mini-x report

Course Code: BSDSESM1KU

Student	Email
David A. Feldner	dafe@itu.dk
Marius Thomsen	mariu@itu.dk
Marius W. S. Nielsen	mawn@itu.dk
Markus Grand Petersen	mgrp@itu.dk
Michael Daniel Fabricius	midf@itu.dk

1 Systems perspective

1.1 Description of the project

mini-x is a blazingly fast twitter/x clone written in Rust with the Actix framework. Originally rewritten from MiniTwit - this contains the original files provided by ITU

1.2 Design and Architecture of the project

mini-x is designed with a focus on performance, scalability and security. The application is structured into two main parts: the API server and the frontend client as seen in `main.rs`. The API Server and Frontend Client are two sides of the same coin. The frontend part lets users interact with mini-x through a user interface whereas the API Server lets you interface with the application using JSON data for compatibility with the course simulation.

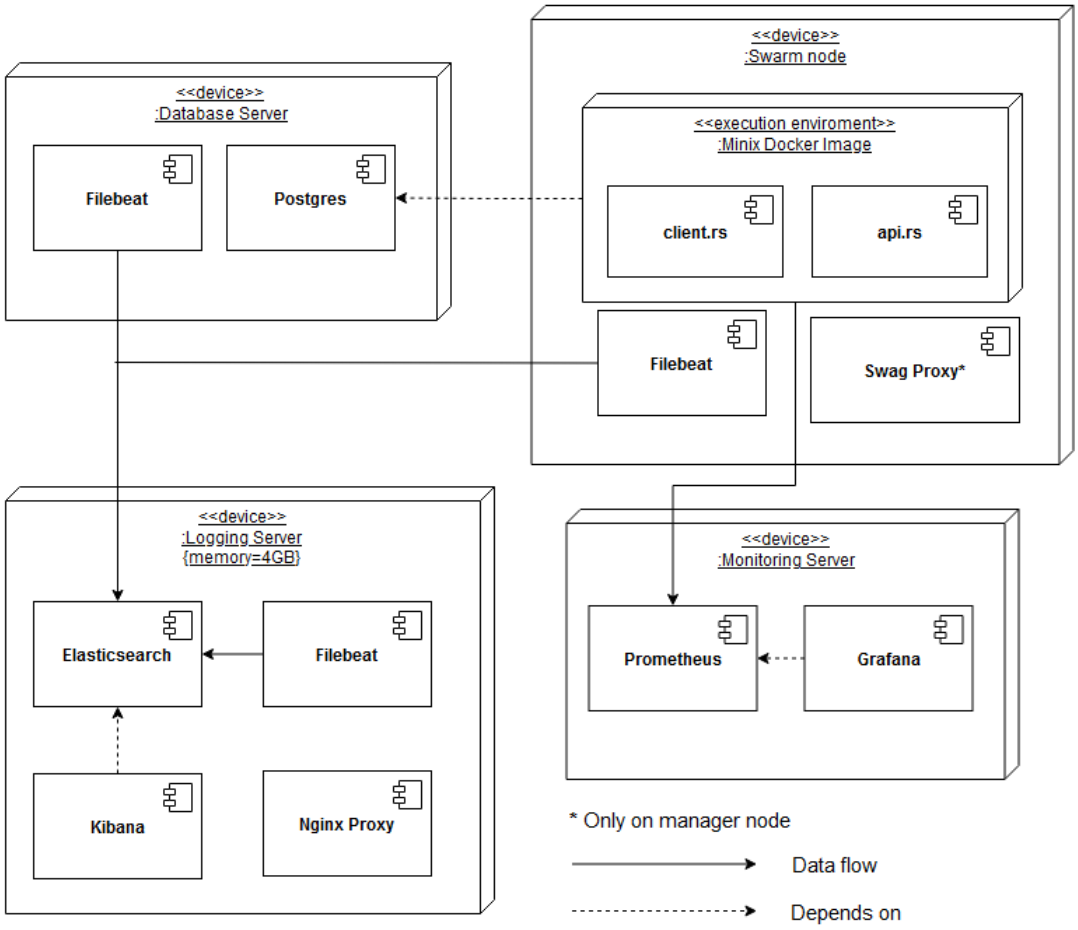

Figure 1.1 - 3+1 Deployment viewpoint showcasing the project's architecture

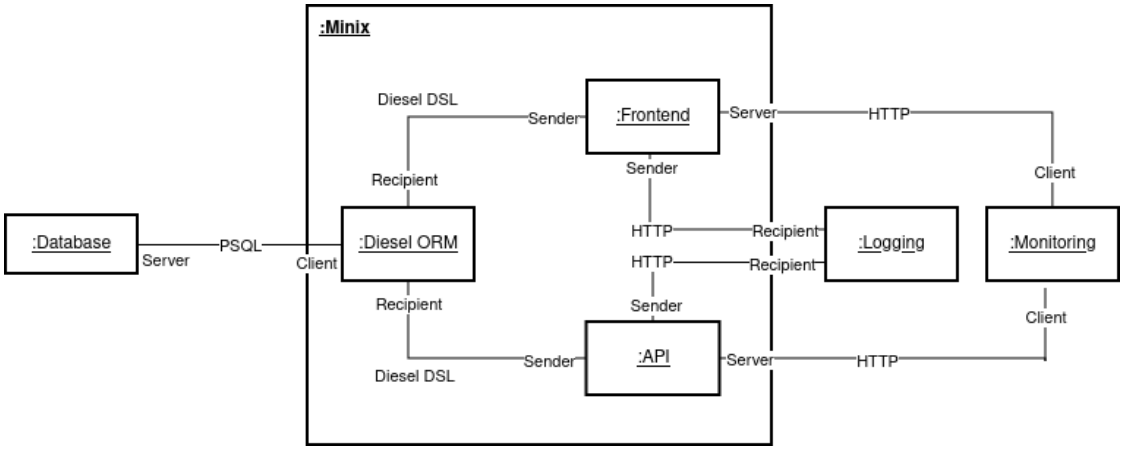

Figure 1.2 - 3+1 Component and Connectors Viewpoint showing the components of mini-x and their connections

1.2.1 Frontend Client & API Server

The API server and frontend client are built using Actix-Web, a blazingly fast web framework for Rust. They handle all HTTP requests related to user authentication, message posting, retrieval of messages, following users, and provide endpoints for each.

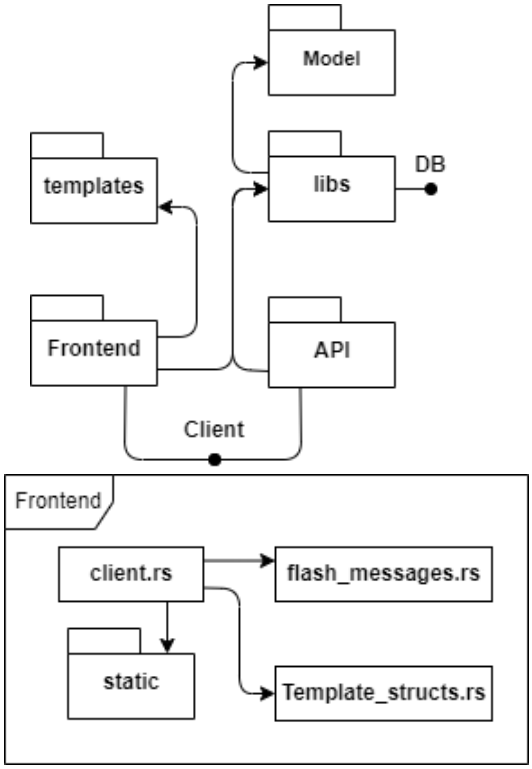

Figure 1.3 - 3+1 Model viewpoint showcasing the structure of the application.

