IT UNIVERSITY OF COPENHAGEN

DevOps: Software Evolution and Software Maintenance Final Exam

Container Maintainers
Group F

Adam Hvitsted Rose (adjr@itu.dk)
Danielle Marie Dequin (ddeq@itu.dk)
Danyal Yorulmaz (dayo@itu.dk)
Jakob Henriksen (jarh@itu.dk)
Sabrina Fonseca Pereira (sabf@itu.dk)

6th Semester, Spring 2023
Data Science/Software Development BSc

Word Count: 3293

BSDSESM1KU

Contents

1 Introduction

In the project report, we will go through our process of refactoring a MiniTwit application originally developed in Python and Flask. We chose to use Go and the Gin Web Framework for our API, a PostgreSQL database and a simple front-end using only HTML and CSS.

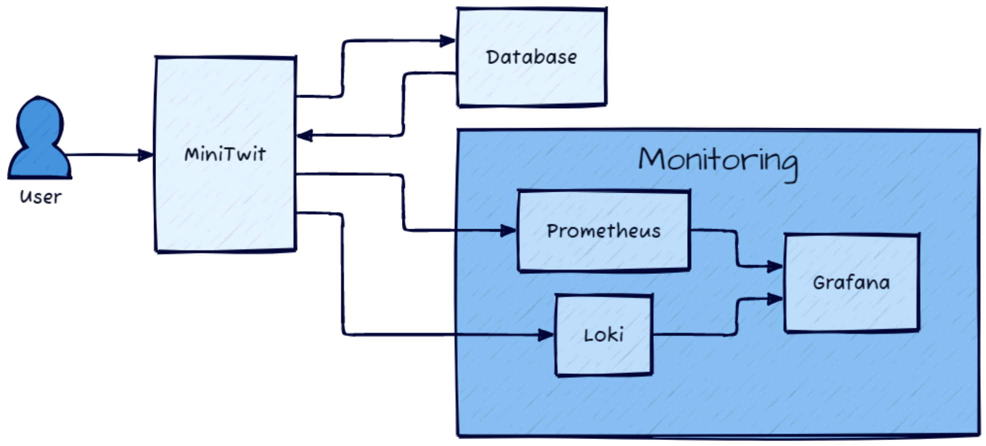
Artifact	Source
MiniTwit	161.35.210.125:8080
Prometheus	65.22.72.94:9090
Grafana	165.22.72.94:3000
GitHub Repo	github.com/ContainerMaintainers/MiniTwit-Golang
Kanban	github.com/oGrafanargs/ContainerMaintainers/projects/3
CodeClimate	codeclimate.com/github/ContainerMaintainers/MiniTwit-Golang
SonarCloud	sonarcloud.io/project/overview?id=ContainerMaintainers_MiniTwit-Golang

Table 1: Project Artifacts and their Sources

2 System's Perspective

2.1 Design and Architecture

The ITU-MiniTwit system is based on a client-server architecture and composed of several distinct components. These components are each isolated in their own environments to ensure a clear separation of concerns and easy maintainability.



Backend This is the core of the MiniTwit application, written in Go and the Gin Web Framework. The backend handles all business logic, processes requests and communicates with the PostgreSQL database.

Frontend The frontend, built with HTML and CSS, communicates with the backend via RESTful APIs to display information to the user and send user requests to the backend.

The infrastructure of the MiniTwit application is containerized using Docker. This ensures that each service runs in an isolated environment with its dependencies, providing a uniform development and deployment process and minimizing discrepancies between different environments.

Application We created our own image for the application and hosted it on Docker Hub to facilitate continuous integration and deployment. We decided to start our own image from Alpine instead of the official Go image as this reduced its size by ∼40%.

Database The database runs in its own container, built from the official Postgres image.

Monitoring We use Prometheus and Grafana for our monitoring systems. Each running in its own container, also based on the official images.

Logging We use the Grafana Loki Docker plugin to aggregate logs, which serves as a data source for Grafana.

Overall, our MiniTwit system was designed with a focus on modularity, scalability, and maintainability. This allows for easy updates and additions to individual components without affecting the system as a whole.

