IT UNIVERSITY OF COPENHAGEN

DevOps, Software Evolution and Software Maintenance (Spring 2023)

Course Code: KSDSESM1KU

Exam project: MiniTwit

Students:
Gustav Vilain Brygger - gubr@itu.dk
Niklas Brynfeldt - nbry@itu.dk
Bjarke Dalhoff Christensen - bjch@itu.dk
Oliver Simon Jarvis - ojar@itu.dk
Amanda Rasille Røn Volf - amav@itu.dk

May 23, 2023

Contents

1 Introduction **1**

2 System's Perspective **2**
2.1 Design and Architecture of MiniTwit 2
2.1.1 Application, Persistence, and Web 2
2.2 Dependencies of MiniTwit system 3
2.2.1 Arguments for programming language and framework 3
2.2.2 Arguments for virtualization techniques and deployment targets 4
2.2.3 Arguments for ORM, DBMS 4
2.2.4 Arguments for Terraform 4
2.3 Important Interactions of Subsystems 5
2.4 Current state of the system 5
2.5 Project License 5

3 Process' Perspective **6**
3.1 Developer interactions 6
3.2 Team organization 6
3.3 CI/CD chains 6
3.4 Organization of Repository 6
3.5 Branching strategy 7
3.6 Development process and tools 7
3.7 Monitoring 7
3.8 Logging 8
3.9 Security assessment 8
3.10 Scaling and load balancing 8
3.11 AI-assistants 8

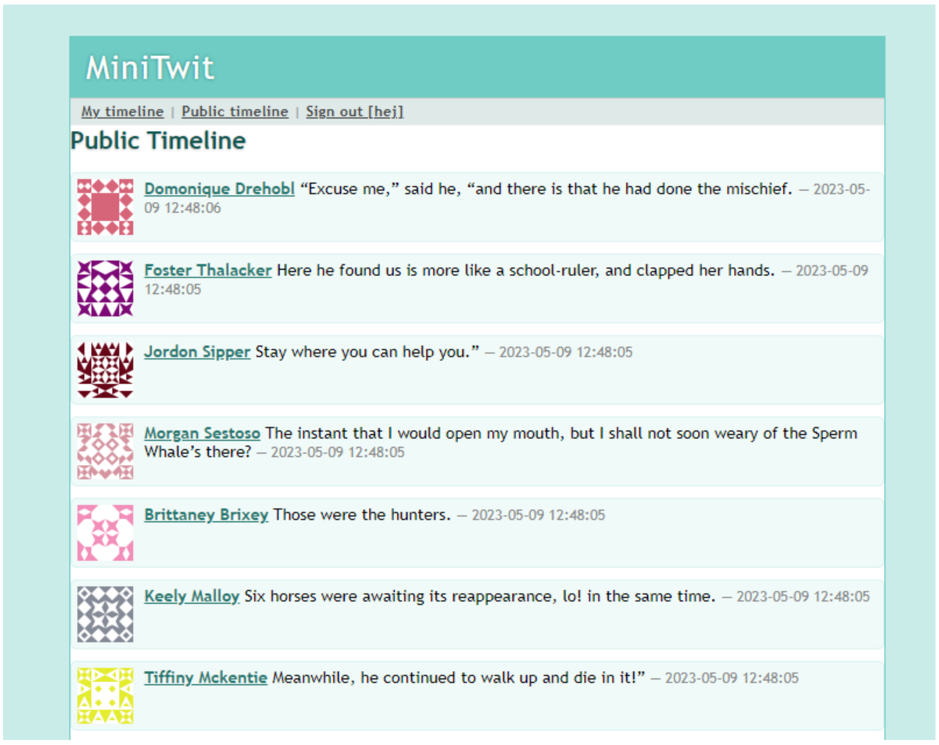
4 Lessons Learned Perspective **9**
4.1 Evolution and Refactoring 9
4.2 Operation 9
4.2.1 Infrastructure as code 9
4.2.2 Scaling and load balancing 9
4.2.3 Importing technical debt 9
4.3 Maintenance 9
4.4 DevOps style of work 10

A Docker Swarm hardware and container setup **11**

1 Introduction

This report will contain a systematic description and illustration of the Go application 'MiniTwit'. The application was developed as part of the course DevOps, Software Evolution and Software Maintenance (Spring 2023).

