KSDSESM1KU

DevOps, Software Evolution and Software Maintenance

IT UNIVERSITY OF COPENHAGEN

Llama GeMsT

Project Report

Professors:
Helge Pfeiffer - ropf@itu.dk
Mircea Lungu - mlun@itu.dk

Authors:
Simon Harwick – harw@itu.dk
Mihaela Hristova - mihr@itu.dk
Gustav Henrik Meding – gusm@itu.dk
Tomas Babkine-Di Caprio - tbab@itu.dk
Eduardo Sfreddo Trindade - edtr@itu.dk

May 22, 2024

1 Introduction

The following report describes the work done during the Spring 2024 DevOps, Software Evolution and Software Maintenance taught at the IT University of Copenhagen. Our report includes the description of the final state of our system, the process we followed to develop and deploy it, lessons we learned, and reflections on technologies used and high and low level decisions we made throughout the semester.

2 System's Perspective

In this section, we describe the architecture of our Llama-MiniTwit system, the technologies that we used to support it, and we provide a rationale for the technologies we chose. This section also shows how a front-end or API request is processed by our system and what is eventually outputted. Finally, we show the results we obtained from using static analysis tools.

2.1 Overall Architecture

Figure 1 is a visual representation of our overall architecture. We highlight its hierarchical nature starting from the hardware level on Digital Ocean to our services level installed on Docker containers.

2.2 How a request is processed when it comes from the front-end/simulator

The diagram in figure 2 shows how our system handles a request coming from the front-end or the API. We generally follow a model-view-controller architecture where the main function acts as the entry point of our application. Depending on the request type, the main function then forwards the request to handlers which redirect, handle logic, pass data to HTML templates or pass data to be written/read to/from the database. We chose this architecture as it would allow us to reuse code as much as possible and to seamlessly extend our code base.

For example, since we separated the methods that communicate with the database, we were able to add an abstraction layer without modifying the core logic of the application. Similarly, we were able to migrate our database from SQLite to a managed MySQL database on DigitalOcean very easily since our logic for connecting to the database was contained in one function which was invoked by several other function when they needed to connect to the database.

2.3 Architecture graphs of the system

2.4 Technologies used

This section describes some of the key technologies used. The complete list of dependencies can be found in the appendix.

2.4.1 Programming Languages

We chose to refactor our application using GoLang (Go). We chose to use it to refactor our app due to its simple syntax, automatic garbage collection and very high performance. Because of its easy to approach syntax, GoLang is generally regarded as a simple language to learn compared to others such as C++. For this reason, this made it a clear choice when we all saw the refactor of ITU-MiniTwit as an opportunity for our group members to learn a new language.

The automated tests that were provided to us and that we used to monitor development on our app and make sure that no buggy code made it to our deployed application were written in Python. These were provided to us in the course repo and we chose not to refactor them to GoLang as we were able to use

