for each individual service on the sidebar.

4. **Ad-hoc text & regex search:** after you have chosen a container's live tail, you can use the built-in search box to highlight plain text or full regex (e.g. `/ERROR|WARN/`) so that during a rolling update, you only see the lines that matter to you.

5. **No extra infrastructure:** Dozzle is just another container on the overlay network (deployed in global mode). There's no Elasticsearch cluster to manage, no Logstash pipelines to configure, and no Kibana dashboards to configure.

6. **Minimal resource footprint:** It does not index or save logs, so CPU/memory stays low even under heavy traffic. That means that instead of a full log

store, our DigitalOcean droplets will have more space for the `app`, `api`, and Postgres services.

7. **Native support for Docker Swarm:** Running in global mode on every node, Dozzle auto-discovers new replicas when you scale via `docker stack deploy`, no config changes necessary as your `deploy.replicas` grow or shrink.

8. **Simple UI:** One page: scroll to the bottom, search for, and download plain text. No need to learn JSON query language or build a dashboard. Great for quick debugging.

ELK stack features we missed out on:

1. **Long-term retention & archiving:** Dozzle only shows live container logs. You would need to construct something like an ELK stack or manually save data.

2. **Structured indexing & fielded queries:** ELK is capable of parsing JSON (timestamps, user-IDs, and request-IDs) and turning them into queryable data, such as "Find all 5xx from client X in the last hour." On parsed fields, Dozzle's free-text/regex search is incompatible.

3. **Role-based access control:** Kibana offers users, teams, read-only views, and dashboard-level permissions. Dozzle has no auth at all.

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 a lot more management, and you could almost endlessly improve it. We feel that Dozzle provided us with what we needed. With our monitoring, we have the opportunity as developers to quickly use the free search on Dozzle to find our error.

3 Reflection Perspective

3.1 Evolution and Refactoring

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 that 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 then 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.

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 completely 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.

4 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., the database index issue described in section 3.2 or a tutor 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.

References

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