1.2.2 Swarm Node

The swarm node depicts the group of devices on which our application is replicated. One of the swarm nodes acts as a load balancer i.e. delegating incoming requests to different swarm nodes. The manager node also hosts a Swag proxy to enable HTTPS for user requests, and each swarm node uses a Filebeat container to ship logs to Elasticsearch.

1.2.3 Logging Server

The logging server uses Elasticsearch to store logs, Kibana to serve logs, and Filebeat to collect its logs. It also hosts an nginx Proxy with basic authentication (username and password) to secure Elasticsearch and Kibana. The device also needs 4GB memory to run Elasticsearch without any worries.

1.2.4 Database Server

The database server hosts the PostgreSQL database. The server stores all persistent data, including user information, messages, follower relationships and metadata. The database schema is defined using Diesel ORM, which provides type safety and compile-time guarantees for database interaction.

1.3 Dependencies of mini-x

- Rust and Actix-web:
  - The backend service is written in Rust, with Actix-web as the chosen framework due to its performance and ease of use for developing REST APIs. Actix-web acts as a web server.
- PostgreSQL
  - Used for data storage in a robust and scalable way.
- Diesel.rs

    - Diesel was used for ORM to ensure safe database interactions. Diesel provides type safety and convenient DSL for Rust, such that complex SQL queries can be constructed safely.
- Docker and Docker Swarm:
    - Docker is used for containerization to ensure that the app runs identically across varying environments. Docker swarm manages a cluster of Docker Engines so we can spread workload horizontally.
- EFK stack:
    - Elasticsearch, Filebeat and Kibana were used for logging
- Prometheus and Grafana
    - Prometheus collects data from our API and frontend. The data is then shown in Grafana for monitoring

1.4 Current state of mini-x

Since we are using Rust, we cannot share a grade of code quality, as many of the popular Static analysis tools do not offer support for the main body of our code. However, the Rust compiler is super strict regarding memory safety, and won't compile if anything is unsafe. We also use clippy and rustfmt in our merging workflows to make sure that our code is uniform in style no matter who writes it.

This is a sequence diagram of a client requesting a resource and its complete journey through our system.

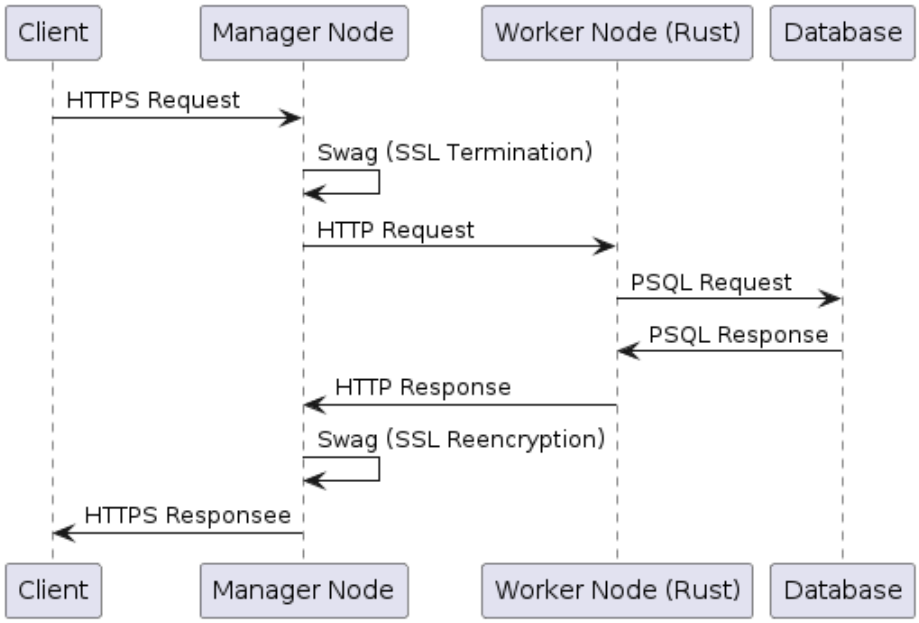

Figure 1.4 - Sequence diagram of a user request to our service

The most interesting part of this diagram is probably our reverse proxy Swag, which wraps our application with an SSL certificate, making sure traffic from the user to the server stays encrypted, even on unsafe networks. Here we run swag on the manager node that terminates the encryption and forwards it to the worker node that needs to process the request. The reverse happens when the response is transferred back to the client, wrapping the HTTP request with encryption.

