DevOps 2025

IT University of Copenhagen

Team Sad People

People

- *Gábor Tódor* - gato@itu.dk
- *Zalán Kálny* - zaka@itu.dk
- *Nicklas Koch Rasmussen* - nicra@itu.dk
- *Nicolai Grymer* - ngry@itu.dk
- *Sebastian Andersen* - seaa@itu.dk

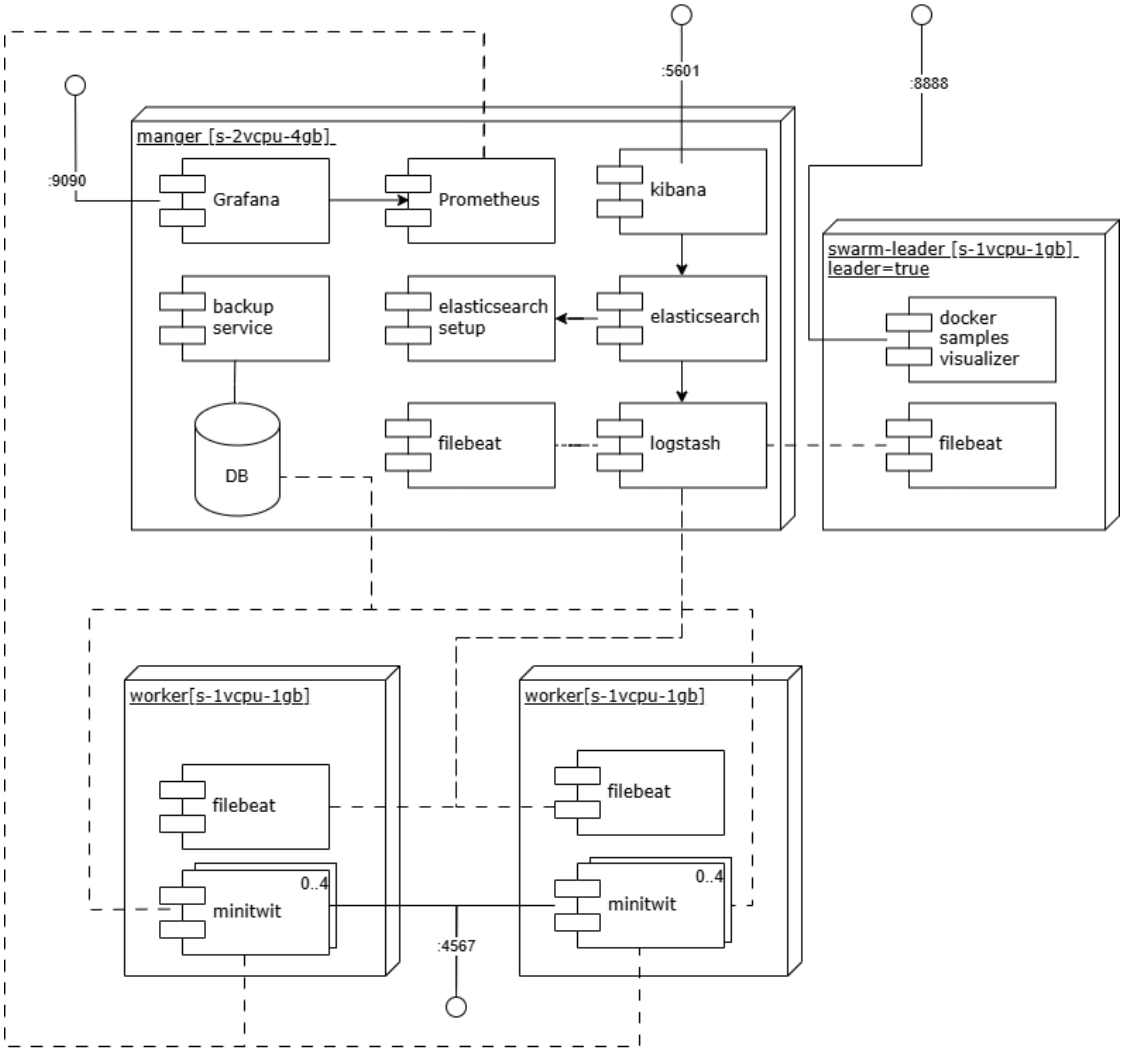
Subject

- **School:** *IT-University of Copenhagen*
- **Course manager:** *Helge Pfeiffer* - ropf@itu.dk
- **Course Code:** KSDSESM1KU