2.2 Dependencies of Our ITU-MiniTwit System

The following list includes all technologies and tools used in our system.

Web Application:

- **Go** - Chosen programming language for our back-end.
  - **Gin** - Web framework for Go, helps with creating endpoints.
  - **GORM** - Used as an ORM library for Golang.
  - **bcrypt** - Package used for hashing the stored passwords.
  - **Prometheus client** - Used for creating metrics that are tracked and displayed on the endpoint "metrics".
  - **GoDotEnv** - Used for storing "secret" variables.
- **HTML** - Markup language used for displaying data on the frontend, following the Go Template Syntax.
- **CSS** - Styling language to make our displayed data look nice.
- **PostgreSQL** - Query language to store our data in a database.

Monitoring Setup:

- **Prometheus** - Used to create monitor metrics of our systems. These include the number of requests to our endpoints and the status of our system (i.e. is our system up).
- **Loki** - The "Loki driver" plugin is used as our logger. It collects all logs from the MiniTwit Docker container and aggregates them.
- **Grafana** - Used to display our dashboards. Prometheus and Loki are added as data sources here, and we have dashboards configured to display both Loki logs and Prometheus metrics.

Infrastructure:

- **Docker** - Builds our application.
- **Docker swarm** - Management and scaling of Docker containers.
- **DigitalOcean** - Our choice of cloud service provider.

CI/CD:

- **Vagrant** - Script for our Continuous Deployment to Digital Ocean.
- **GitHub actions** - Pipeline for Continuous Integration.

  - **Cypress** - UI/end-to-end tests, automatically run in the pipeline.
  - **GitHub actions for Docker**
  - **appleboy/ssh-action@v0.1.8** - Executing remote ssh commands.
- **SonarCloud** - Checking code quality.
- **CodeClimate** - Also checking code quality.

Development:

- **GitHub** - Storing the project. Uses Git as version control.
- **Python** - For simulator tests.
- **Docker compose** - Building the application with one command.
- **OWASP ZAP** - Pen-testing our website.

Documentation:

- **LaTeX** - Writing this report.
- **Markdown** - Lightweight way of writing documents, used e.g. in our README and SLA.
- **ChatGPT** - Helped us get quick answers.

2.3 Important Interactions of Subsystems

As shown in Fig. 1 the MiniTwit app is dependent on the database and Grafana, which are both separate containers. Images for both Prometheus and Grafana are configured in the *docker-compose.yml.* Both have a specified port and volumes pointing to corresponding configuration files. Within the Prometheus configuration, we have set up to scrape data from MiniTwit.

Grafana has a volume built on the contents of the directory **provisioning** in our repository. Within this is a yaml containing a list of data sources for Grafana. Here both Prometheus and Loki are added as data sources.

Loki receives the logs from the MiniTwit app.

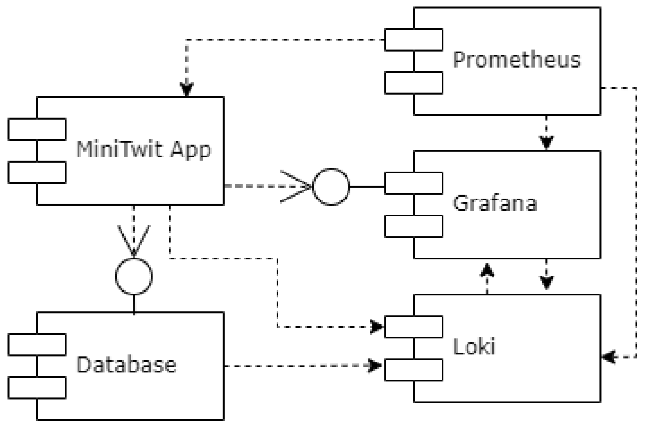


Figure 1: Component diagram showing subsystem interaction.

2.4 The Current State of Our Systems

From a purely transcendental point of view, the quality of the code is subpar.

It is especially the *reliability* of the system which is questionable, with a large amount of downtime caused by an unknown error (this is expanded on in 3.5). Static code analysis, shown in Fig. 2 does not, however, detect many large reliability issues.