Note that the API and the web interface have similar functionality. Where the difference is in the contents of the HTTP and HTTPS requests.

2 Process' perspective

2.1 CI/CD Explanation

To ensure that our system is always in a healthy condition and that as we have the highest quality as possible we have several workflows and tools to enable this.

Workflows/Tools and their purpose

- Tool: Docker
  - Used as our containerizing software to ensure consistency in the environment, load balancing and reliability
- Tool: Github actions
  - Our CI/CD chain is built with GitHub Actions, which allows us to run workflows when changes are pushed to our Github repository.
- Workflow: Continuous Deployment - Run on push to main
  - Our main workflow which builds the new code, pushes the latest image to Docker Hub and then deploys the update to our docker swarm.
- Workflow: Publish - Run on push to main with version tag
  - Whenever we reach a new milestone we push a new git tag. This workflow then builds our application, archives the current source code in a zipped file and makes a release along with all build artefacts.
- Workflow: Rust Format Check - Run on PR or push to main.
  - Runs static code analysis tool (linting tools). For this, we use the Rust packages clippy and rustfmt and run them against our code. This catches any linting and code quality issues.
- Workflow: Test mini-X - Run on PR
  - Runs the API and frontend Python tests we inherited against our service. This ensures we only merge when all tests pass.
- Workflow: Generate Report PDF - Run on push to any branch
  - Generates a PDF report from the markdown in our repository. This uses the `baileyjm02/markdown-to-pdf@v1` GitHub Action to generate the report with a headless browser, then `stefanzweifel/git-auto-commit-action@v5` to commit the result back to the same branch.

