# Test1 - Group Q

DevOps, Software Evolution and Software Maintenance

KSDSESM1KU

IT-University of Copenhagen

May 23, 2024


| | |
|---|---|
| Carl Bruun | carbr@itu.dk |
| Christian Stender | ches@itu.dk |
| Nadja Brix Koch | nako@itu.dk |
| Theis Helth Stensgaard | thhs@itu.dk |

# 1 System's Perspective

## 1.1 Design and Architecture

Our ITU-MiniTwit systems consists of a web service and an API service, both written in Golang, which act in a client-server architecture. Data is stored in a PostgreSQL database.

Her kommer der et fucking diagram

The UML deployment diagram in figure **??** shows how the different components in our system is deployed. All technologies and components are described in detail in the following section.

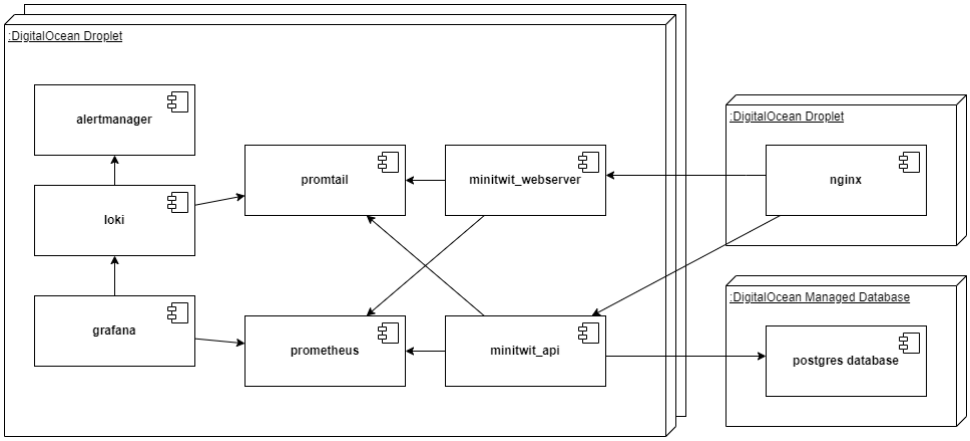


Figure 1: UML Deployment Diagram

## 1.2 Technologies and tools

This sections provide a overview and description of the technologies and tools utilized throughout our project work.

### List of Technologies and Tools Utilized

- **Go**: A statically typed, compiled programming often used for backend development.
- **Go/Gorilla**: A toolkit for Golang, that provides packages for building web applications. Used for routing and session management.
- **SQLite**: A lightweight SQL database engine that is ideal for embedded database applications. Used in initial setup and for testing.
- **Docker**: A platform for shipping and running applications, ensuring consistency across different environments. Containerization is used deliberately throughout our system and in delivery.
- **Docker Swarm**: A container orchestration tool, included in Docker by default, to manage a cluster of Docker nodes as a single virtual system.
- **DigitalOcean**: A cloud infrastructure provider offering scalable compute and storage solutions. Used for deploying and managing the application and database.
- **Prometheus**: An open-source monitoring toolkit used for monitoring metrics in our cloud environments.
- **Grafana**: An open-source analytics and monitoring platform that integrates various data sources to visualize our data and metrics.

- **PostgreSQL**: A open-source object-relational database system, known for extensibility and speed. Used after the transition from SQLite managed by DigitalOcean.
- **Promtail**: An agent that ships log file to Loki and part of the Grafana logging stack. Used for gathering logs.
- **Loki**: A log aggregation system designed to be scalable. Used with Grafana for log querying and visualization.
- **Nginx**: A high-performance web server and reverse proxy server. Used for load balancing.
- **CertBot**: A open-source tool for automatically using Let's Encrypt certificates to enable HTTPS on web servers.
- **Terraform**: An infrastrcucture as code tool that allows users to define and provision data center infrastructure using a high-level configuration language.

**Technology choices**

Our initial choice of language for refactoring ITU-MiniTwit was based on a detailed feature mapping of the system and a comparison of programming languages (see Appendix **??**).

This analysis led us to initially select Crystal/Kemal. However, we soon discovered that Kemal's documentation was insufficient, making it challenging to work with. Consequently, we switched to Golang, which offered similar features but had much more comprehensive documentation.

## 1.3 Subsystem interactions

Figure **??** shows the flow of information seen from the perspective of a user accessing the website. Initially the user reaches the load balancer, nginx, connecting and forwarding the request to one of the webserver replicas that then looks back to nginx with a request for a backend api. When the backend receives the request it sends the query for recent messages to the database as well as logs and updates monitoring metrics before returning the response to nginx and then to the webserver. Ultimately the webserver serves the final page to the user through nginx using the messages receives from the backend.