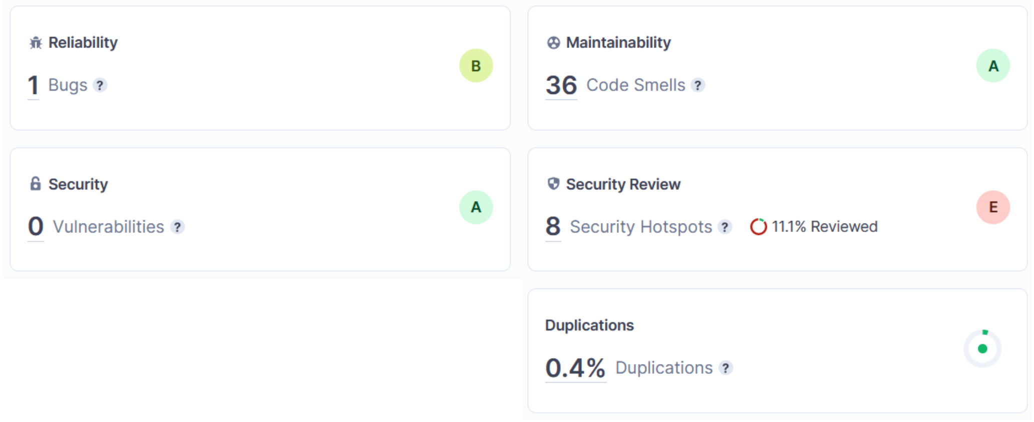


Figure 2: SonarCloud scan results of the development branch.

As shown in Fig. 2, static code analysis indicates that the code is *maintainable*. The low complexity of the system also reduces the threshold of when the system is unmaintainable. Individual files are generally shown to be maintainable, as shown in Fig. 7b.

The system is *portable* to a certain extent. It is dockerized and can be deployed on a docker swarm on any machine(s) with docker. The current Terraform deploy script does, however, rely on specifically using DigitalOcean as IaaS provider.

2.5 Chosen Licence Compatibility with Direct Dependencies

We chose the MIT License for our project and to ensure compatibility with our direct code dependencies we checked the licenses for each package we used (Table 2).

Language	Package	License
Go	Gin	MIT
Go	GORM	MIT
Go	crypto (bcrypt)	BSD 3-Clause
Go	Prometheus client	Apache License 2.0
Go	GoDotEnv	MIT
Go	testify	MIT
Python	requests	Apache License 2.0

Table 2: Code dependencies and their licenses

The MIT, BSD 3-Clause, and Apache License are compatible with each other as they are permissive and unlike copyleft licenses such as the GNU General Public License (GPL), do not require modifications and derivative works to be distributed under the same copyleft license.

3 Process' Perspective

3.1 Team Interaction and Organization

Meeting times

We treated each week as a "sprint" where we would mainly work asynchronously. Once per week, three hours before lecture start, we would meet up at ITU to synchronize on work that had been done from last week and assign new tasks. We kept this going consistently throughout the whole course with few exceptions, such as in times when the workload would be especially high. In which case, we would have a meeting mid-week.
The mediums used for interaction were:

- Discord server - messages and voice calls
- Weekly in-person meetings at ITU
- GitHub project board for organizing tasks/issues

Work delegation

Tasks would be created from users, taken from weekly course tasks, or created internally based on the status of implementing different tools or functionality.

Task delegation was not the ideal scenario for us. As all of us were new to the concepts introduced throughout the course, we were not able to pair up someone experienced with someone inexperienced. Therefore, we would typically try to assign two people to a task, but as technical debt and workload began piling up, we sometimes had to assign one person to a task.

3.2 CI/CD chains

The Continuous Integration/Continuous Deployment (CI/CD) process in our MiniTwit system is implemented using GitHub Actions. The CI/CD workflow is designed to automate the integration, building, testing, and deployment of software to improve productivity and ensure reliability.

The CI workflow is initiated on the creation of a pull request to main/development, or on push to main/development.