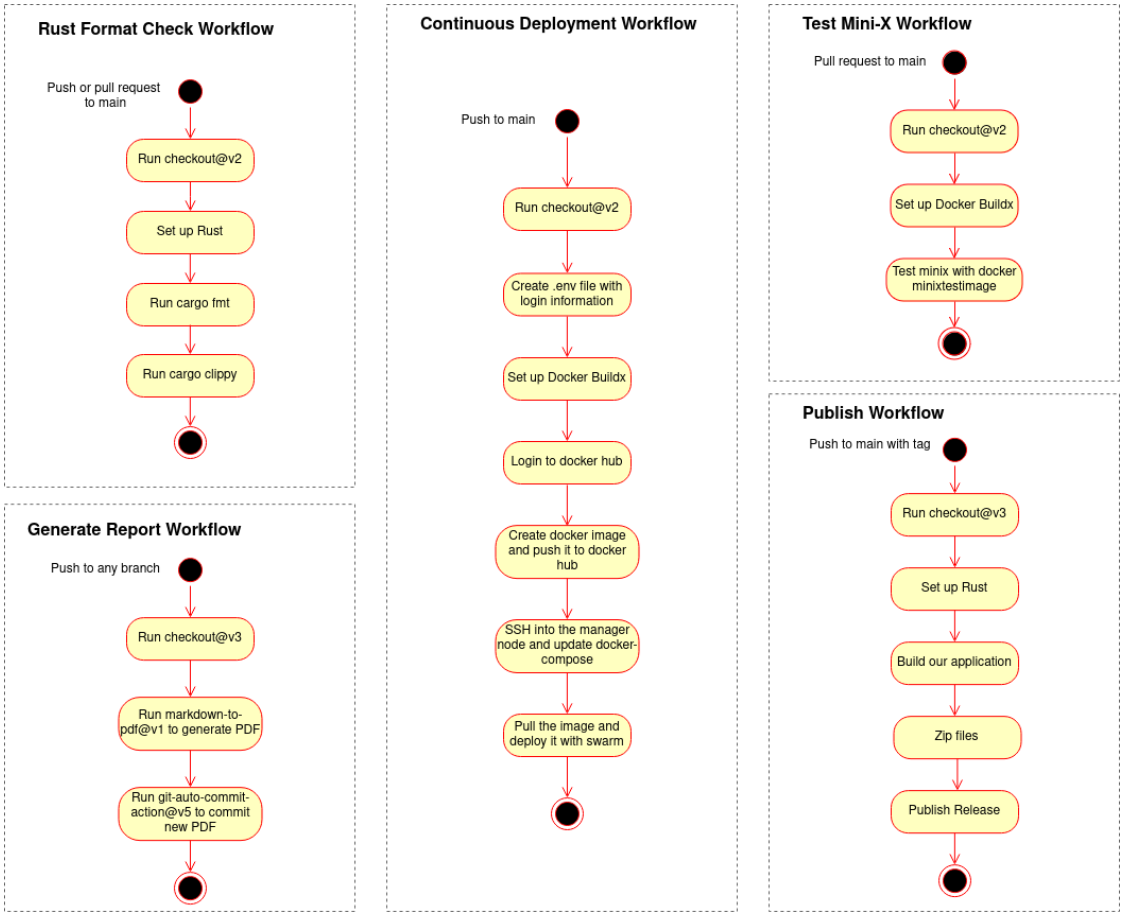

Figure 1.5 - Activity diagrams of our Github workflows

2.2 Monitoring

To monitor our systems, we use the `actix_web_prom` Rust package alongside Prometheus and Grafana. This allows us to collect, store and visualize metrics specific to our Actix Web application. Prometheus is our primary monitoring system, collecting and storing metrics as time series data. It has a powerful query language, PromQL, to facilitate the possibility of retrieval and analysis of custom metrics, however, only default metrics from the `actix_web_prom` package are collected at the moment. Grafana provides the visualization layer and includes interactive dashboards for viewing metrics collected by Prometheus.

Using the `actix_web_prom` package, we leverage the `PrometheusMetricsBuilder` to collect and expose a variety of important metrics:

- `api_http_requests_duration_seconds_bucket` : Represents the bucketed count of HTTP request durations in seconds. Useful for understanding the distribution of request durations and identifying latency issues at various time intervals.
- `api_http_requests_duration_seconds_count` : Tracks the total number of HTTP requests. This monitors the traffic volume hitting the API.
- `api_http_requests_duration_seconds_sum` : Sum of the total duration of all HTTP requests in seconds. Useful for calculating the average request duration over time.
- `api_http_requests_total` : Cumulative count of all HTTP requests received. Provides a high-level view of API usage over time.
- `scrape_duration_seconds` : Duration of scraping metrics from the target. Ensures that the metrics scraping process itself is efficient and does not cause significant overhead.

2.3 Logging

To log our system, data is collected and stored as log data from other components in the system. Among the data collected are IP address, URL, request method, response code, and user agent. We use the EFK stack (Elasticsearch, Filebeat, and Kibana) to provide powerful search capabilities. Elasticsearch is the core of the EFK stack and is responsible for storing and indexing the log data. Kibana provides a user-friendly web interface for searching and visualizing the logs stored in Elasticsearch. It connects to the Elasticsearch service and uses a Docker volume to persist data. Filebeat is responsible for collecting log data from various sources such as Docker containers and forwarding it to Elasticsearch.

2.4 Security

2.4.1 Assets and their value

- User information (username, passwords, messages, etc.)
- Computational power of our six virtual machines
- Availablity (users can use the application)
- Users (having users using the application holds value)

2.4.2 Threats and Risks to Assets

- Availability
  - Using just a single machine with ffuf, we could overload our application.
- User information
  - Obtaining the database password (stored in discrod), one could read all user information.
  - The application ran with HTTP meaning user information was a target to eavesdropping.
  - Access to logging VM could proivde some user information (e.g. user posted message at xx:xx, ip address of user)
- Computational power
  - Obtaining our ssh key (stored in discord) would allow free access to any of our virutal machines.