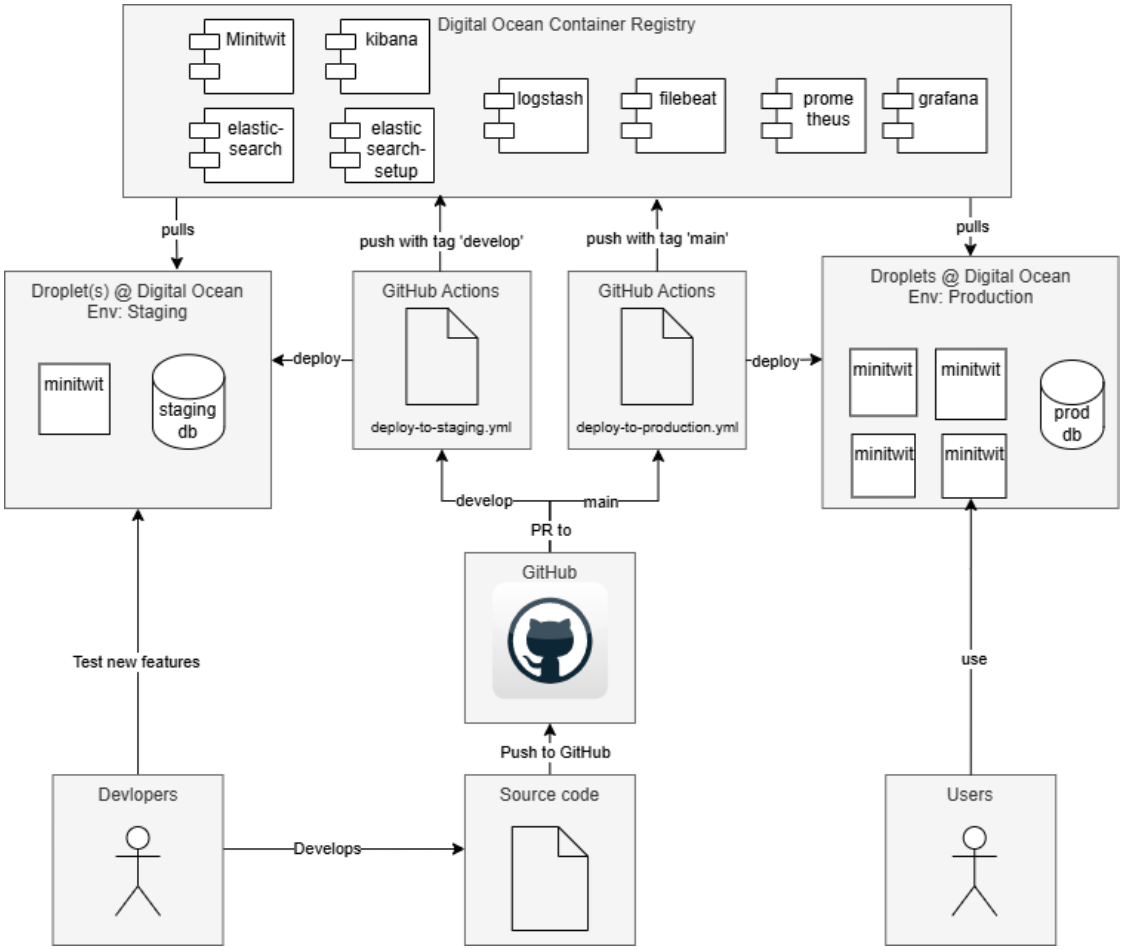
System

Design and architecture of the *ITU-MiniTwit* systems

The current achitecture of the minitwit system. The ideal architecture can be found in the appendix.



The current deployment flow:



All dependencies of *ITU-MiniTwit*

Application

Tech/Tool	Description
Ruby	Programming language
Sinatra	Lightweight Ruby web framework for building simple web applications
Sequel	Ruby Database ORM
pg	Ruby PostgreSQL library
digest	For md5 and sha2
json	For json manipulation
dotenv	To work with the .env file
rack	Defines a standard way for Ruby web frameworks and web servers to communicate
prometheus/middleware/exporter	Prometheus Ruby client to expose metrics to Prometheus

Tech/Tool	Description
active_support/time	Powerful time-related extensions to `Time` and `Date`

Database

Tech/Tool	Description
PostgreSQL	The database management system
eeshugerman/postgres-backup-s3:15	To automate PostgreSQL database backups and upload them to S3 using PostgreSQL

Testing, Linting and Static Analysis

Tech/Tool	Description
Python	Programming language
requests	Python library for making HTTP requests
Playwright	End-to-end testing framework for web apps
pytest	Popular Python testing framework for writing simple tests
rspec	Ruby testing framework focused on behavior-driven development
SonarQube	Static code analysis tool to measure code quality
standardrb/standard	Ruby file linter
hadolint/hadolint	Dockerfile linter
erb_lint	ERB (Embedded Ruby) template linter

Deployment

Tech/Tool	Description
ubuntu	Popular open-source Linux distribution
NGINX	High-performance web server and reverse proxy server
Docker	Platform for developing, shipping, and running applications as containers
GitHub	Cloud-based platform for hosting and managing Git repositories
GitHub Actions	CI/CD service by GitHub for automating workflows like testing and deployment
GitHub Secrets	Secure way to store and manage sensitive information (e.g., API keys) in GitHub workflows
Vagrant	Tool for managing and provisioning virtual machine environments using simple configurations

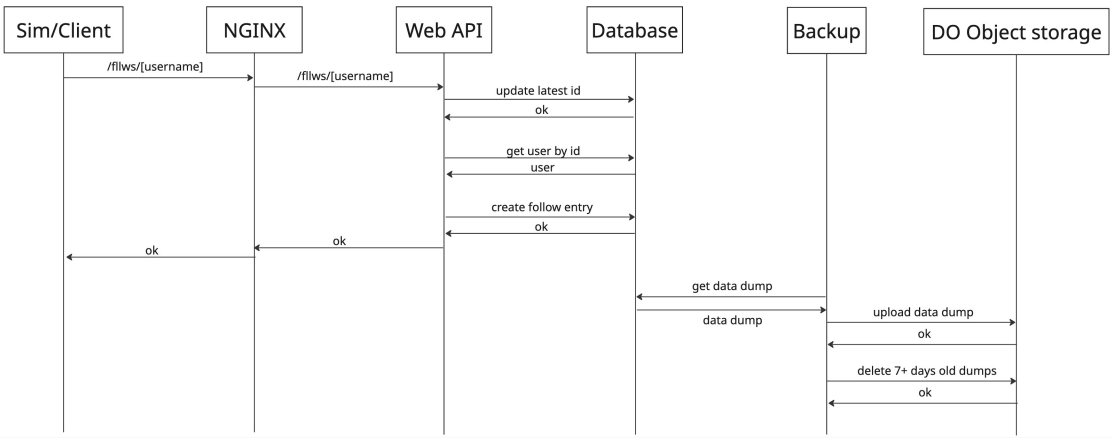
Tech/Tool	Description
Terraform	Infrastructure as Code (IaC) tool for automating cloud infrastructure provisioning
Digital Ocean (DO)	Cloud service provider offering scalable computing resources and services
DO Container Registry	Service to store and manage Docker images
DO object Storage (S3)	Object storage service, similar to AWS S3, for storing files and backups