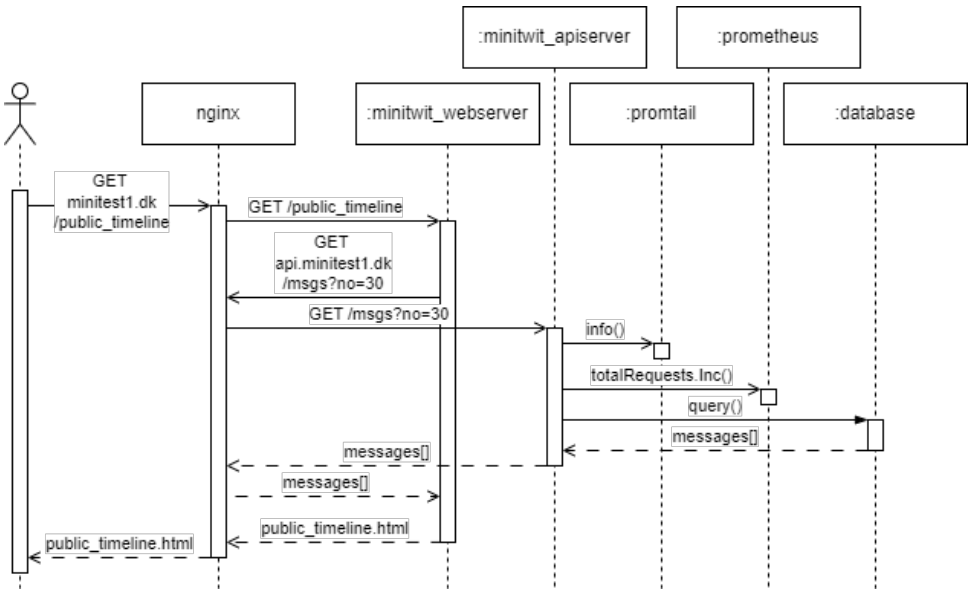


Figure 2: Sequence diagram of the user interacting with the system.

The simulator interaction is very similar to the interaction of the user except for the fact that it does not need to go through the webserver. The flow of a POST request can be seen in the sequence diagram in figure **??**. One thing to note is that we have not been stress testing the user perspective in this course as all requests have come from the simulator. This means that we do not have data to reflect on the performance of our nginx setup which results in a lot of back and forth messaging as seen in the first diagram.

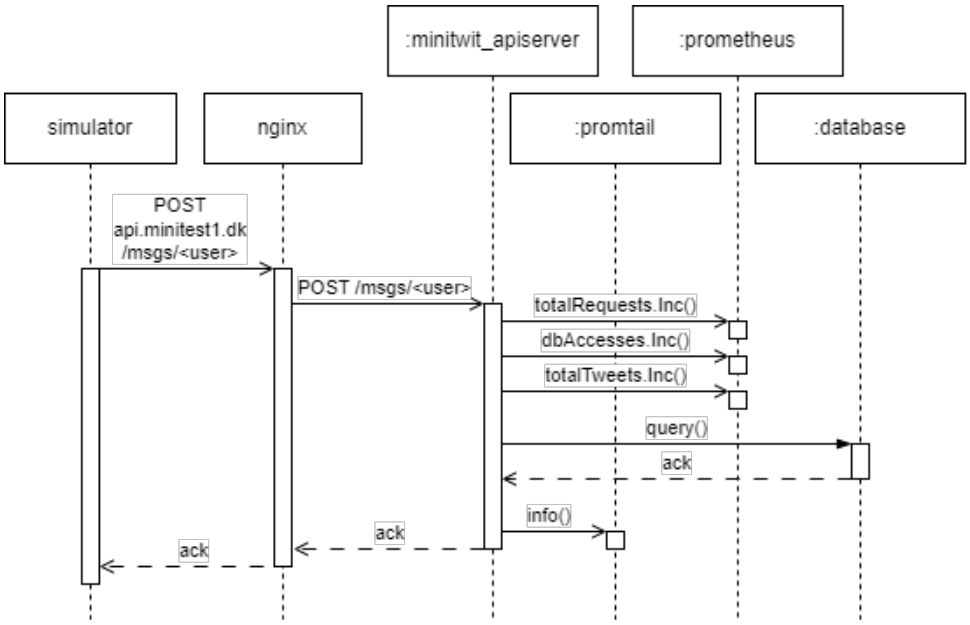


Figure 3: Sequence diagram of the simulator interacting with the system.

## 1.4 Current state

At the time of writing, our ITU-MiniTwit system is being evaluated using the static code analysis tools, Code Climate and SonarCloud. For an overview, see Appendix **??**. In summary, both tools reveal that while the system has a good level of maintainability and security, the following areas require improvement:

- **Code Smells**: Identified in Code Climate, these primarily consist of functions with an excessive length and functions with multiple return statements.
- **Duplications**: Identified in SonarCloud, we exceed the suggested amount of duplicated code. This is primarily duplicated code blocks between our API and the front-end, related to database connections.
- **Test Coverage**: Implementing and configuring comprehensive test coverage to accurately reflect this metric in both Code Climate and SonarCloud.
- **Maintainability issues**: The specific issues noted in SonarCloud should be resolved according to their estimated effort to maintain and ensure a low technical debt ratio.