The CD workflow is initiated on push to main. It is made to work with an older version of our application (one that does not have scaling or load balancing).

CI workflow - testing workflow - cypress tests

The cypress tests are end-to-end tests that simulate a browser.

- Checkout
- Create .env file with necessary environment variables
- Setup go
- Run the MiniTwit application using go, with an in-memory database
- Run cypress tests

CI workflow - testing workflow - sim API tests

The sim API tests test the simulator API. They are the same tests as were provided to us when the simulator first started.

- Checkout
- Create .env file with necessary environment variables
- Setup go
- Sleep for two minutes
- Run the MiniTwit application using go, with an in-memory database
- Install pip and pytest
- Do a health check on MiniTwit
- Run sim-API-tests with pytest

CI workflow - static code analysis workflow - shellcheck

Shellcheck analysis scans our shell files for problems and errors.

- Checkout
- Run Shellcheck action

CI workflow - static code analysis workflow - govulncheck

The vulnerability check scans our go-files for possible vulnerabilities.

- Checkout
- Run govulncheck action

CI workflow - static code analysis workflow - other

CodeClimate and SonarCloud are also used for static code analysis, but do not exist in our workflow files.

CD workflow

- Checkout
- Log in to Docker Hub - credentials stored in github secrets
- Build and push the MiniTwit web application (push to Docker hub)
- Deploy to server using ssh. The remote script 'deploy.sh' is run

After a new push to main has happened, a release is made manually.

3.3 Organization of Our Repository

Our repository is set up is as a mono-repository with all of our services. As most of our services are containerised and the specification for the service is defined within our docker-compose.yml file, we had no issues with having directories for each specific service.

3.4 Applied Branching Strategy

For this project we decided to go with two standard branches; one for each environment. The **development** branch is for the development environment and **main** is for the production environment.

The **development** branch is for testing our code. Here is where every newly developed feature is merged as per default when creating a pull request. This would allow us to discover if any new features would introduce unforeseen bugs. It is also the branch that every new feature should branch out of.

The **main** branch is our production environment. This is our ”live” code and is what the current state of our product looks like. This is also where we define what is ready to use for our users.

Whenever a new feature is to be initiated, this should happen in a new branch. This new branch should be named after the new feature. Following this naming convention allows other developers to keep track of what each branch is for.

This was all agreed on internally but later decided to be explicit, so the “CONTRIBUTING.md” file was pulled into our repository [2].

3.4.1 Development Process and Tools

Our team used a variety of software development tools and methodologies to ensure efficient collaboration, project organisation, and task tracking.

GitHub Issues and Projects

We used GitHub Issues to record and track all tasks, improvements, and bugs associated with our project. This made it easier for our group to keep track of what needed to be done, as well as assign tasks. We set up a Kanban board within GitHub Projects where we kept the backlog of all issues and their current state. The states we defined are as follows:

New - Tasks identified but not fully defined or prioritized.

Ready - Tasks with clear acceptance criteria and ready to be worked on.

In Progress - Tasks under active development.

Review - Completed tasks to be reviewed by another developer.

Done - Tasks that have passed all acceptance criteria.

This system made it easier for us to keep track of which stage each task was at and served as a visual cue of our workload.

Discord

Discord was our chosen platform for remote communication. We used it for pair-programming collaboration, helping each other when we had any issues and planning next steps.

We found that these tools helped us maintain transparency, encouraged collaboration, and ensured that every team member was kept in the loop regarding the project’s progression.

3.5 Monitoring

Metrics we monitor:

- Number of Get requests to all MiniTwit endpoints
- Number of Get requests to all simulator endpoints
- If the MiniTwit app is up/down
- If Prometheus is up/down

Early on we were at the manual stage of monitoring [1]. As shown in Fig. 3 we experienced downtime twice, both were a consequence of not having a monitoring system in place. We only learned that the system was down by manually checking, and subsequently fixed the issue each time.