Monitoring + Logging

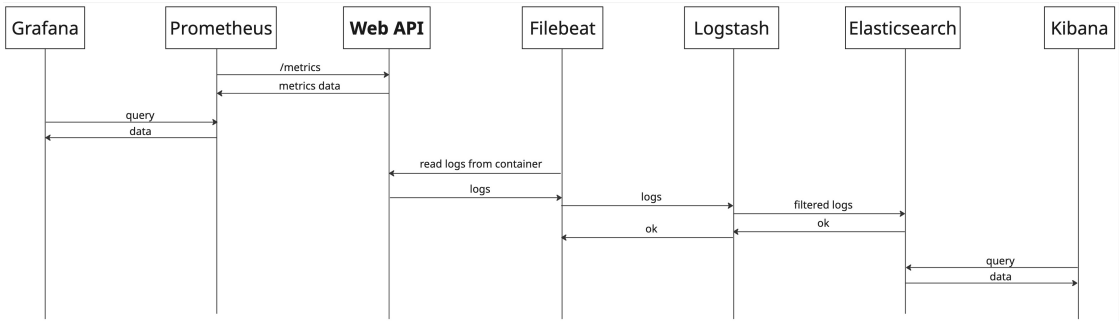
Tech/Tool	Description
Elasticsearch	Distributed search and analytics engine used for indexing and querying logs, metrics
Logstash	Data processing pipeline tool that collects, transforms, and forwards data to Elasticsearch
Filebeat	Lightweight logshipper for forwarding and centralizing log data to Logstash or Elasticsearch
Kibana	Data visualization and exploration tool tailored for Elasticsearch, used for dashboards and log analysis
Prometheus	Open-source monitoring and alerting toolkit designed for metrics collection
Grafana	Visualization tool used to create dashboards from time-series data like Prometheus metrics

Important interactions of subsystems

Both the simulator and client contact the same API application, so both sequence diagrams look identical. The following sequence diagram uses the simulator request endpoint `/fllws/[username]` as the baseline. The following sequence diagrams do not take Docker Swarm into account, as the underlying communication is hidden.



For monitoring and logging, we have also included a sequence diagram to show how they interact with each other.



Current state of the system based on static analysis

At the current state of the project all major functionalities are implemented and out on production with only a few minor issues remaining, which are listed on the Github repository's Issues page here.

Throughout development we also used static analysis tools such as SonarQube and CodeClimate to ensure code quality. The quality assessment reported by SonarQube can be seen on the image below:

We managed to avoid/solve *Security* and *Reliablity* issues, and the remaining *Maintainability* issues are mainly "multiple string literals" problems, which we deemed non-crucial. Code duplication was also kept at minimal, coming in at 7.9% for the the entire codebase (*Note: source code provided by the course, such as the flag_tool or the simulator was excluded from the quality assessment*).

Arguments for the technology choices

Programming language and framework

We considered several programming languages and frameworks for the API application, including C# with Razor-Pages, Ruby with Sinatra, and Go with Gorilla. We chose Ruby + Sinatra because of the readability and learnability. More in Appendix.

Database

We went with PostgreSQL. Both MySQL and PostgreSQL are great options for our purpose - familiarity with PostgreSQL and it being open-source motivated our choice. More in appendix.

Virtualization techniques and deployment targets

Containerization and virtualization

We have chosen Docker to containerize our applications, as it allows us to create lightweight, reproducible environments, packaging all necessary dependencies together, making development, testing and deployment all easier.

For virtualization, on our staging environment, we use Vagrant to provision VMs as it allows us to set up and configure reproducible machines from code.

More advanced Infrastructure as Code (Terraform)

We use Terraform instead of Vagrant for our production environment because it is flexible and more powerful, allowing us to manage our infrastructure as code across multiple droplets. Vagrant is used for staging because it provides a simpler setup, allowing us to quickly spin up and tear down instances as needed.

Hosting: Digital Ocean vs AWS/Microsoft Azure

They both offer rental of VMs. We chose Digital Ocean as our cloud provider mainly because of its simplicity and great tooling. Because of the simplicity we can focus on everything else, rather than learning how a complex tool works. Pros and cons found in Appendix.

CI/CD system

We have chosen Github Actions as they offer a free, scalable, and secure CI/CD solution with seamless GitHub integration, customizable YAML workflows, reusable actions, multi-platform support, and efficient automation through parallel execution and event-driven triggers. More about the advantages in the Appendix.

Testing (Unit, UI, E2E)

Minitest vs Rspec

We chose RSpec, as we are prioritizing readable and maintainable technologies. It is also a very popular and large toolkit, which is great for testing.

Selenium vs Playwright

Playwright is a fast, open-source browser automation tool backed by Microsoft that launches browsers quickly, runs tests in parallel, and uses optimized headless mode for faster execution than Selenium. It supports Chromium, Firefox, and WebKit natively, auto-waits for elements to reduce flakiness, and uses a persistent WebSocket connection instead of Selenium's HTTP-based WebDriver. We tried to implement both Selenium and Playwright. For us it was impossible to get Selenium to work as we had critical problems with the external browser drivers needed for the Selenium framework to work. Playwright did not need external drivers since it has built-in protocol clients; it was also easy to setup and use for developing new UI tests.

Monitoring and Logging

For monitoring we chose **Prometheus** and **Grafana** as our stack. For logging we went with the **ELFK** stack. More in the appendix.

Process

This perspective tries to clarify how code or other artifacts come from idea into the running system.

CI/CD chain (tools)

Our CI/CD pipeline uses two main branches: `main` (production) and `develop` (staging), with GitHub managing the repository and issues. Features are developed in `feature`-branches, merged into staging for testing and review, and then into production after passing all checks. This process involves three phases: development on a feature branch, review and testing in staging, and final approval and deployment to production. More info in Appendix.

Automated Testing and Quality Gates

All pull requests and pushes to staging and production trigger GitHub Actions workflows that build a Docker container with a PostgreSQL database for testing. The pipeline includes unit tests (Ruby Rack), E2E tests (Playwright), simulation tests (Python), and static code analysis (SonarQube with ≤3% Ruby code duplication). Branch protection rules enforce this process, and Ruby and Docker code are auto-linted on every push using `standardrb` and `hadolint`. More in appendix.

Build and Deployment Process

We deploy using GitHub Actions, which builds containers and uploads them to Digital Ocean's container registry. We also upload a new version of the Ruby application, and if there are updates (detected by `git diff`) to the configs for either our monitoring or logging stack, we also push a new version of that. If tests pass, then we automatically continue to deploy. Currently we differentiate between containers designated for the staging and production environment by assigning them such a tag.