The functionality of the application was given in a similar application written in Python and Flask provided as part of the course material. This application was outdated and the structure was suboptimal. Thus, we rewrote this application to a more maintainable system. Our developed application was able to handle registering users, login/logout, posting messages, and follow/unfollow functionality. The application consisted of a user interface, as seen in figure 1, as well as endpoints used by a simulator throughout the course to mimic real users' interactions with our application.



Public Timeline

Domonique Drehobl "Excuse me," said he, "and there is that he had done the mischief. — 2023-05-09 12:48:06

Foster Thalacker Here he found us is more like a school-ruler, and clapped her hands. — 2023-05-09 12:48:05

Jordon Sipper Stay where you can help you." — 2023-05-09 12:48:05

Morgan Sestoso The instant that I would open my mouth, but I shall not soon weary of the Sperm Whale's there? — 2023-05-09 12:48:05

Brittaney Brixey Those were the hunters. — 2023-05-09 12:48:05

Keely Malloy Six horses were awaiting its reappearance, lo! in the same time. — 2023-05-09 12:48:05

Tiffiny Mckentie Meanwhile, he continued to walk up and die in it!" — 2023-05-09 12:48:05

Figure 1: MiniTwit Application Frontend

If any errors were detected by the simulator, for instance, if the wrong response was received or if the system responded too slowly, these would be logged to allow us to handle them appropriately.

The rest of this report will explain the application from different perspectives and finally reflect upon what we learned throughout the project.

2 System's Perspective

2.1 Design and Architecture of MiniTwit

The MiniTwit application architecture has been designed following software design guidelines such as low coupling, high cohesion, and the Single Responsibility Principle (SRP). These concepts are closely related and helped us create software that is easier to work with. The SRP states that each part of a software system should be responsible for one thing and therefore only have one reason to change[1]. Cohesion is closely related to the SRP and it is a measure of the degree to which related functionality is grouped in a single coherent unit[2]. In general, the goal is to have modules with high cohesion as this ensures that a module's code is focused on a single aspect of the application's functionality. This is vital as too much functionality in a single module can result in code that is hard to understand and change. Finally, coupling refers to the degree of interdependence between the modules within a system and it measures how well-defined the boundaries between different modules are. Low coupling implies that modules' knowledge of other modules' internal implementation details has been minimized[3]. Together the SRP, low coupling, and high cohesion are key enablers when it comes to reducing complexity and thereby ensuring a strong foundation for designing extendable software.

To achieve low coupling and high cohesion the MiniTwit application has been split into three layers namely persistence, web, and application as seen in figure 2. As prescribed by the SRP each layer is responsible for one thing and we have minimized the dependencies between the layers to make MiniTwit extendable. This also reduces cognitive load as it is easier to navigate a system where related functionality is grouped.

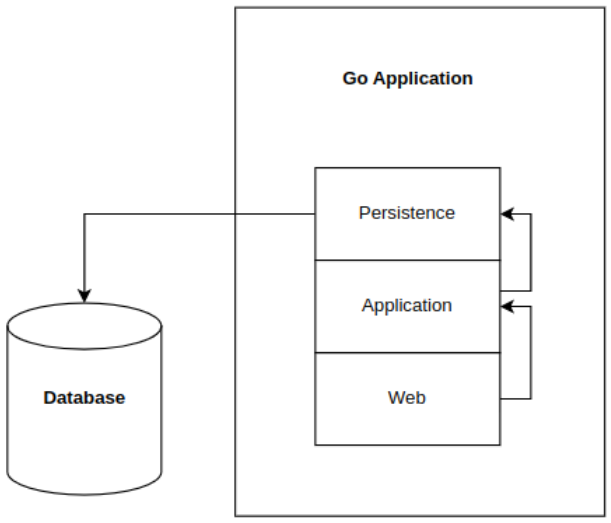


Figure 2: MiniTwit Architecture

2.1.1 Application, Persistence, and Web

In the application layer, all code related to core business logic is found. Business logic is separated from other parts of the application as doing so allows developers to understand and modify business logic without impacting other components of the application. Secondly, separating business logic promotes reusability, as the code can easily be shared across different parts of the application.