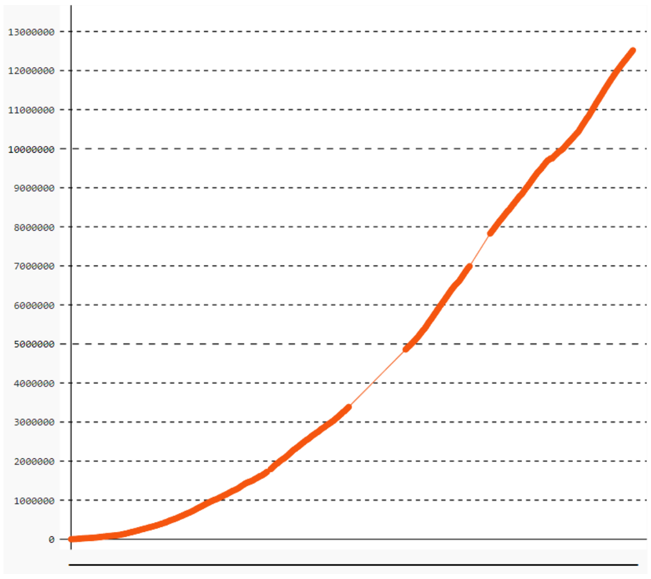


Figure 3: A plot showing the latest IDs sent and reported by the simulator over the course of eight weeks. The two thin spots in the line indicate the two times our system was down. The x-axis is the duration of the simulator, and the y-axis is the ID.

The first metrics we set up for monitoring were the number of GET requests to each of our endpoints using Prometheus. This data collected for monitoring is viewable on the "/metrics" endpoint. The data is also connected to Grafana and configured with a dashboard.

We automated the process of setting up this dashboard when booting up our system. To do this, we downloaded the configuration file of the dashboard as a JSON, and saved it to the **Grafana** folder in our repository. Since Prometheus is added as a data source to Grafana, as described in section 1.3, when booting up our system with the *docker-compose.yml* the Grafana container will point to the configuration file and load Grafana using our downloaded dashboard template.

To improve our ability to be reactive, we used the built-in Prometheus metric "up" which will track if the MiniTwit app and Prometheus are down. The corresponding panel was added to our dashboard, as shown in Fig. 4. This still requires manually checking the dashboard. Therefore, the next step would be to set up an alert to notify the team when the system is down.

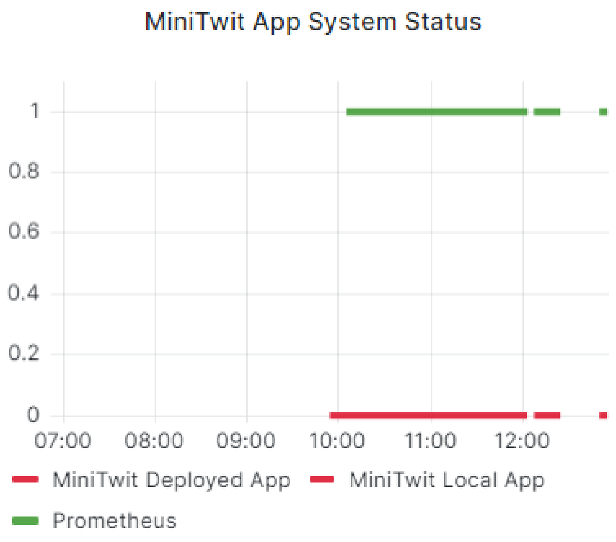


Figure 4: A screenshot showing the panel on our dashboard that shows the system status of our application.

Throughout the project, we considered monitoring different metrics. We initially monitored the current load of the CPU in percent but did not find the information relevant. An additional metric we would like to track is the load time for each endpoint to improve our response time to any latency within our systems. From a business perspective, we would like to assess the activity level in the application based on user behaviors by monitoring the influx of users, the frequency of user tweets, and the interactions between users.

3.6 Logging

We implemented logging to document incoming HTTP requests, database queries, and application-produced warnings or errors. To centralize and manage these, we used Grafana Loki. The Loki plugin was installed in the DigitalOcean droplet that contained our MiniTwit application and was then specified as the logging driver for the MiniTwit container in our *docker-compose.yml*. The first time we ran this we logged into Grafana, connected the Loki data source, and configured a dashboard to display the logs.