The deployment process involves SSH'ing into the manager node of the Docker Swarm, that is running as droplets (virtual machines) on Digital Ocean, and running a deploy.sh script, which simply pulls the newest version of the stack from the container registry.

On pushes to main, we automatically create a new release, which includes bumping the application with a new minor-update, meaning 1.0.0 turns into 1.1.0. If we wish to introduce a patch or major update (or do no release at all), we can specify in the commit message.

We orchestrate the containers using Docker Swarm, and given the size of our application, we currently follow the direct deployment rollout strategy, where we simply push a new version to all worker nodes at once. This is a point for improvement.

Environment Management and Infrastructure

We use Terraform scripts for setting up our production environment. For artifact management, we use Digital Ocean's container registry, where we differentiate between container versions using staging and production tags.

To distribute secrets that GitHub Actions can access, we set up GitHub Secrets to keep an access key to Digital Ocean on which we deploy our application.

Rollback Strategy

To roll back, it would require manually SSH'ing into the server and modifying the compose script to depend on a specific container in Digital Ocean's registry. This is definitely a weak point, making it time consuming to

rollback and represents an area for future improvement.

Monitoring and Observability

Once the feature is successfully integrated into the production codebase, we use Prometheus and Grafana to monitor the application, ensuring that the feature introduces no error, and that operation levels remain the same. In case of noticeable changes, we use Kibana to navigate logs to help diagnose the problem. Kibana queries Elasticsearch, which receives logs from Logstash, who in turn accesses log-files using Filebeat.

Choice of Architecture

We chose to have a staging environment, as it helped us understand how to properly integrate features and architecture changes before proceeding to do so on production. Given this projects work included a lot of architecture change and adding new technologies, this helped us immensely in preventing down-time by ensuring that the config worked on a deployed environment.

Monitoring

Monitoring is configured in our system using Prometheus and Grafana. Prometheus handles time-series based raw metric collection, Grafana handles metric visualization.

How it works in our system

Our Minitwit application uses an existing Ruby client library of Prometheus, and exposes raw metrics on the `/metrics` endpoint. The Prometheus service periodically scrapes the data from this endpoint of the application. In Grafana, these collected metrics are visualized using highly customizable dashboard panels. In each panel, metrics from Prometheus are queried at regular intervals using PromQL queries. We also set up email alerting for certain panels, this way we can be notified when certain conditions, thresholds etc. are met.

Panels configured on our Grafana dashboard

- **HTTP response count by status codes:** Time-series. Shows the number of HTTP responses during the last minute at any given point of the time range, grouped by status codes.
- **HTTP error response count:** Time-series. Shows the number of HTTP client- and server-side error responses during the last minute at any given point of the time range. Email alerting is also set up for this panel, when the server-side (5XX) error count during the last minute hits a certain threshold.
- **Latency percentiles:** Time-series. Shows the median, 95th and 99th percentiles of latency (request duration) in milliseconds at any given point of the time range.
- **Average latency:** Gauge. Shows the average latency in milliseconds over the given time range.
- **Total registered users:** Stat. Shows the total number of registered users in the Minitwit application.

As seen in the list above, aside from the *Total registered users* panel, we mainly do infrastructure monitoring in our system. We also planned to include more meaningful application-specific monitoring too such as the number of new users/new posts made in the last X minutes; due to time constraints, however, we did not implement these.

Logging

Logging is configured in our system using the ELFK stack: Elasticsearch, Logstash, Filebeat and Kibana.

How it works in our system and log aggregation

First, Filebeat handles log shipping by reading and collecting logs from each of our Docker containers' log files. These logs are then forwarded to Logstash, which processes and transforms the log data as needed. Logstash then sends the processed logs to Elasticsearch where they are indexed and stored for efficient querying. Finally, the aggregated logs are visualized using Kibana.

The reason we also used Filebeat for log shipping is because it is much more lightweight than Logstash. Traditionally, Logstash is the log aggregator which collects, transforms and forwards logs for further processing, but since we have multiple physical nodes in our system, each node would require one Logstash instance running on them. Instead, each node has a Filebeat instance running which handles log shipping for the containers running on that node.

What we log in our system

Filebeat forwards all logs, from all Docker containers in the system. In Logstash, we filter based on the logging levels (filtering/parsing is specific to each service's logging format). We try to parse each log record to extract the logging level; if the parsing was successful, all *DEBUG*- and *INFO*-level messages are excluded, everything else is forwarded to Elasticsearch for indexing. Additionally, Logstash also drops many unneeded fields in each log record, so that the number of indexed fields will stay relatively small.

Security Assessment

By running through the OWASP Top 10 list on security assessment, we have done the following analysis:

- **A01:2021-Broken Access Control** The system has two levels of access control, a user or public user. We have found no vulnerabilities for user-specifc endpoints. But anyone can access the API this allows malicious websites to make authenticated requests to the API on behalf of logged-in users.
- **A02:2021-Cryptographic** Upgraded to HTTPS, but still exposes its port over HTTP, leaving credentials vulnerable to interception. Passwords are hashed with SHA256 but without salting, and the hard-coded simulator protection key in the public GitHub repo undermines its security.
- **A03:2021-Injection** We use the ORM Ruby Sequel, which includes sanitization of input before constructing SQL statements. Developers can create raw SQL statements, but we have opted not to do this given the impracticality and security risks.
- **A05:2021-Security Misconfiguration** Following a ransomware attack demanding bitcoin, we closed ports and changed default passwords to improve security. However, `ufw` was later found disabled, exposing all services, and overly permissive CORS settings remain a known vulnerability(explained in A01).
- **A06:2021-Vulnerable and Outdated Components**

- Our system allows weak passwords and lacks proper email validation or confirmation, making it easy for bots to create accounts one of which accounts for 99.9% of activity. On the developer side, 2FA is not enforced for DigitalOcean access, and important security updates, like Ruby 3.3.7 to 3.4.4, have been repeatedly postponed.
- **A09:2021-Security Logging and Monitoring Failures** We experienced a log overflow causing our production service to fail. This failure did not cause any warnings, causing three days of downtime for our application. We will elaborate on how we fixed this when reflecting on system operation.
- **A04:2021-Insecure Design, A08:2021-Software and Data Integrity and A10:2021-Server-Side Request Forgery** We have not been able to identify any issues regarding this.