In the persistence layer, the database setup is configured. We use an Object Relational Mapper (ORM) called Gorm to interact with the database as it abstracts away some of the complexity that

[1]https://blog.cleancoder.com/uncle-bob/2014/05/08/SingleReponsibilityPrinciple.html
[2]https://www.codurance.com/publications/software-creation/2016/03/03/cohesion-cornerstone-software-design
[3]https://www.martinfowler.com/ieeeSoftware/coupling.pdf

comes with a relational database. The persistence layer also handles migrations and data seed.

The web layer exposes our API and is implemented using a lightweight Go web framework called Gin. Gin routes requests to controllers which coordinate the request/response cycle including session management and validating. When request processing has finished HTML is generated and returned to the client.

2.2 Dependencies of MiniTwit system

For the MiniTwit application to run, we are depending on first and foremost Digital Ocean whose servers the application is hosted on. The goal was to run our MiniTwit application on 7 servers using the following 9 containers:

- The Database based on the latest Postgres image is responsible for holding all the users' data
- The MiniTwit application's server is based on the latest Golang image
- A Redis instance for storing sessions and the integer 'latest'
- Elasticsearch for indexing logs such that we can search and analyze them
- Filebeat for collecting and forwarding the logging to Elasticsearch
- Kibana for visualizing the results from Elasticsearch
- Prometheus for monitoring the application in terms of CPU usage and request time
- Grafana for visualizing the monitoring
- Nginx for load balancing the incoming traffic to the different services

From within each of these containers, we depend on a large number of libraries of which the major ones we consider to be:

- Gorm used for object-relational mapping allowing us to perform CRUD operations in Go.
- Gin used for routing the HTTP requests.
- x/crypto/bcrypt library for hashing the user password.

For initializing the containers we used Docker Compose and Docker Swarm. Hence we also depend on Docker, Docker Compose and Terraform which we intended to use to provision all the servers and perform the following tasks:

- Connect to docker swarm nodes via token
- Manage ssh keys on nodes
- Manage configs and secrets on nodes

In a more broad sense, the application also depends on Go, Ubuntu, Python, and SSH protocol to work. At last we also depend on pre-build git workflows such as *docker/build-push-action@v2* or *rymndhng/release-on-push-action@master*.

2.2.1 Arguments for programming language and framework

We initially set out to use FastAPI in Python but eventually settled on Go for multiple reasons. From a personal development perspective, learning a new language is rarely a poor choice. Go is well-known for being relatively fast to learn, whilst maintaining speed. From a more technical perspective, Go has advantages over many languages. It's fast, it has concurrency built-in, and it has strong standard library support, with the most important features, even templating, being available as a standard package.

We chose Gin as our web framework as it offers a minimalistic and fast approach to building web applications. It also provides essential features like routing, middleware support, JSON handling, and request/response binding. All features that would have taken a long time to implement from scratch. Gin is well-maintained, has over twice the amount of GitHub stars than the next most popular Go web framework, and is one of the faster Go web frameworks [4]. Whilst implementing the application in FastAPI might have been faster and easier, due to both prior Python knowledge, and general simplicity, the speed and features provided by Go are a clear advantage.

2.2.2 Arguments for virtualization techniques and deployment targets

Docker was chosen because it's the industry standard. Other container strategies are also valid, but given knowledge of the need for orchestration (in either Docker Swarm or Kubernetes) Docker was an obvious choice.

We used Digital Ocean mainly for budgetary reasons and for ease of deployment. Using the GitHub Student Starter pack, $200 in free credits were made available to us. This was enough to ensure that the application was able to run for the duration of the course. It allowed us to test out different deployment strategies going from single-server and containerized applications being spun up with only Docker Compose to more advanced strategies such as running multiple servers and services to minimize downtime.