2.4.3 Assessment and Action

The biggest threats to our system are a denial of service attack and evaesdropping of user information. The reason for this is that these threats are far more accessible than direct access to our virtual machines, which requires personal files or passwords. From this assessment we chose to implement HTTPS for the application.

2.5 IaC Strategy

We use Vagrant as our tool for IaC, along with the vagrant-digitalocean and vagrant-docker plugins. In our vagrant file, we have described all the virtual machines, called droplets on digital ocean, needed for our service, allowing us to bring our service up with a single command. Vagrant will automatically start new droplets, open necessary ports, transfer the docker-compose yml, and other needed files and environment variables, install docker and start docker-compose on the VM. Since all our VMs run Docker, they only need configuration files, and will automatically pull all Docker images needed.

To allow for horizontal scaling and high availability, we use docker swarm to serve our frontend and API. We use 3 separate VMs described in our vagrant file, which each hosts an instance of our application in the swarm. Docker swarm automatically load balances to share workload between Docker engines within the swarm. Unfortunately vagrant does not set the swarm automatically. Since we don't know the public IPs of all nodes before they are up, we can not make them connect. Instead, we have a shell script called `SwarmSetup.sh` which remotes into each machine and connects them to the swarm. To update the swarm we use the default docker swarm rolling update policy, which updates one node at a time until all nodes are up to date.

3 Lessons learned perspective

3.1 Evolution and refactoring

The initial refactoring from python2 to working python3 was primarily done with a tool called 2to3 which handled pretty much everything.

After that, we chose to rewrite it in Rust - a language that none of us had used before. It turns out that it was an effective method to learn a new language. Because we already had the overall architecture of the code, we only had to worry about learning the language.

After we had the initial program in place we primarily extended the project with CI/CD improvements and wrapped everything with Docker.

We then migrated to an ORM still using SQLite3, shortly after we migrated to Postgres.

Lastly, we enabled logging, monitoring, HTTPS, and a complete IaC.

3.2 Operation

3.2.1 Data loss

During the project, we had 2 incidents that caused data loss. The first incident happened right at the start of the simulator when we deployed the database without a volume. We had also set up a deployment workflow for when code was pushed to the main or merged with a PR. This meant that approving any PR or pushing to the main would cause our database to be deleted. Since everyone on the team was not aware of this issue, we ended up deleting our database. Because of a recent backup and quick action, we quickly got up and running again - now with a volume for the database.

When we switched to having our Postgres instance on its server, we had set the password to be 'Postgres' on a Postgres server running on port 5432 - the default Postgres port. This caused our database to be deleted by adversaries before we changed the password. Quite a silly mistake, we thought having an insecure password for a day or two would be fine, but that was not the case. Again due to backups, we were able to restore some of the data.

3.2.2 Adding indexes live in production

After looking at our response time from day to day we noticed that it was higher as time went on. This was a big problem for us. After hooking up our Postgres database to a locally running client of mini-x with timers in the code, we could see that the database was the culprit and our application was still blazingly fast. After

seeing which queries were slow we put an index on the database while it was in production and immediately resolved our speed issues. Some queries were still slow, and we fixed those by rewriting our ORM code to another query.

3.3 Maintenence

Since we wrote the application in Rust with speed in mind, everything else ran smoothly throughout the project. Our primary bottleneck was always our database. We could see this by timing our endpoints.

4 Usage of LLM's in mini-x

With regards to the use of LLMs in mini-x, ChatGPT was used in the early stages of development as a means for fast researching. Since the group was unfamiliar with the programming language known as Rust, we used ChatGPT as an introductory tutor extracting the basics of the language far quicker than searching through docs. As we began the port to Rust, we used Copilot as another aid in the process of learning the API of new frameworks such as Actix-web, PostgreSQL and Docker.