Applied strategy for scaling and upgrades.

We have used both horizontal and vertical scaling.

For the logging and monitoring it was nessesary to scale vertical where we scaled from 1 CPU, 1GB RAM (s-1vcpu-1gb) to 2 CPU, 4GB RAM (s-2vcpu-4gb) to handle the workload associated with monitoring and logging.

We increased the number of node/droplets from 1 to 4 to increase availability when we upgraded from docker compose to docker swarm with a docker stack deployment containing multiple replicas of the minitwit application.

To handle higher user load we first switched from SQLite to PostgreSQL to get a more reliable database. After that we also indexed the database to ensure efficient data access.

Use of AI

This project used both Copilot and chatbots (from OpenAI and Anthropic) to support development—Copilot for line-by-line code help, and chatbots for elaboration, comparison, creation, and problem-solving prompts. Claude 3.7 Sonnet outperformed ChatGPT o1 in understanding code, configs, and bugs, offering more detailed and accurate responses.

For full description read Appendix.

Reflection

Evolution and refactoring

Database migrations

The first migration from SQLite to Postgresql happened at a stage, where no active users were using our platform (The simulator was yet to start). This meant that we could safely upgrade without having to move over data, which would have otherwise been a hassle given the SQL dissimilarities.

Given the educational purpose of the project, we later sought out an opportunity to perform another database migration. Such an opportunity arose when database optimization became a necessity. Before optimizing, we deemed it necessary to introduce an ORM (see Issue#85 and Issue#121), which would improve the developer experience as well as migration experience going forward. Given the new database structure introduced by the ORM, although quite similar, we needed to migrate from one database with one schema, to another database with another schema. The approach taken involved extracting data from one postgres instance in the shape of SQL Insertion statement, which we then manipulated to fit the new data schema, and then simply ran the SQL insertion statements in the new database.

As soon as the migration was done, we switched to the new application image, meaning we now served requests from the new database. This approach involved having 5 minutes of forgotten data, and 3 seconds of lost availability. We found this price and strategy reasonable, although the 5 minutes of lost data, could have had serious impact on the business. As we will later discuss, we found that using logical replication proved to be a much nicer approach to copying data.

Transition from docker compose to docker swarm (networking problems).

Transitioning onto a cluster of machines with docker swarm came with multiple obstacles.

Docker compose versioning problem (moving to stack).

Firstly, the docker compose version that supports `docker stack deploy` is a legacy version of docker, and there is a difference in the features and syntax supported which caused some problems. The `docker stack` does not take `build`, `container_name` and `depend_on` into consideration, and also handles unnamed volumes differently. Therefore, we had to rewrite the compose scripts to make them compatible with docker stack. One of the biggest change that also resulted from this is that from this point on, we had to build the configuration files of each service into Docker images, and publish them to our DigitalOcean registry. This came with several complications, such as automatically building images in our workflows during deployment, and ensuring correct versioning/tagging for each image, since publishing all images in every deployment would take a lot of resources and time. This change also solved another one of our problems: previously, we manually copied all configuration files onto the server using `scp`, which made every deployment error prone.

Secondly, the swarm nodes were able to communicate with each other, but self-instantiated virtual networks defined in the docker-compose file did not propagate to worker nodes, leaving application containers unable to contact the database, and prometheus unable to collect monitoring events. To accommodate the issue, we destroyed and redeployed new virtual machines, and this time used the VPC IP address to define the IP address of the manager node. This meant that workers are referring to the manager using the virtual network layer, which solved the communication issue.

Logging issues

The initial deployment of our logging stack was quite problematic (see Issue#164 and Issue#184), as Elasticsearch turned out to be very resource heavy, consuming almost ~60-80% CPU at times (before introducing logging, it was ~10% at peak load), and also taking all available RAM. We tackled this in two ways:

- We scaled the droplets vertically, giving them more resources, as previously discussed. This was necessary because the stack has larger minimal resource requirements than what we had.

- We introduced Logstash filters, which dramatically decreased the number of fields indexed by Elasticsearch, lowering its resource consumption greatly.

We also encountered another issue with Filebeat, after switching to Docker Swarm. We found that one Filebeat instance needs to run on each node, because every instance needs direct access to read the containers' log files. This was solved by introducing global replication for not only the Minitwit application, but for Filebeat as well.

Large amount of features cloging up in staging (impossible to migrate to production)

Fully implementing new features sometimes took longer than a week. Due to the development workflow described earlier, the staging environment was used as a development/testing environment. This resulted in features being gate keeped by other partially implemented features. Some of the new features also required changing others, such as the migration from docker compose to docker swarm requiring a full rewrite of the docker compose scripts. These made it impossible to migrate the changes to production in time and delayed the release of the full implementation of the logging and monitoring.

Operation

Database logical replication resulting in db crash

Migrating from docker compose to the docker swarm included the use of a PostgreSQL feature: logical replication, which allows PostgreSQL instances to live sync data from one running PostgreSQL instance to the other. This feature is typically used to keep a hot stand-in database ready. In our case, it meant we would actively sync data from the active production database, onto the new production database, allowing us to switch from one stack to the other with zero downtime, as the stand-in database would become the new default.

Unfortunately, after switching a few days later, the pub/sub mechanism of logical replication in PostgreSQL accidentally corrupted a tracking file, meaning the database would immediately crash on start. This problem was accomodated by running `pg_resetwal` to reset the corrupted file, and then unsubscriping from the expired subscription. The subscription does not provide any value at this point, as we swapped from the old to the new production machine, and the old one has been turned off.