Using multiple accounts furthermore allowed us to create a parallel Digital Ocean setup for our development branch. This was useful to test our CI/CD pipeline without interfering with the production environment.

2.2.3 Arguments for ORM, DBMS

In regards to choosing Object-Relational Mapping (ORM) and Database Management System (DBMS), we settled on using Postgres as our DBMS and Gorm for ORM. In terms of choosing DBMS, we considered SQLite and a Digital Ocean managed Postgres instance. Postgres provides better scalability, concurrency, and ability to handle user levels (e.g. admin vs user) compared to SQLite.
While we could have deployed a Postgres using Digital Ocean Managed Databases, we determined that for our needs it made better sense to deploy a database ourselves. This was done for cost reasons ($7/month for the smallest droplet vs. $15/month for the cheapest Digital Ocean database), but also to ensure that we had full control over the database.

For ORM we considered XORM and Beego but due to our limited experience with Go, we choose Gorm since it is widely used, well documented, and has better superior features like sanitizing input[5].

2.2.4 Arguments for Terraform

We intended to use Terraform to store our infrastructure as code. Terraform is the industry standard, and storing our infrastructure as code will help generalize and speed up our provisioning.

[4] https://web-frameworks-benchmark.netlify.app/compare?f=gin,jester,kemal,phoenix,sinatra,fastapi

[5] https://blog.logrocket.com/comparing-orm-packages-go/&https://sumit-agarwal.medium.com/gorm-vs-xorm-part-1-d156ba9de404

2.3 Important Interactions of Subsystems

In the image below the interactions between our subsystems can be seen:

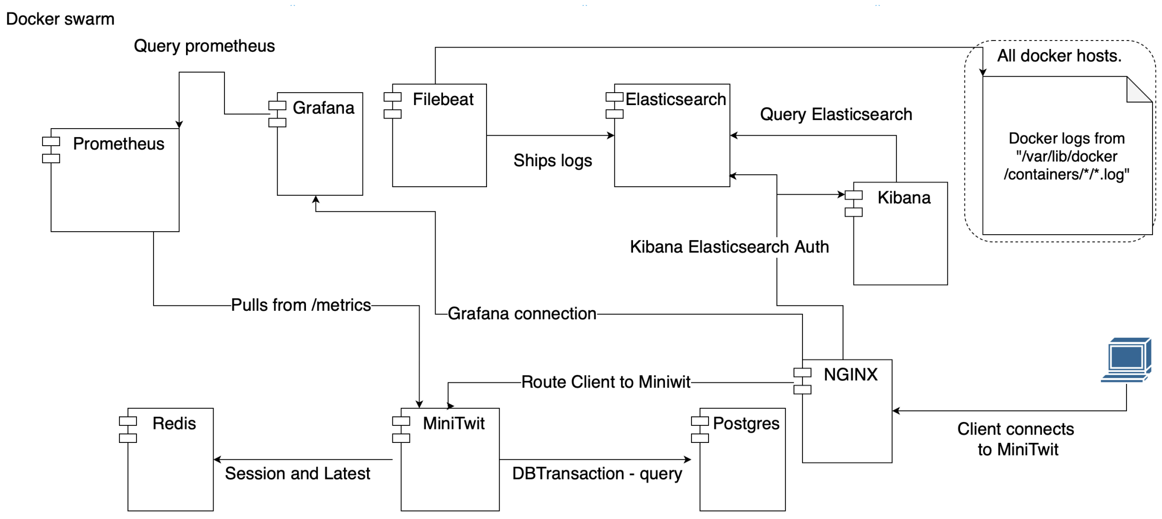


Figure 3: Subsystem interactions

Here we mainly focus on how a client and our systems interact with one another in our intended design of the Docker Swarm. The different hardware components and how the services are supposed to be deployed in the Swarm Cluster can be seen in appendix A.