Addressing these areas will improve the current state of our codebase and ensure long-term maintainability. Den siger current state, men jeg har skrevet noget der siger lidt om hvordan vi har brugt de tools, fjern hvis ikke fedt During the project we have however listened to the feedback from these tools and made changes. In one instance we refactored our api and src directories from being a single python

file to many different files based on their responsibilities resulting in an easier overview of the code base and increasing maintainability.

# 2 Process' Perspective

This section provides an overview of the journey from idea to system, specifically detailing CI/CD chains, system monitoring, logging, security assessments, scaling and availability strategies, as well as the use of AI-assistants.

## 2.1 CI/CD Pipeline

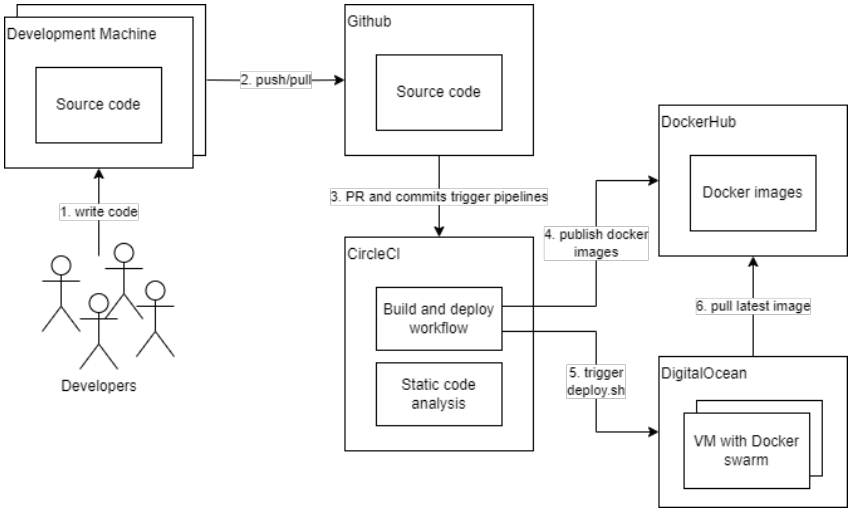


Figure 4: Visualisation of the CI/CD pipeline.

Our pipeline for CI/CD is depicted in figure **??**. The source code on Github is linted by a CircleCI workflow on each commit, and is analysed by several of the tools mentioned in the section below when a pull request is opened. When a pull request is merged into main, our `build_and_deploy` workflow is triggered, which can be seen in figure **??**. Releases have been done manually on Github once every Thurday evening.

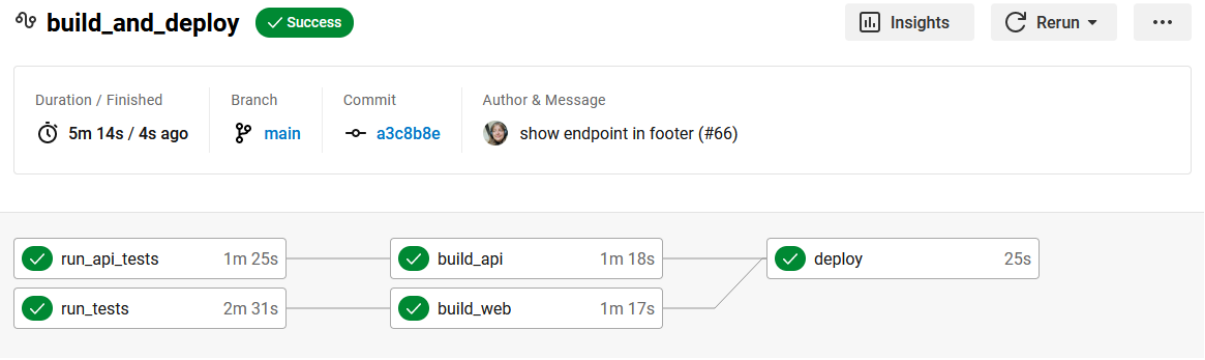


Figure 5: The `build_and_deploy` workflow on CircleCI.

### Tools and Technologies

- **CircleCI**: A continuous integration and delivery platform that automates the build, test and deployment processes of our project.
- **CodeClimate**: A platform that provides code review, offering insights into code quality, maintainability and test coverage, helping us ensure high standards and to improve code health.
- **Github**: Version Control Management
- **Hadolint**: A lint-tool that analyzes Dockerfiles to find common issues.
- **Shellcheck**: A shell script analysis tool that identifies syntax errors and common issues.
- **Sonarcloud**: A cloud-based code quality and security service that analyzes code to detect bugs and vulnerabilities.
- **Vagrant**: A tool for building and managing virtualized development environments. Used initially for VM instantiating.

#### Technology choices

The decision to manage our CI/CD pipeline on CircleCI is based on the pros and cons in appendix **??**. Github Actions would also have been a good choice for this, but as there were no obvious downsides to either or, we went with the technology that we had not work with before as to gain experience with it.