Log overflow problem

After quite some time into development, our production droplet became overwhelmed by the large volume of logs being generated by the different containers. This log overflow consumed all available disk space on our droplet, resulting in the droplet becoming inoperable, also shutting down our whole system. We did not notice this issue for a few days as Grafana also shut down, being unable to send alerts to us. After noticing the issue, we were also denied SSH access into the droplet. We had to use DigitalOcean's recovery console to regain access to the server, delete unnecessary log files, and restore normal operation. Learning from this incident, we introduced log rotation in our docker compose file which limits the maximum number and size of generated log files.

Backup strategy

Although it's great to solve problems on your own, sometimes others have done a great job already. And this proved to be the case for backing up a Postgres database. Simply adding the `eeshugerman/postgres-backup-s3:15` image to the manager node and configuring environment variables, we successfully set up a cron job that automatically backs up daily, and sends the backup to DO's space storage, which is S3 compatible. Using the exact same script, it also includes functions that easily allow restoring from a previous backup. The latter is a crucial step, for when things are burning. The container likewise provides clean-up functionality, such that only 7 days of backups are kept.

Maintenance

Upgrading to NGINX, setting up UFW, moving to domain

Upgrading to NGINX, we learned multiple things about running a system. First being that having a staging environment to learn how to run commands in the correct order proved great when building a functioning shell script that immediately upgrades the production service. Second, that although we had configured our UFW with all the right ports, actually the service was disabled by default. Only by realizing that the 443 port was open (although not specified) did we realize that UFW needs to be activated. From this we gathered that it is important to double check the firewall and other security measures, to ensure they are configured properly. Luckily, our database was not exposed by PORT from the docker network, and therefore inaccessible, but having full access to other ports may have exposed other vulnerabilities on the machine.

Simulator IP protection stopped simulator access

Refactoring simulator requests to only be accepted from a single IP address helped us prevent other malicious users from interfering with the active simulation requests. However, when the new update was pushed, unfortunately the IP protection feature also protected us from the actual simulation IP. Although this worked perfect locally, moving it onto production showed that the feature declined all simulator api requests. From this we learned about the importance of being able to quickly roll back an update. During this experience, we found that we did not have a previous versioned container ready to roll back to, and instead had to allow all IP's which was possible by passing in `ACCEPTED_IPS=*`, which essentially disabled the whole IP protection.

Style of work

Reflection on the workflow

Each Friday after the lectures we met up to have an extensive meeting about the current state of the project. First we shared what features we had worked on the past week, what kind of problems we had faced and how we had solved them. We did this to keep everyone up to date with all the different new technologies and featues implemented.

Second we discussed the content of the lecture and inspected what new features to implement in the next week. Then we discussed *how* to implement the new features and the pros and cons of the different options. For each task we created a new issue on GitHub.

Third we discussed *when* to implement the new issues. The group members had different schedules and varying capacity due to other commitments such as hand-ins for other courses. We took this into considaration

when we delegated the work. We typically worked in three teams:

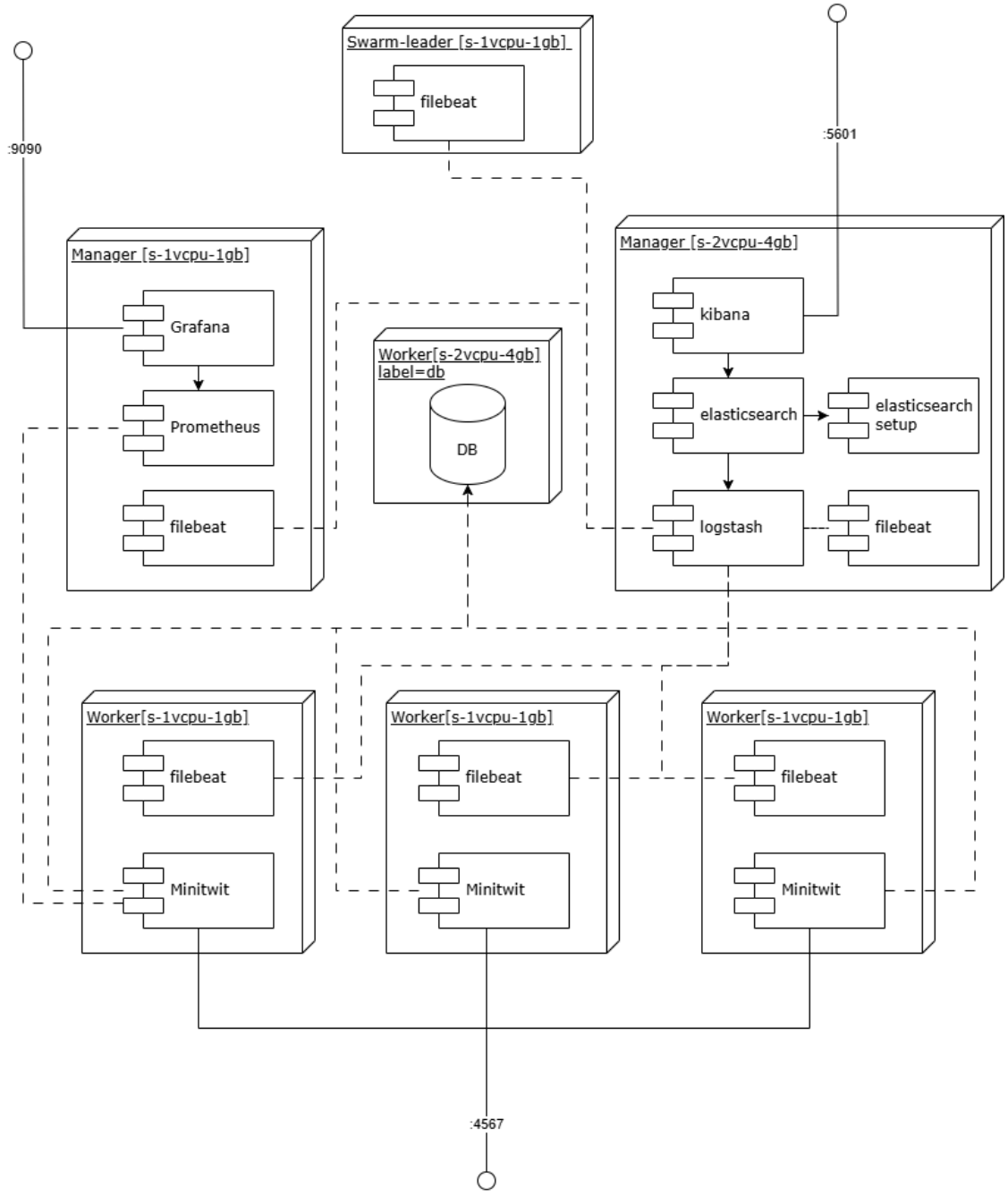
1. Nicolai
2. Gábor and Zalán
3. Sebastian and Nicklas

These Friday meetings worked very well for us, as we all had a very busy schedule. These meetings allowed us to delegate the work, inspect the progress, adapt the plan and be up to date in terms of the implementation details, while still going in depth into the subjects.