2.4 Current state of the system

In the current state of the system, our stack consists of:

- Application running on Digital Ocean
- Run tests in CI/CD pipeline within an isolated environment
- Monitoring the application with Prometheus and Grafana
- Code quality checks using Golangci-lint in the CI/CD pipeline checking for typecheck, gosimple, unused, etc.
- Logging using Elasticsearch and Kibana

We did not manage to get Docker Swarm and Redis up and running. Docker Swarm has yet to be set up in our CI/CD pipeline and Redis seemed to be running locally but a bug in production seemed to cause Redis to malfunction.

2.5 Project License

To choose a license, we needed to know what type of software we were going to use, i.e. what packages and their given licenses.

We used the Go library 'go-licenses' to try and determine what packages we used within our Go application [6]. Then we looked at all major pieces of software and frameworks and determined they all used either MIT, Apache, or BSD Clause 3.
We determined that an Apache 2.0 License was compatible with our dependencies and our desire to make the project open source [7].

[6]https://github.com/google/go-licenses
[7]https://ghinda.com/blog/opensource/2020/open-source-licenses-apache-mit-bsd.html

3 Process' Perspective

3.1 Developer interactions

Throughout the development of the project, the team has interacted primarily through Discord. Further, the team met physically on Tuesdays.

3.2 Team organization

We organized ourselves as a cross-functional team. Sometimes we worked on tasks individually but for the majority of the project we worked together, typically in pairs. Every opinion and input from each team member was acknowledged equally.

3.3 CI/CD chains

Our CI/CD chain consists of the following GitHub Actions workflows:

- Continuous-deployment - Builds Docker image, pushes it to the registry, SSH onto the server, and runs the deployment script.
- Dev - Functions as the Continuous-deployment workflow but deploys to a test environment similar to production.
- Test - Runs the test Docker compose file which creates a closed environment with the code on the current branch and runs the test.
- Main Release - This uses an external action namely *rymndhng/release-on-push-action@master* for creating a release every time a push to main is made.
- Lint - Uses the external action 'golangci/golangci-lint-action@v3' which checks the code with multiple linters.

Every time a pull request was made to a branch, the 'Lint' and 'Test' workflows were run. The workflows that handled deployment were triggered manually. Further, we have set up CodeClimate and SonarCloud to help us maintain an appropriate code standard. The SonarCloud check runs on every pull request targeting our main branch (see section 3.5 for a description of the branching structure).

3.4 Organization of Repository

The project only consists of a single repository containing the application. We chose this because it is a small application and thus splitting it into multiple repositories would have complicated the development process unnecessarily. This single repository contains several folders:

.github/workflows	Contains all our workflow files used with GitHub Actions.
api_test	Includes files to run the simulator and a test of the API. Primarily used during the development of the endpoints.
remote_files	Has all files that are used when deploying to the server. This includes various logging and monitoring files, our deploy script, and the docker-compose file.
report	This folder is where our report files are saved.
src	Contains all our application code which is further divided into subfolders, thereby separating the logic of the system.
tests	Our test files are located in this folder. It contains a Docker file, a docker-compose file, and a test file to run our tests without external impact.

Table 1: Repository folders

3.5 Branching strategy

We used two static branches (main and dev) as part of our branching strategy throughout the development of the project. The main branch was only used for functioning and tested code, and was deployed to production. The dev branch was used for development and testing. Additional branches were created when new functionality was to be implemented, or bugs had to be fixed. All temporary branches were derived from the main branch. An example of our general branching strategy can be seen in figure 4 and 5.

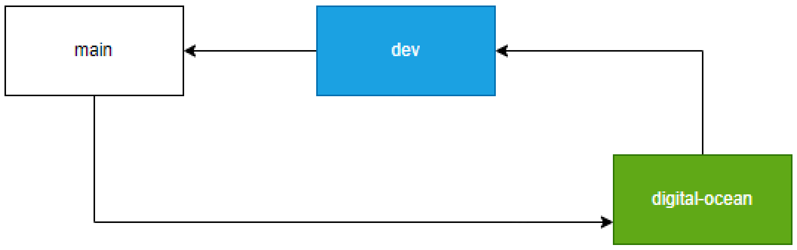


Figure 4: General branching strategy

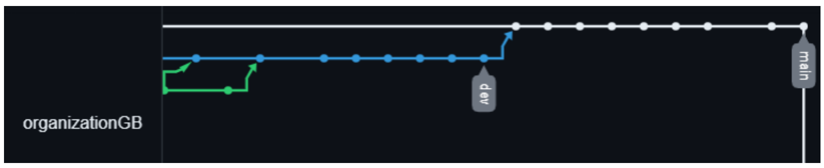


Figure 5: Example use of branching strategy

It can be seen that the temporary branches are derived from the main branch, and when the functionality is implemented, it is merged into the dev branch to be tested before the code is put into production.