## 2.2 System Monitoring and Logging

#### Monitoring Strategies and Tools

In managing our system's health, we rely on two key tools: Prometheus and Grafana. Prometheus is a pull-based monitoring system that pulls/scrapes our defined metrics. We instrument the system with the Prometheus *client_golang* library, from which we create the metrics, register each metric using Prometheus' default register, finally creating a HTTP endpoint exposing the metrics endpoint for Prometheus to scrape [**?**]. Our approach is mainly whitebox monitoring in application-based metrics tracking the internal workings of the system. However, we do use a few blackbox monitoring elements by tracking specific endpoint responses.

Grafana is an open-source platform used for visualising and analysing time-series data. In our setup, we use Grafana as a dashboarding tool that uses our Prometheus server as the primary data source for monitoring. This allows us to create dashboards that provide real-time insights of our measures.

As from the Monitoring Maturity Model by James Turnbull [**?**], our monitoring strategy combines proactive elements, such as monitoring uptime and tracking error rates to anticipate and prevent issues, with reactive elements, such as tracking specific error codes and status of specific user requests to respond promptly to issues that arise.

#### Key Metrics Monitored

We have defined monitoring metrics for both the API and the web client. However, given the API's consistent workload from the simulation, this has been our primary focus. We have defined a total of 14 metrics for the API:

- **http_requests_total**
- **database_accesses_total**
- **errors_total**
- **get_user_id_requests**
- **get_user_id_requests_failed**
- **follow_requests**

- **follow_requests_failed**
- **unfollow_requests**
- **unfollow_requests_failed**
- **post_message_requests**
- **post_message_requests_failed**
- **not_found**
- **bad_request**
- **internal_server_error**

These monitoring metrics are visualised in a Grafana dashboard named `Minitwit_Dashboard` as shown in Figure **??**.

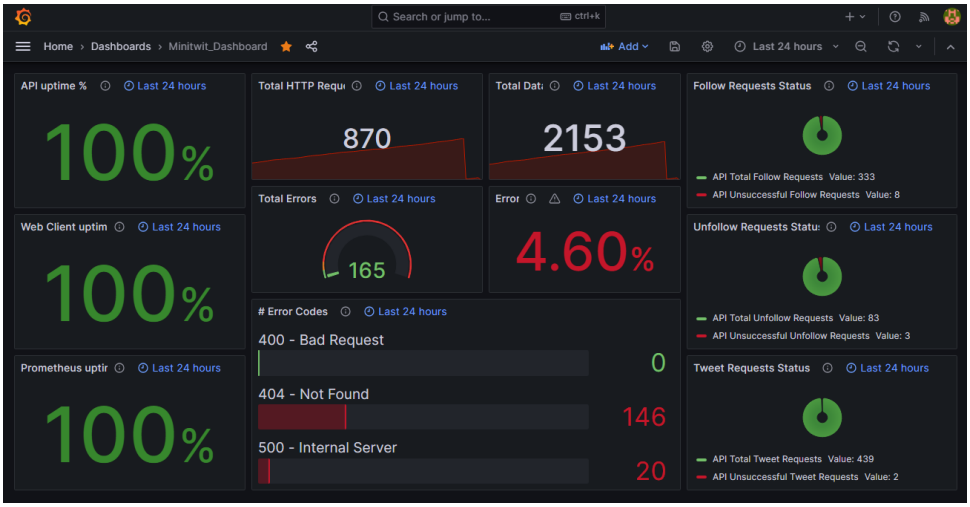


Figure 6: Minitwit_Dashboard - API Monitoring

The key metrics are categorised and displayed as follows:

**Total HTTP requests and Database accesses**, **Total Errors and Error Ratio**, **Error code distribution** of the total errors, and **Specific user request success/failure** (follow, unfollow, tweet).

Additionally, the visualisation shows the uptime percentage of the API, Web Client, and Prometheus, which is derived from predefined Prometheus metrics.

### Log Aggregation Techniques

For logging we have used a setup with Promtail, Loki and Grafana. Initially we went for an ELK stack, however after not being able to get it to work and discussing whether it was necessary to do it with that stack, we searched for alternatives and used a setup with Grafana which was already a part of our tech stack. At first this did not work, but when we added Prometheus' AlertManager it worked. The job of AlertManager is to remove duplicate log entries and route them to the correct receiver.

Our techniques included logging general activities of the system with the purpose gaining of an insight into how the system was being used although this was not super useful in our context, but could have proven pivotal as an auditing mechanism if we had experienced database specific issues. This happened on one of our log levels, namely INFO. We also had a level for errors and logged all errors in our system. When we noticed we had a concerning amount of errors of the same type, we created a dashboard view for the specific error. This setup can be seen in figure **??**