Appendix

Ideal Architecture

The ideal architecture with less pressure on the swarm leader node:



Choosing technologies

Programming Language and Web Framework

- **C# + Razor-Pages**
  - Familiarity with both the language and framework

    - Big enterprise standard
    - Enterprise-grade software
    - Very popular in Denmark
    - Verbose + A lot of boilerplate
    - Compiled => faster execution
    - Founded by Microsoft (Trustworthy)
- **Ruby + Sinatra**
    - No familiarity
    - **Interesting (to learn new technologies)**
    - Lighter than Go
    - **Very readable & learnable**
    - **Rapid development**
    - Interpreted => Slower
        - Reports of lower scalability
- **Go + Gorilla**
    - Compiled => Faster
    - Great for concurrency
    - Not designed for web applications
        - Reports of longer development time
    - More low-level features (e.g. pointers)
    - DevOps => popular for microservices
    - Founded by Google (Trustworthy)

We chose Ruby + Sinatra because of the readibility and learnability.

It allows us to do rapid development.

We also find it interesting to learn new technologies.

Database

- **SQLite**
    - SQLite locks during writes
    - Does not scale as well
    - Great for development
- **MySQL**
    - Simpler feature set
    - Industry standard
    - Great for large amount of reads
    - Oracle owned
- **PostgresQL**
    - Advanced query features
    - Industry standard
    - Open-source

Deployment

- **Digital Ocean**
  - **focuses on simplicity**
  - **easy to learn CLI/interface**
  - **great tooling**
  - sufficient credits to cover our needs
- **AWS/Microsoft Azure**
  - industry leading, popular
  - customizability
  - steeper learning curve
  - more enterprise focused

Digital Ocean also provided its own container registry, which was our choice when looking for a registry later into the development process, given it's integration with the DigitalOcean platform. Only after using it, did we realize that the biggest upsides are primarily for users of digital oceans other deployment tools, that are not running on rented VMs. Also given the price of DO's container registry, we would most likely migrate to Docker Hub, if the project had continued.

GitHub Actions

- Seamless integration into Github
- Cost-effective: it is free for open source, public repositories
- Extensive Marketplace, reusable workflows, pre-built actions
- Scalability and flexibility
- Supports multiple runners: Linux, macOS, and Windows
- Highly customizable workflows: You can define the workflows in YAML files
- Built-in security features, secrets management, role-based access control
- Parallel execution: Supports matrix builds and parallel jobs, reducing build and deployment time
- Tight GitHub integration: Workflows can trigger on pull requests, pushes, issue comments, and other GitHub events, enabling efficient automation

E2E Testing

Playwright launches browsers faster and runs tests in parallel by default.Headless mode is optimized, leading to faster execution times compared to Selenium.

Auto-waiting for elements prevents flaky tests, whereas Selenium often requires explicit waits.

Supports multiple browsers out of the box: Playwright works natively with Chromium, Firefox, and WebKit (Safari engine)

Architecture: Playwright uses a WebSocket connection rather than the WebDriver API and HTTP. This stays open for the duration of the test, so everything is sent on one connection. This is one reason why Playwright’s execution speeds tend to be faster.

History: Playwright is fairly new to the automation scene. It is faster than Selenium and has capabilities that Selenium lacks, but it does not yet have as broad a range of support for browsers/languages or community support. It is open source and backed by Microsoft.

While newer and with less community support, it offers modern features and performance advantages. We tried to implement both Selenium and Playwrigt, the Selenium was difficult to implement due the the required browser executables. Playwright was easier to setup and develop. So that became our choice for E2E tests. More info in Appendix.

Unit testing

- **Minitest**
  - Built-in
  - Often standard in web frameworks
  - Very light and efficient
- **RSpec**
  - Very readable DSL
  - Very popular alternative to minitest
  - Large toolkit

Monitoring

- **Prometheus** provides several features which are useful for us:
  - centralized, pull-based metric collection
  - metrics are time-series based -> can show changes over time
  - had ane existing Ruby client library -> easy setup & integration with our system
- **Grafana:**
  - integrates well with Prometheus
  - supports a wide range of metric types
  - (relatively) easy to use by writing PromQL queries
  - supports alerts (e.g. email)

Logging

- **ELFK stack - Filebeat + Logstash + Elasticsearch + Kibana**
  - very popular, pretty much industry standard
  - nice interoperability (Kibana is tailored for Elasticsearch)
  - Logstash + Elasticsearch can be very resource-heavy -> Filebeat addresses this with lightweight log collection
- **Grafana + Loki**
  - less resource-heavy
  - could integrate well into our existing Grafana service
  - less popular choice

We chose the ELFK stack mainly because it's the most popular choice, and we wanted to get familiar with it.

CI/CD Pipeline

In the following section, we will discuss the CI/CD pipeline of our system, and for this, we discuss two key branches: `main` and `develop`. The `main` branch includes the code running on our production environment, and `develop` branch includes the code running on our staging environment. For the sake of communication, we will simply address these branches by `production` and `staging`.

We use GitHub for handling our repository and tracking the process with their issue system. We use a branching strategy, where features written in issues are worked on in `feature`-branches. Once ready, they are then merged into `staging` and then into `production`. This enables us to test and deploy the feature before production, at the cost of slightly longer delivery times. This means that for features to make it through to production, it includes three phases:

1. We work on the issue using a `feature`-branch. Developers work on and finalize the feature on this branch.
2. Once ready, a pull-request is created to merge the `feature`-branch into `staging`, where tests, linting, static code analysis must pass and 1 fellow team member must also approve the request, before being able to merge it into staging.
3. Once deployed to the staging environment, if the staging environment sees no failures and passes a manual test, a pull-request into `production` is made. Once approved by tests, linting, static code analysis and a fellow team member, the feature is pushed into main.