3.6 Development process and tools

When meeting on Tuesdays we discussed what needed to be done if we weren't already working on a task. We tried to start using a Kanban board but due to us not establishing clear guidelines on how to use the tool, we didn't manage to successfully implement it. In our development process, we utilized pair programming a lot. This also lead to an iterative development process in that we pushed small changes often to switch programmer. Further, the team agreed that when larger new implementations were to be introduced to the main or dev branch, a pull request should be made to allow team members to review the code.

3.7 Monitoring

The deployed MiniTwit application was monitored using Grafana, which is a dashboard web application for visualization of data. We used Grafana to monitor the simulator's interaction with our application. On our dashboard, we included a visualization of the CPU load percentage as well as an overview of the duration of requests to the different endpoints. Here it was possible to see the metrics for a single endpoint. This allowed us to analyze CPU load spikes, as they might be due to resource-intensive endpoint requests. Additionally, we monitored the total number of responses to assess the availability of the application.

Data was collected for visualization; we used Prometheus, an open-source monitoring system that collects and stores various metrics data from the web server. Prometheus pulls the metric data from the MiniTwit application from a metrics endpoint and stores it so it can be queried by Grafana and shown in the dashboard.

3.8 Logging

Our logging setup is based on Elasticsearch, Kibana, and Filebeat as visualized by figure 6. These open-source tools are all developed by the same company which eases the integration process. Filebeat collects logs from the application containers after which they are forwarded to Elasticsearch. Elasticsearch is a search and analytics engine and here the logs are indexed so they can be queried by Kibana where the logs can be visualized by interactive dashboards. In our setup, Filebeat appends the container name to all logs so they can be filtered and visualized per-container basis.

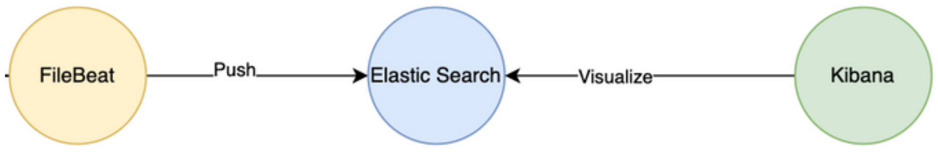


Figure 6: Logging setup

3.9 Security assessment

From our security assessment, we learned that we were most at risk of attacks based on unauthorized access to our codebase and our weak authentication and authorization mechanisms. The first risk was likely to occur as it was possible to find passwords and usernames for the database in our public codebase. We made sure that these were no longer stored directly in the source code by importing them from GitHub secrets instead. The second identified risk could be improved by changing the default passwords to follow security standards.

3.10 Scaling and load balancing

Our choice for scaling strategy was to divide our services into micro-services and deploy them in a Docker Swarm cluster. This meant that we could scale horizontally and vertically. In this design, we had multiple instances of loadbalancers, that would be placed on different nodes. The purpose of having multiple instances of the MiniTwit application services is that we try to ensure up-time if a container on another server node breaks for some unforeseen reason. Our services that were single-points of failure given the design, were Redis, Postgres, and Elasticsearch, the latter only meaning our logging would be down.
Since we used Docker Swarm to orchestrate our micro-services, we would now be able to take advantage of their built-in functions in the update strategy. Our choice was to use a stop-first rolling update strategy, which is mentioned in the Docker Swarm documentation [8]. This aims to achieve a close to zero downtime as we roll out updates for the application containers and allows us to gracefully update each container as we move to a newer version of the application.