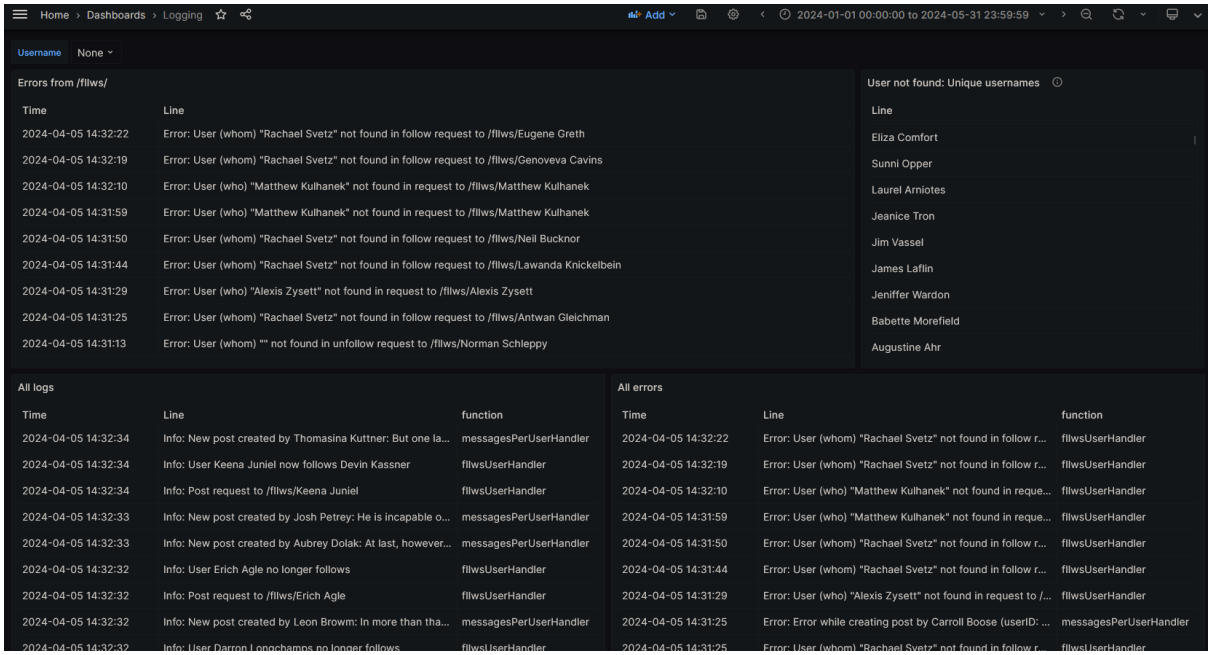


Figure 7: Grafana dashboard of logs.

## 2.3 Security Assessment

In our efforts to ensure the security of the ITU-MiniTwit system, we have utilized various tool for risk identification, such as Snyk, Metasploit and SonarCloud. To assess the security state of our system we have identified threats and provide a risk analysis.

### Risk Identification

Our primary asset is the ITU-MiniTwit web application, which includes several components, such as the frontend, backend, database and the associated cloud infrastructure.

One of the most common vulnerabilities for web applications involves injecting malicious SQL statements. Other similar threat sources consists of cross-site scripting (XSS) or misconfiguration, e.g. incorrectly configured servers, databases or applications that can expose vulnerabilities.

From this we can construct several risk scenarios:

- **Scenario 1**: An attacker performs SQL injection on the web application to download sensitive user data.
- **Scenario 2**: An attacker exploits a cross-site scripting vulnerability to steal session cookies and impersonate users.
- **Scenario 3**: An attacker exploits a misconfiguration of the web server allowing unauthorized access to system resources.
- **Scenario 4**: A developer with legitimate access exposes sensitive data or sabotages the system, either intentionally or accidentally.

### Risk Analysis

To analyze the identified risks we evaluate using a risk matrix to prioritize them based on their likelihood and impact.

| Risk Scenario | Likelihood | Impact | Priority |
|---|---|---|---|
| SQL Injection | Low | High | Medium |
| XSS | Low | Medium | Low |
| Misconfiguration | Medium | High | High |
| Developer error | Low | High | Medium |

Table 1: Risk matrix based on identified risks.

To mitigate the risk scenarios the following strategies are employed:

- **Static code analysis**: The likelihood of our risk scenarios is covered in part by SonarCloud that has security rules setup, which analyzes the source code and allows detecting SQL Injection and XSS.
- **Secrets**: To minimize the risk of exposing sensitive data, we use secrets in Docker swarm, GitHub and CircleCI.
- **Automation**: Automatic configuration e.g. using tools like Vagrant or Terraform ensures we avoid misconfigurations.

#### Security Hardening

Other security measures that we have in place are using https on our domains and using Docker secrets for sensitive data such as our database connection string.

Our HTTPS certificate is obtained through CertBot. However, due to difficulties with the standard HTTP validation method, especially concerning load balancers as described in the article "How To Acquire a Let's Encrypt Certificate Using DNS Validation with acme-dns-certbot," [**?**] we opted for an alternative approach. We used the acme-dns-certbot tool [**?**], which interfaces CertBot with a third-party DNS server for validation instead of HTTP. This method is especially beneficial for issuing certificates for individual servers behind a load balancer, where traditional HTTP certificate validation would require setting validation files on each server.