We automated the process of creating the log dashboard in a similar manner to how we setup the monitoring dashboard. We saved the JSON containing the log dashboard configuration within **grafana/-dashboards** in our repository. The respective data source configuration was also appended to the automatic.yaml file containing the list of data sources within the Grafana directory.

In terms of further improvements, we would like to incorporate performance-related logging metrics such as total API responses and RAM usage. This would allow us to understand our system's load-handling efficiency and highlight anything requiring adjustment.

3.7 Security Assessment

We started our security assessment by identifying threats against our website.

Identify Threat sources	Construct Risk scenarios
SSH into droplet	Attacker knows IP address, computer name for a droplet, and can add their own private key to gain access.
SQL injection	Attacker performs SQL injection on web application to download sensitive user data.
Hacking DigitalOcean account	Attacker using brute force or some other way to hack into our Digital Ocean account, giving them access to our whole application.
XSS Attack	XSS attack into HTML forms to inject malicious JavaScript.
Connect to PostgreSQL remotely	Attacker uses PSQL to access our stuff.
Access .env file in webserver container	Hacker accesses the .env file inside the webserver docker container (which contains credentials to the database).
DDoS attack	Attacker uses a DDoS attack, causing application to shutdown.

Table 3: Identified risks and set up scenarios for each risk.

After identifying the risks, we analyzed the potential impact if they were to happen. We visualized this with a Risk Matrix, as shown in Fig. 5.

Risk Matrix

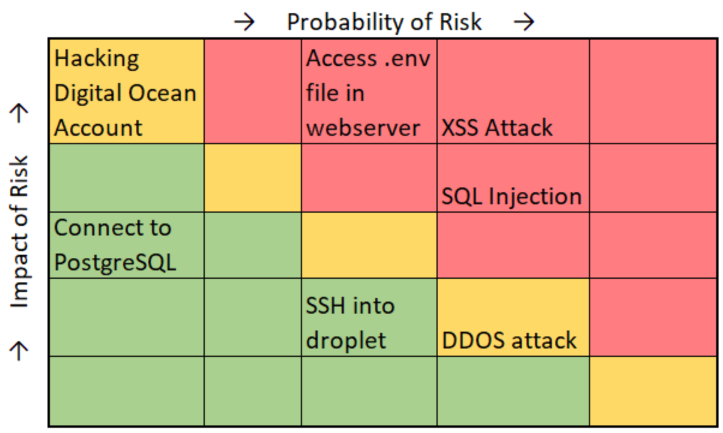


Figure 5: Risk Assessment

Vulnerability scanning

We used OWASP ZAP to scan our system for vulnerabilities. The results from the scan showed us that we had several threats against our website, including SQL injections and XSS attacks as we had predicted from our analysis. After digging into the threats that OWASP ZAP was reporting, we tried to replicate them manually on the site without any success.

We researched how to avoid e.g. SQL injections in the Gin framework, and learned that we were already doing what we could to prevent it, in the backend. It could be that the scan reported this vulnerability due to the fact that you can create an account with symbols like '-' and ';'.

Vulnerability fixing

A vulnerability from our early analysis that we decided to get rid of, was the access to our .env files in the web server.

3.8 Applied Strategy for Scaling and Load Balancing

Scaling and load balancing are newer additions to the system and are not a part of the development branch or main branch. Auto-scaling is not a feature of the system. Our scaling/load balancing implementation is based on the example repository from the exercise session.

Scaling is done via Docker swarm, with which services can easily be replicated across multiple nodes (droplets, in our case). Each node runs an instance of the Docker engine, thus allowing the Docker swarm to spread the workload across multiple droplets. Load-balancing is achieved with Nginx configured to route traffic to any of the nodes.

Our default configuration with Docker swarm has the MiniTwit service replicated nine times, the load balancer replicated twice, with the rest of the services only having one replica. Figure 6 shows an example of interactions between nodes in our swarm cluster.

There are three manager nodes (one of which is a leader), and three worker nodes. The manager nodes orchestrate the cluster and delegate work to worker nodes. The 'leader' node's IP-address is the main entrypoint for the user.