3.11 AI-assistants

The team has used AI-assistants to aid in the development and deployment of the MiniTwit application. We used OpenAI's ChatGPT and Phind.
Using these assistants aided us because they provided a quick overview of the possibilities with short and general explanations regarding the technologies we used as well as how to approach development tasks, such as describing the different components used in a docker-compose file or bootstrapping a Go application using the Gin framework. However, using them also hindered us as we often had to double-check whether the information provided was accurate and ended up using more time than if we obtained the information without them.

[8]https://docs.docker.com/engine/swarm/swarm-tutorial/rolling-update/

4 Lessons Learned Perspective

4.1 Evolution and Refactoring

The ideal approach to refactoring the old application would be to analyze, divide and concur the old application into subtasks and define the overall design of the new application and then start the refactoring process. However, we started refactoring without having an idea of how the result/end system should be. Thus, the process became very unstructured. We have learned that this could have been avoided by better planning aided by e.g. a Kanban board and weekly standups. This caused the refactoring process to take longer than anticipated and left us behind schedule. An example can be seen in this commit which is from a branch we ended up not using.

After refactoring the evolution of our application started. Our problem in regards to planning persisted, we did manage to divide tasks but aggregating them afterwards could be challenging both in terms of information sharing and merging code. The introduction of new technologies further complicated this process. Unfinished tasks from the backlog, coupled with a rapid expansion of the stack due to the exercises for the current week, resulted in a fractured team and codebase. This could have been avoided with less time pressure and better planning.

4.2 Operation

4.2.1 Infrastructure as code

We experienced trouble configuring Terraform: due to some latency with Digital Ocean in spinning up containers, we regularly encountered an undebuggable error during provisioning. We eventually solved it by adding sleep commands between the provisioning of different droplets, in order to ensure that they were all up and running before we tried to connect to them.
The lesson here was that Terraform isn't an infallible tool. It's a tool intended to streamline and simplify the provisioning process, but still ultimately depends on many sub-processes which can fail.

4.2.2 Scaling and load balancing

We found Docker Swarm to suffer from confusing terminology and a lack of clear documentation. The documentation is also spread out over multiple different services, such as Docker Compose and Docker Stack.
Nginx was also difficult to set up. Both due to implementation difficulties, but also in conceptually understanding where to have the load balancers (i.e. outside the swarm or inside the swarm).
Kubernetes was presented to be too technical and complicated. But the larger ecosystem and the wider availability of resources concerning Nginx and Kubernetes makes us wonder whether there is a lesson to be learned. Due to bad documentation, Docker Swarm's simpler interface might ultimately not be more user-friendly than the technically challenging, but more widely adopted Kubernetes.

4.2.3 Importing technical debt

During the development of this application, a lot of different tutorials and examples have been used. This has in some cases resulted in using old and deprecated versions of software. As an example, our stack ended up relying on a version of Grafana from 2017, which resulted in errors that were challenging to debug.
This was fixed by upgrading to a newer version of the software.
Other examples include online resources often using old versions of docker-compose.
The lesson here is to use examples and tutorials that are up to date. This was a general problem throughout the development process.