### 2.4 Scaling and Upgrade Strategy

From our monitoring dashboard we could at one point identify several API crashes on our single Digital Ocean node due to 100% CPU usage. As a method to regain availability by removing the single-point-of-failure and reach a generally more robust service, we chose to employ scaling and replication through Docker Swarm.

#### Docker Swarm

We chose Docker Swarm as our container orchestration platform because it is included by default with Docker and is simpler to use compared to alternatives like Kubernetes and OpenShift. Docker Swarm allowed us to set up a cluster consisting of manager and worker nodes, effectively creating a distributed system. In this setup, worker nodes run our stateful services, API and web client, independently, while the manager node handles service scheduling across the nodes using the routing mesh, Docker Swarm's built-in load balancing mechanism, and maintains the swarm state.

We first deployed the Docker Swarm as a separate cluster beside our single-node production server in Digital Ocean to ensure a transition that would not disrupt our live server. For each service, including our API, we added an additional replica in order to distribute the workload across multiple containers.

However, as the inherent load balancer, the routing mesh, of Docker Swarm is rather low-level, in which it from-the-shelf only redirects traffic to replicas when a node is "unhealthy", or dead, we opted for a dedicated load balancer to complement the routing mesh by employing a more sophisticated traffic distribution. Simply redirecting traffic when a node is crashed that initiated the whole focus on scaling and replication.

#### Nginx

We chose to utilise Nginx as both reverse proxy and load balancer. This involved adding a new droplet in our Digital Ocean project outside the swarm to run an instance of Nginx, which we configured to handle a more sophisticated traffic distribution between the replicated services.

To configure Nginx, we separated the configuration files into two parts, one for the API and one for the web client. The configuration file for the API includes an upstream block that lists the IP addresses of the replicas in the swarm cluster along the API ports, allowing Nginx to direct traffic to the specified instances. Additionally we added server blocks for HTTP and HTTPS traffic, effectively redirecting all HTTP traffic to HTTPS. Similarly, the configuration file for the web client follows the same pattern but includes the IP addresses and service ports of the web client in the upstream block instead. See appendix X for the configuration file of the API service.

### 2.5 AI Tools Utilized

To automate the production of easy or recurring code such as a new prometheus monitoring metric, generating skeleton code, code completion and helping us gaining a high-level understanding of the ITU-MiniTwit system, Microsoft Copilot was used by multiple developers. It was quite bad at guessing the intended implementation logic of new features.

## 3 Lessons Learned Perspective

### 3.1 Challenges with Evolution and Refactoring

We experienced that when adding new technologies to our system, other parts of our system stopped working. An example of this, which had a huge impact on our project, was when trying to add Docker Swarm, we stopped receiving logs. When parts of our system failed, we tried getting them to work again by finding guides online that was specifically targeted at making the newly added technology and the existing 'broken' technology work together. This has taught us the importance of modular solutions that are open for extension and are easily refactored. One of the agile principles state that "Working software is the primary measure of progress", and in line with this, we have truly experienced in this project how broken software halts progress.

### 3.2 Challenges with Operations

One challenge that occured due to our choice of CI/CD management tool, was that we were not able to run our automated tests until a pull request was merged to our main branch. Running tests locally

was a lengthy process as it required several docker containers being built and started. Therefore, we sometimes approved pull requests with code that would fail our tests. This resulted in situations such as with commit `7661b99`[1] and `8071acb`[2], where a new branch and pull request must be created in order to salvage the main branch. A solution to this could have been to set up Github Actions with the same tests as in our CircleCI workflow, but we decided to not do this, as it seemed redundant. However, we have learned that if running tests locally takes too much effort, it will not be done, which is the reason why automated tests are so important. And while pushing code to main that does not pass the tests can seem bad, we found that it is not as dangerous as it sounds, due to another lesson we learned in this project: the power of a fault tolerant CI/CD pipeline. Having the code on Github decoupled from the code running on our virtual machines, has made it safe to push code that we believed should be on our main branch. It has reduced our fear of pushing code, which ultimately has allowed us to continuously deliver and deploy, which again is in line with the agile principles[3].

## 3.3 Challenges with Maintenance

In this project, we have especially learned the importance of tooling such as monitoring and logging. In the beginning, monitoring was limited to the official status-page[4], where it was easy to see if *something* was wrong, but not what it was. We were able to see that we got almost no errors beside a bunch on our `/fllws` endpoint, which made us wonder whether we had implemented it wrong. This is were logging becomes a powerful tool, but unfortunately, we have had problems with this continuously through our project. At first, we spent a long time on a stack that we could not get to work. After switching to a Promtail/Loki/Grafana stack, we could finally see that the implementation seemed to be right but that there were a lot of users that were not able to be found. However, we quickly lost our logs again, when we started to work with Docker Swarm, as mentioned earlier, and again it left us completely blind to the cause of the errors we experienced. We have learned that while monitoring is an important tool to give you an overview of whether the system is up and running, it is of no help when the system is not up and running. Ensuring the ease of discovering bugs in your system is essential in making sure it is operational. In the end, it meant that one of our Github issues[5], which was identified early in our process, was never solved and closed.