Development environment: local => feature-branch => staging => production

As explained in the Process section when developing new features you branch off `develop` then implement the changes and test them **locally** via the local docker development environment `docker-compose.dev.yml`. Then changes are pushed to a remote branch so another person can continue working on the changes. When the feature/tasks is completed a pull request is created. When the changes are approved they merge into `develop` and trigger a new deployment to the staging environment. If the changes work in the staging environment a pull request from `develop` into `main` can be created. Once the pull request is approved a new release and deployment to production is triggered.

Automated Testing and Quality Gates

Pull-requests as well as pushing to staging and production, include several tests that are performed using workflows that trigger a GitHub action, which builds a Docker container with which these tests can be performed. On top of the web API container, an associated PostgreSQL database is instantiated, to perform E2E and simulation tests.

- Unit tests are performed using Ruby Rack
- E2E tests are performed using Playwright
- Simulation tests are performed by instantiating a new environment, and using Python to perform requests (these tests helped us identify a misaligned status code response - read Appendix)
- Static code analysis using SonarQube, which requires ≤3.0% code duplication in the Ruby application.

GitHub **branch protection** rules ensure that developers follow this workflow. Concretely it prevents users from merging directly into the `develop` and `main` branch.

On top of the above, Ruby and Docker code is formatted and linted on push to any branch. This is done using the GitHub action modules `standardrb/standard-ruby-action@v1` and `hadolint/hadolint` respectively.

Full Security Analysis (OWASP Top 10 list)

By running through the OWASP Top 10 list on security assessment, we have done the following analysis:

- **A01:2021-Broken Access Control** In the system only two levels of access control exist in the system. Either you are a user, who can post, follow and unfollow, or you access as a public user. For user-specific endpoints, we have not found any vulnerabilities. CORS settings however, allow anyone to access the API. This misconfiguration allows malicious websites to make authenticated requests to the API on behalf of logged-in users.
- **A02:2021-Cryptographic** We've upgraded from HTTP to HTTPS, but still expose the port of the application, meaning IP:PORT still gives users access to the service in non-encrypted ways, such that network eavesdroppers can capture username and passwords. The hashing algorithm has been upgraded from MD5 to SHA256, but unfortunately without salting, allowing attackers who gain access to the database to easily crack passwords with rainbow tables or brute force attacks. Lastly, the simulator protection-key is hard-coded which means anyone with access to the public github repo, can essentially bypass that security measure.
- **A03:2021-Injection** We use the ORM Ruby Sequel, which includes sanitization of input before constructing SQL statements. Developers can create raw SQL statements, but we have opted not to do this given the impracticality and security risks.
- **A04:2021-Insecure Design** Given the tiny feature set, we could not find anything particularly noteworthy about the design.
- **A05:2021-Security Misconfiguration** After experiencing a ransomware attack, requiring bitcoin for our data, we closed ports and changed the default password to prevent future attacks. Similarly, we discovered that `ufw` was disabled by the end of the course, which exposes all services to the web. Lastly, we are aware that CORS settings are overly permissive as elaborated in A01.
- **A06:2021-Vulnerable and Outdated Components** Our system has very weak password checking, which allow users to create easily hackable accounts. Simultaneously, weak email validation and not sending a confirmation email makes it particularly easy for bots to create users. In fact, 99.9% of our activity is from a single bot.
- On the developers side, we did not require 2FA to log into DigitalOcean, bringing our level of security down to the weakest login-type of the five team members. And technically, Dependabot has been suggesting a Ruby update from `3.3.7` to `3.4.4`, which have been postponed multiple times.
- **A08:2021-Software and Data Integrity** We have not been able to identify any issues regarding this.
- **A09:2021-Security Logging and Monitoring Failures** We experienced a log overflow causing our production service to fail. This failure did not cause any warnings, causing three days of downtime for our application. We will elaborate on how we fixed this when reflecting on system operation.

- **A10:2021-Server-Side Request Forgery** We have not been able to identify any issues regarding this.

Other issues and bugs

Returning wrong status code (misalignment with simulation)

We implemented the simulator test in our testing workflow. It runs to ensure that the endpoints are available, works and return the correct status codes. After we had implemented the simulator test they failed and we realised that one of our endpoints was misaligned with the specification. The endpoint returned the wrong status code. By implementing the simulator tests we discovered the issue in a very early stage.

Stale README.md throughout project

Throughout the project we have not always been the best to update the README.md, we have prioritized implementing the features for each deadline over continuously updating the documentation, in line with the Agile Manifesto: *Working software over comprehensive documentation.* Due to features clogging up in staging we had plenty of problems to solve to get working software.

Use of AI

This project included the use of both chatbots and in-editor help using copilot. These were provided by OpenAI and Anthrophic.

Copilot increased speed by solving minor problems through it's line-for-line help. To increase the precision, prompt-like comments would be added prior to the line of interest, or specific prompts would be used to concretely specify the desired change.

Chatbots on the other hand involved four primary types of prompts:

- Elaboration: Please explain X technology
- Comparing: What is the difference between X and Y technology
- Creation: I want to X
- Solving: I want X, but instead Y happens.

Elaboration and comparison were primarily used at the planning stage of implementing new technologies, or for developers unfamiliar with existing technologies already implemented.

Creation is used throughout the implementation of technologies or features, but as these become more integrated into the system, the scope of the problems being addressed tend to shrink, focusing on smaller, more specific changes. As more code is added to the codebase, solving unwanted behavior becomes more important, and makes out large parts of prompts.

Additional reflection on use of chatbots: we found that Claude 3.7 Sonnet provided better code-based responses as well as understanding misconfigurations and bugs. It gives detailed descriptions of different variables and potential flaws in the code and configs. This is measured against ChatGPT o1.