4.3 Maintenance

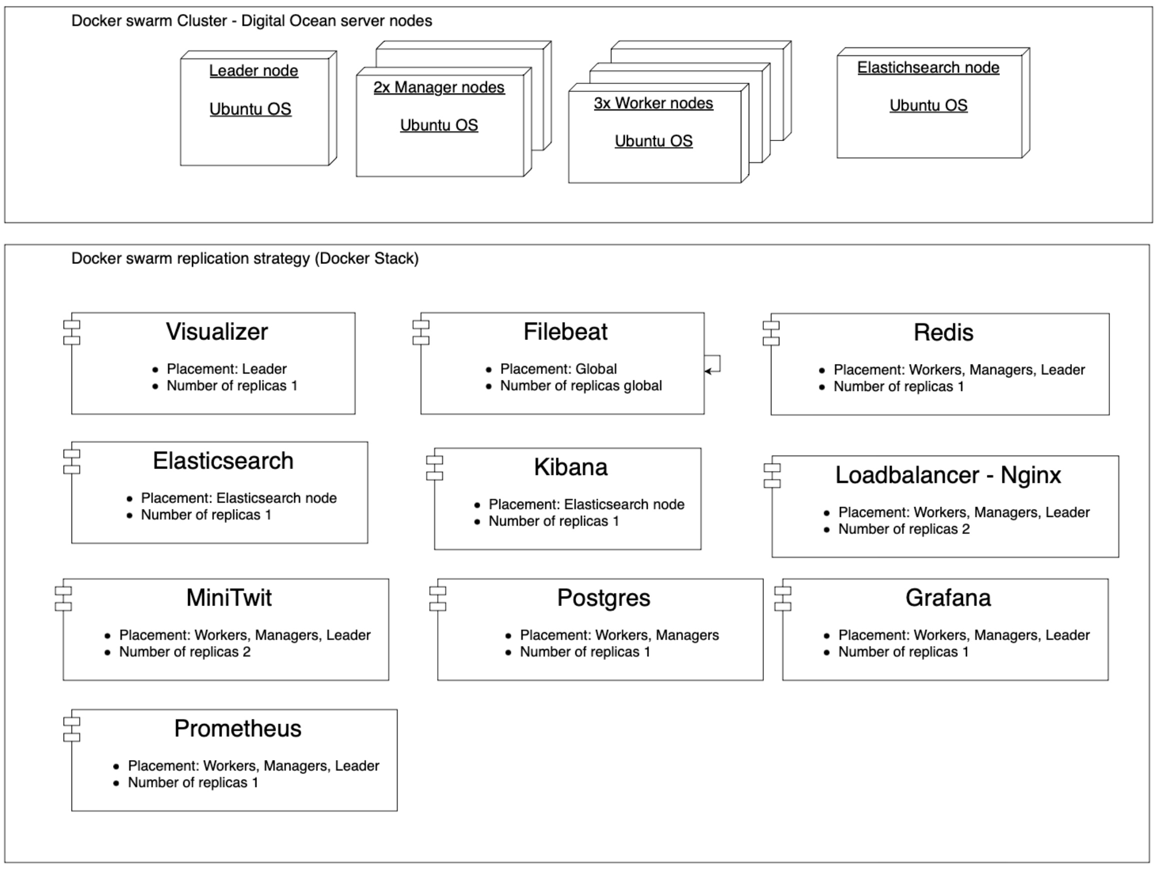
In regards to maintaining the application, we faced issues discovering errors as we did not set up a way for us to get notified of any errors. None of the frameworks and libraries that were used in a final release were deprecated during the project, but issues were created on GitHub by fellow students which we could have handled more formally to ensure that we solved them adequately.

4.4 DevOps style of work

In the process of developing the MiniTwit project, we worked as a team to create an infrastructure that would support a DevOps style of work by enabling us to implement continuous integration and continuous delivery. We strived to deploy features once they were completed and had passed our testing pipeline. This strategy was chosen over deploying several features simultaneously. Our setup enabled us to deploy automatically through GitHub Actions and thus minimizing any manual interference.

Appendices

A Docker Swarm hardware and container setup


Docker swarm Cluster - Digital Ocean server nodes
Leader node
Ubuntu OS
2x Manager nodes
Ubuntu OS
3x Worker nodes
Ubuntu OS
Elastichsearch node
Ubuntu OS
Docker swarm replication strategy (Docker Stack)
Visualizer
Placement: Leader
Number of replicas 1
Filebeat
Placement: Global
Number of replicas global
Redis
Placement: Workers, Managers, Leader
Number of replicas 1
Elasticsearch
Placement: Elasticsearch node
Number of replicas 1
Kibana
Placement: Elasticsearch node
Number of replicas 1
Loadbalancer - Nginx
Placement: Workers, Managers, Leader
Number of replicas 2
MiniTwit
Placement: Workers, Managers, Leader
Number of replicas 2
Postgres
Placement: Workers, Managers
Number of replicas 1
Grafana
Placement: Workers, Managers, Leader
Number of replicas 1
Prometheus
Placement: Workers, Managers, Leader
Number of replicas 1