#### Switching to https

Toward the end of the simulation we experienced occasional system shutdowns as mentioned in a previous section. While making the system scalable we also obtained a https certificate. After getting our endpoint updated in the simulator, we experienced that we got no more requests. As this was nearing the end of the project, we wrongly assumed that there was not being sent anymore requests. We talked to another group, who were still receiving messages, so we assumed something was wrong with our new endpoints. We tested creating users on the webserver and writing messages, and we tested posting messages with Postman to our API, both working as intended. We therefore have an assumption that something might have gone wrong when updating our endpoints in the simulator, and that this might be the reason why we on the simulator have accumulated several million errors in the final weeks.

[1] `https://github.com/TheisHS/test1-itu-minitwit/commit/7661b997afbc0de3742e7df066b37245ef7f18bd`
[2] `https://github.com/TheisHS/test1-itu-minitwit/commit/8071acb95fa1010bb102b45b32634bb6b6790e70`
[3] "Deliver working software frequently, from a couple of weeks to a couple of months, with a preference to the shorter timescale".
[4] `http://206.81.24.116/status.html`
[5] `https://github.com/TheisHS/test1-itu-minitwit/issues/41`

## 3.4 Reflection on DevOps Practices

Log: feb9 (git branch setup, later march2 argue if static analysis removes need for manual reviews), feb25 (CI/CD, using pipelines in your projectm (discuss including security scan to workflow)), automating provisioning (docker, vangrant..), instrumenting, scaling cloud environment + loadbalancing,

One part of our practices that does not adhere to the DevOps methodology is the configuration of our virtual machines. Configuration files were only stored inside the virtual machines, meaning that if we had to make any changes, we had to ssh into the machine. By not having configuration as part of our automated deployment pipeline, we created a DevOps anti-pattern which lead to an erosion of our automated cloud provisioning throuwhere we had no version control of the configuration and no easy way to inspect the code. Ultimately, it lead to more time spent on fixing human errors and on configuring new virtual machines.

# 4 Appendix

## A Choice of programming language

| | Python3 Flask | Crystal Kemal | Ruby Sinatra | Golang |
|---|---|---|---|---|
| Our experience | Moderate experience | No experience | No experience | Moderate experience |
| Types | Dynamically typed | Statically typed | Dynamically typed | Statically typed |
| Performance | Moderate performance | High performance | Moderate performance | High performance |
| SQLite support | yes | yes | yes | yes |
| Middleware support* | yes | yes | yes | yes |
| Release date | 2010 | 2016 | 2007 | 2011 |
| Deployment | Deployed using virtual environments to manage dependencies. | Compiled into a single binary executable with all its dependencies. | Requires the presence of the Ruby runtime environment and dependencies. | Can be deployed as single binary executables. |

Table 2: Programming language strengths

* Middleware in web APIs is used as a design pattern to intercept and manipulate HTTP requests. `https://azure.microsoft.com/en-us/resources/cloud-computing-dictionary/what-is-middleware`

## B Choice of CI tool

As our code is stored on Github, we've eliminated Gitlab CI as as an option. Travis CI is only free for a single month (for students), and we've also eliminated this.

| Github Actions | CircleCI | Our considerations |
|---|---|---|
| Is free ... "Cheapest for people with public repositories" | 3000 minutes for free pr month | We have a public repo. |
| Runs full pipeline automatically | Can be paused and wait for human interaction | We don't have a usecase for needing human intervention before deploying if the code passes all the tests we stup, and the CircleCI feature (even though nice) is not needed. |
| More than CI/CD - can also automate manual tasks like generating changelogs or versioning releases | Only CI/CD, but specialised in this. | We only need CI/CD for now. |
| Slower than CircleCI | Faster than Github Actions | Do we need speed? |
| Only Windows, MacOS and Linux | Every operating system | We only need Linux |
| Configuration can be split in mulitple files | Single file configuration | Cleaner setup with GHA? |
| Docker support is still a bit buggy on GHA, works only with Linux. | CircleCI has perfected its Docker support over the years to make it (almost) the de-facto environment for running builds. | We will use Docker, but still only Linux. |
| More granular control by exposing all commands. Complexity increase. | Less complex, has built in commands for often-used services. Less control. | We don't know what we need yet - so maybe more control is nice, but it being easy is also nice. |

Table 3: CI tool strengths.

## C Static code analysis

Here we provide a snapshot of our static code analysis tools from May 19, 2024.