3.9 Using AI Assistants

We used ChatGPT [3] to generate some rough drafts and to help write the SLA. For the SLA use case, it proved very useful - as the specific tone used in judicial documents can be time-consuming to recreate accurately. The same goes for the structure of the SLA.

Due to a lack of sufficient examples in documentation we attempted to use ChatGPT for examples of implementing "cookies" using the Gin framework. We asked for examples of code, but did not end up using it. Therefore we do not know if this is a good use case.

Used DALL-E [4] to generate our mascot.

4 Lessons Learned Perspective

4.1 Evolution and Refactoring

Our group tends to be overambitious. For example, we felt quite optimistic about refactoring to a new language with a new web framework. This came into play at other points during development, especially at the beginning where we would view tasks as the complete feature that we have in mind. We learned quickly to break tasks down into smaller, manageable tasks that could be completed and implemented in a short period of time.

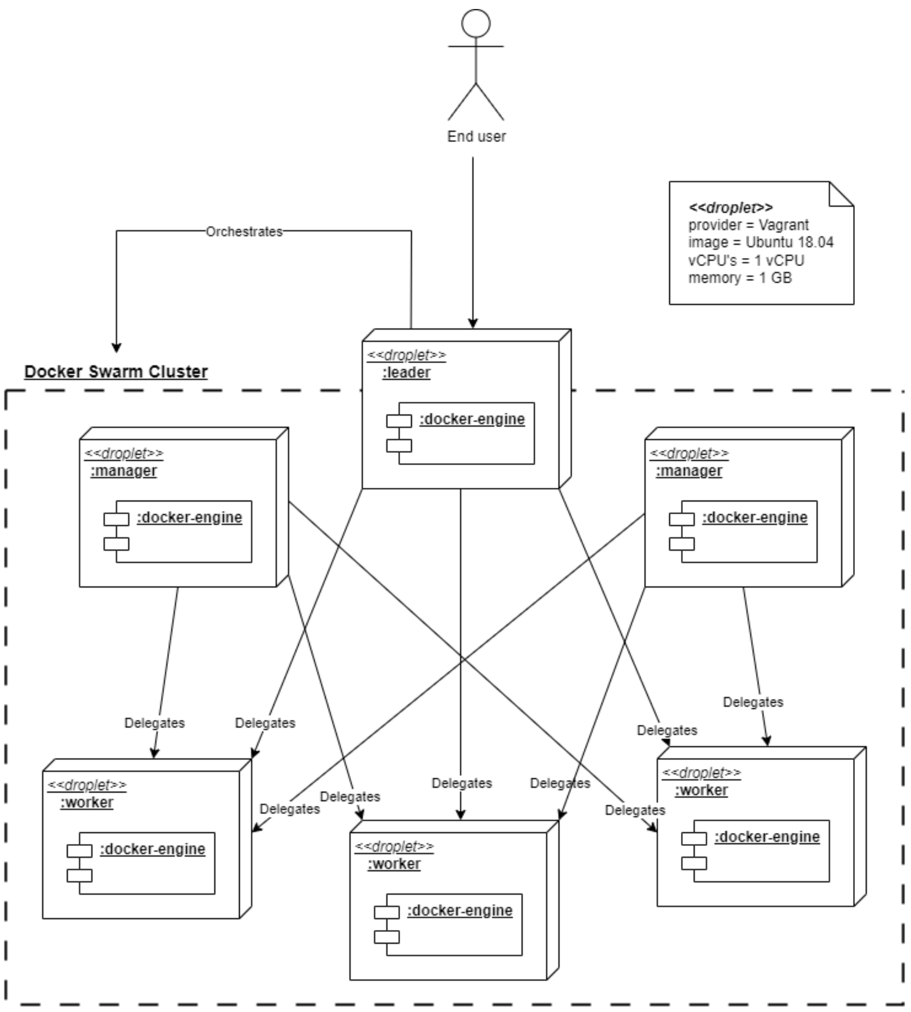


Figure 6: Interactions between nodes in our swarm cluster.