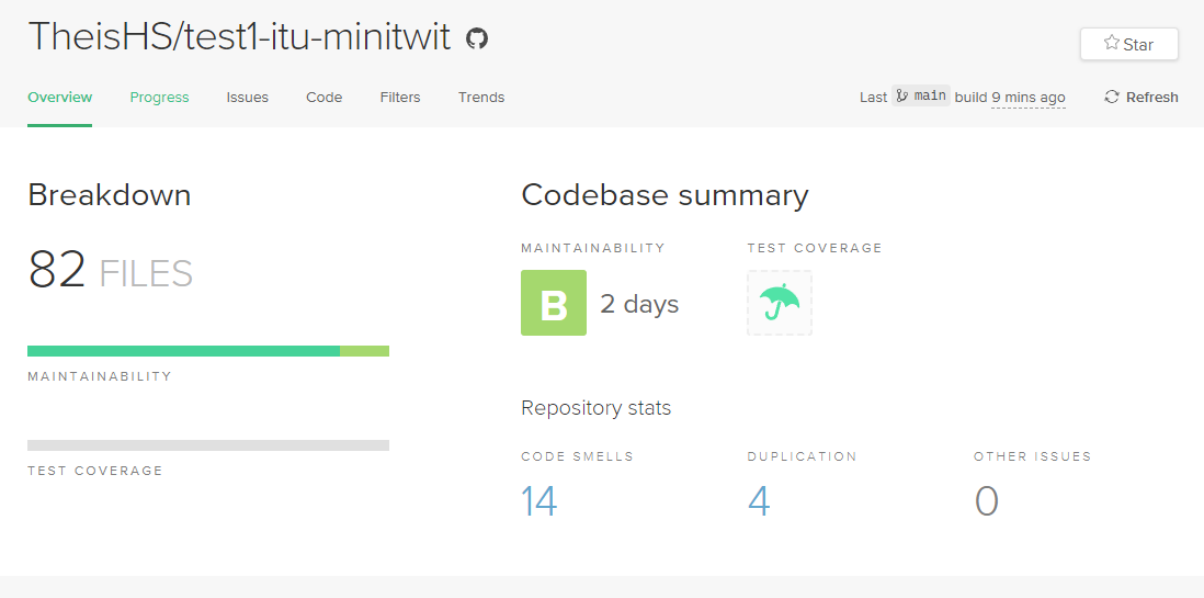


Figure 8: Code Climate summary 19/05/2024.


Figure 9: Code Climate Technical Debt Trends 19/05/2024.

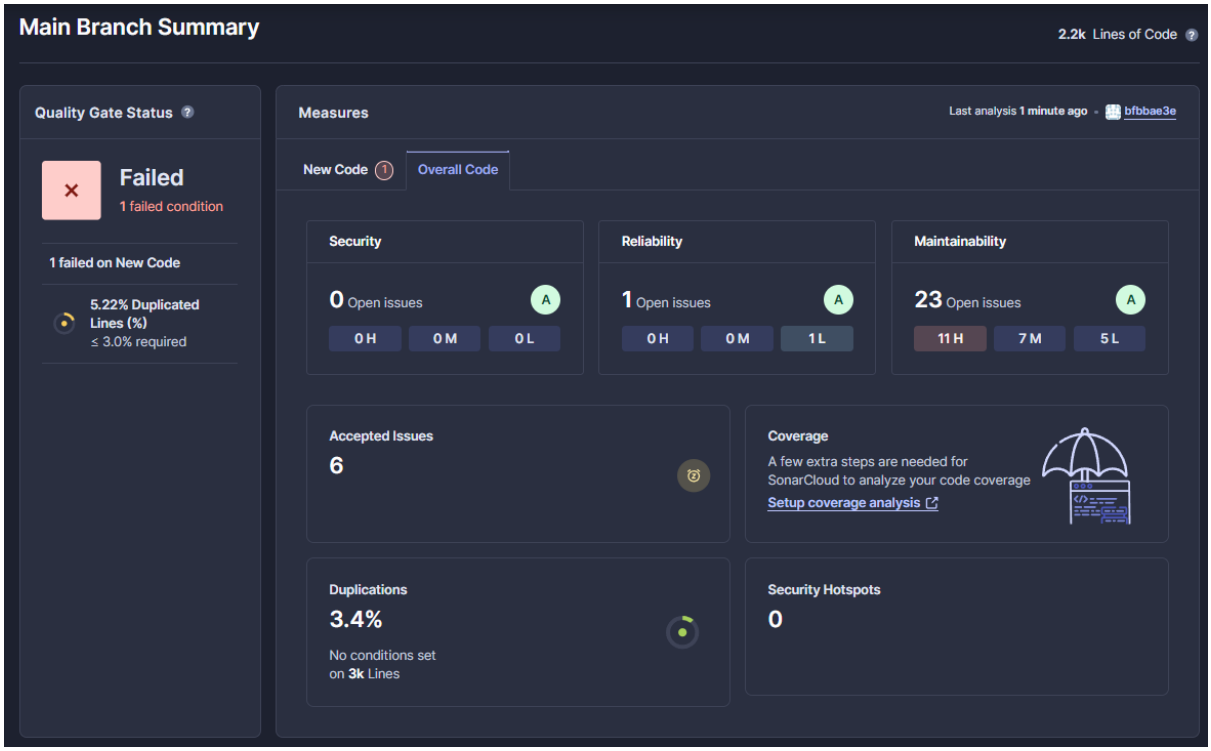


Figure 10: SonarCloud summary 19/05/2024.