4.2 Operation

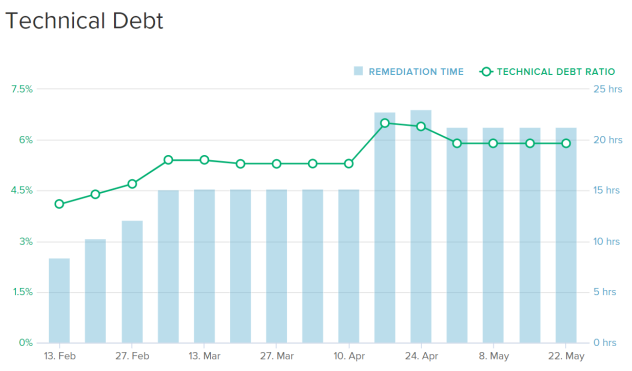
Delivering consistently every week and to a high level of quality was challenging. This was particularly true when we were dealing with a feature or tool that was unfamiliar to all group members, leaving us without a go-to expert within the team. Despite the struggle, it is clear that we have gained a lot of knowledge. While we may not have achieved expertise in every single tool, we're confident in our ability to implement and use these tools in the future to expand our technical proficiency. Embracing the struggle, as it were.

4.3 Maintenance

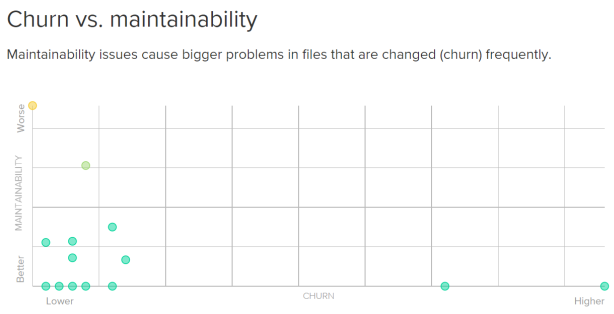
Throughout the course we accumulated a fairly high technical debt, which increased the cost of maintenance. This is shown in Fig. 7a. Due to pressing deadlines we frequently adopted a "good enough" approach, with intentions to return and improve our work at a later stage. However, that later stage seldom arrived with the influx of new tasks, issues, and feature developments. In hindsight, it would have been a better approach to invest more time upfront to ensure each task was completed to a higher standard the first time.

4.4 Our "DevOps" Style

While we did not master the "three ways" of DevOps, our work style was much adjusted compared to how we worked in previous development projects for other courses. Take the *flow* way, for instance; the end goal of each task was not just to implement it, but to implement it in a way that supported



(a) Technical debt



(b) Churn vs. maintainability

Figure 7: Maintainability graphs from Code Climate

CI/CD. We made this a priority. As an example, not only did we implement monitoring, but we made sure that any time the system is booted up again Prometheus is scraping the data to feed to the dashboards that are already configured on Grafana. Previous development projects felt more focused on features, but not the continuous integration of code when it is live. While adjusting to this may have increased time in the beginning, it greatly reduced time spent on integration and deployment when rapidly developing new features or implementing new tools.

Regarding the 2nd way, *feedback*, we adhered to peer review of changes and automated testing. Prior development would usually entail manual testing and no requirements as to who would review changes.

While we have not become masters of DevOps, we earnestly tried to adapt not only our ways but our perspectives. The insights we have gained and the benefits we have seen fuel our drive to learn and inspire us to jump in, take risks, and learn from failure.

References

[1] J. Turnbull. “A Monitoring Maturity Model.” (2015), [Online]. Available: https://www.kartar.net/2015/01/a-monitoring-maturity-model/ (visited on 2023).

[2] ContainerMaintainers. “CONTRIBUTE.md.” (2023), [Online]. Available: https://github.com/ContainerMaintainers/MiniTwit-Golang/blob/development/CONTRIBUTE.md (visited on 2023).

[3] OpenAI. “ChatGPT.” (), [Online]. Available: https://openai.com/blog/chatgpt (visited on 2023).

[4] OpenAI. “Dall-E.” (), [Online]. Available: https://openai.com/product/dall-e-2 (visited on 2023).