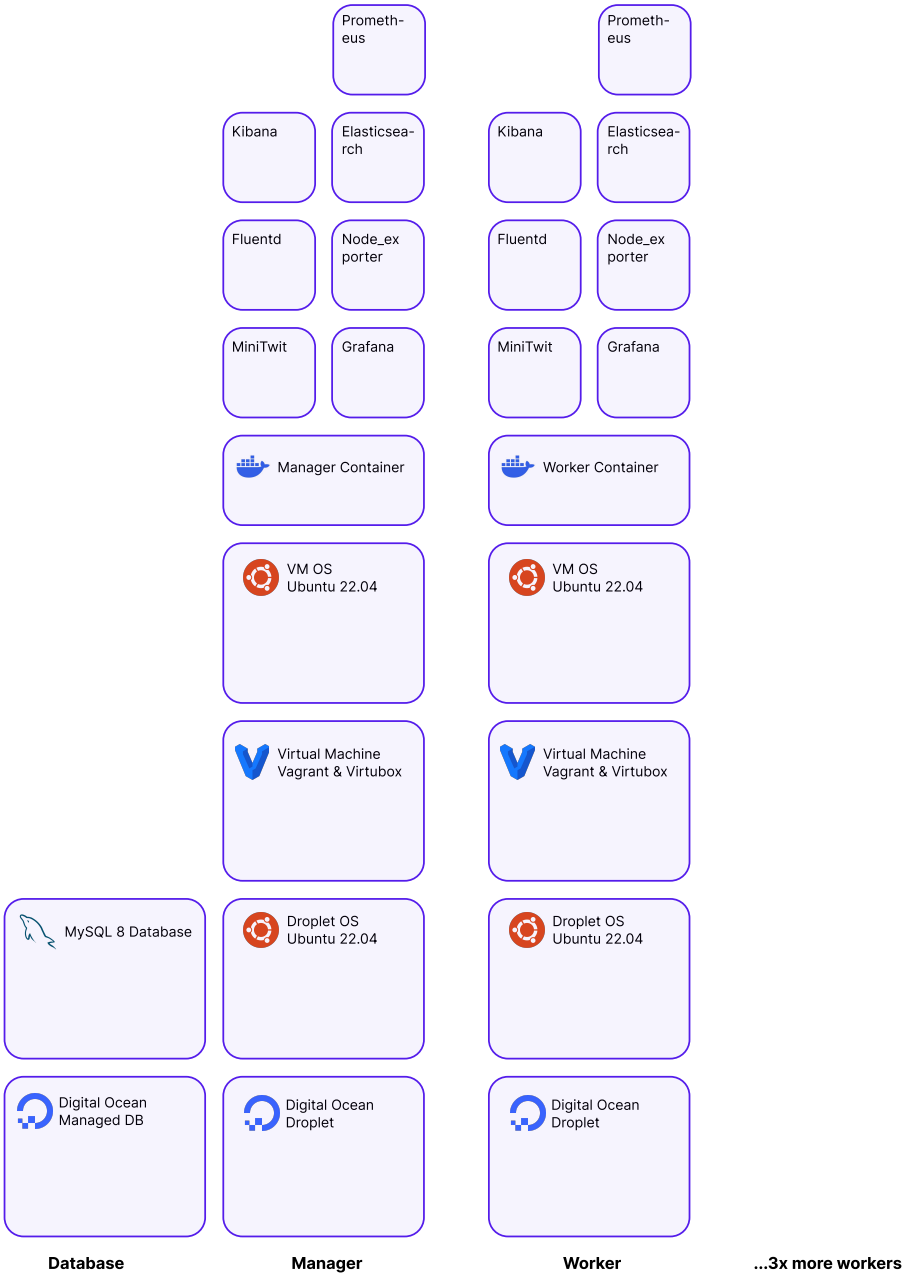

Figure 1: Hierarchical Architecture Mapping

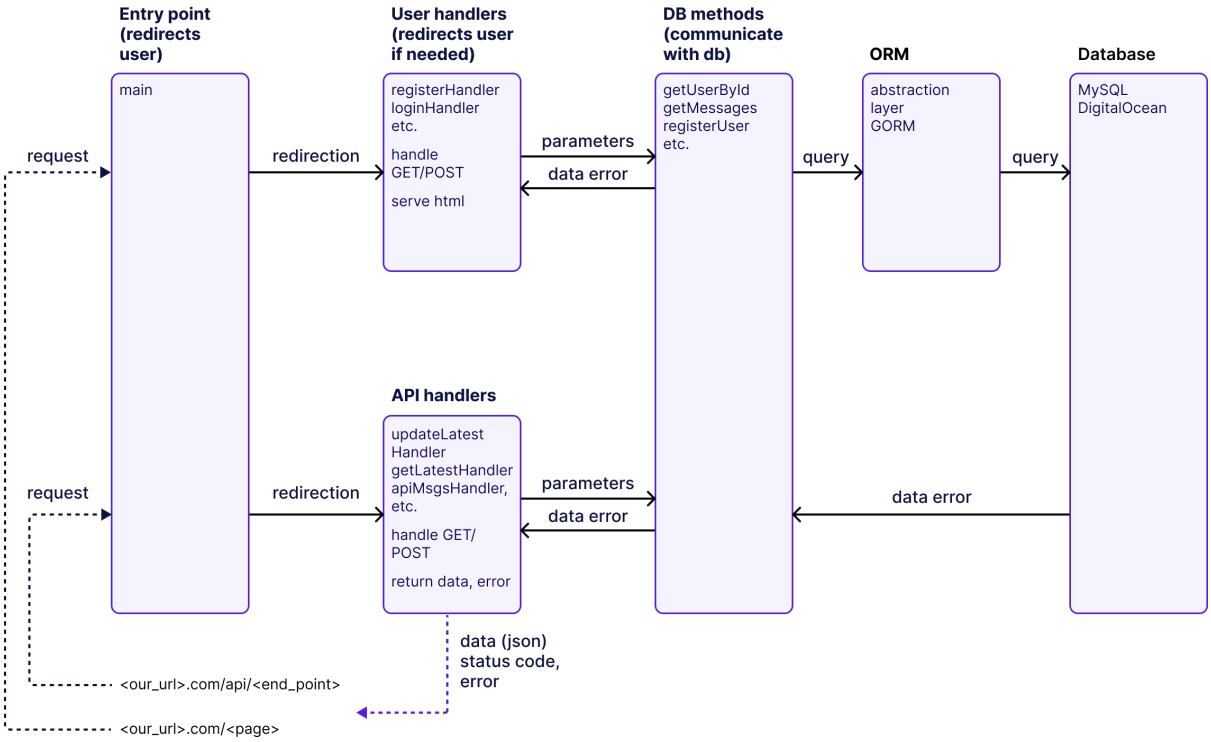

Figure 2: How our system handles a request.

them to test our application as such.

Finally, we used templated HTML files for the front-end of our application.

2.4.2 Containerization and Orchestration Tools

In order to ensure that our system had a high rate of availability (i.e. low down-time), we scaled horizontally our system. Even though our cloud provider Digital Ocean allows increasing droplet size (vertical scaling), we decided to also scale horizontally and thereby increase availability with Docker Swarm. We chose this tool for three reasons:

1. Ease of setup - since we already had experience with containerization Docker Swarm was easy to learn
2. Load Balancing - Docker Swarm has built-in load-balancing thereby increasing performance
3. Fault tolerance - by replicating nodes Docker Swarm increases fault tolerance and availability

2.4.3 Database Technologies

We were first provided with a lightweight SQLite database for ITU-MiniTwit. While this worked well for some time, we eventually realized that we needed a more persistent form of storage. We chose to purchase DigitalOcean's managed MySQL database solution. This was an easy choice for us as we were already using DigitalOcean to deploy our app online and using one of their database solutions allowed us to automatically refuse any incoming connection, except if it originated from our application, which increased the security of our system greatly. We also chose their MySQL solution as it was very easy to duplicate the schema used in the SQLite database that was provided to us and because we would be storing very structured data, eliminating the need for a document based database.

In addition to this, we implemented a database-abstraction-layer via GORM, an ORM library for GoLang. This was done for several reasons:

1. Since GORM supports multiple databases like SQLite, MySQL and PostgreSQL this added flexibility to our system since we were not tied to one single database for the entire lifecycle of our system, and we could easily migrate need be.

2. GORM provides auto-migration, making it much easier to import an SQL table and perform operations on it.

2.5 Static Analysis Tool Results

We gave two static analysis tools access to our GitHub repository: SonarCloud and CodeClimate. Their outputs can be found in figure 3 and 4. Both found some minor issues such as code smells, and maintainability issues, but did not raise any major problems such as bad code practices or security vulnerabilities. Furthermore, installing SonarCloud also gave us access to its bot which reviewed our code quality before we merged features into our main branch.

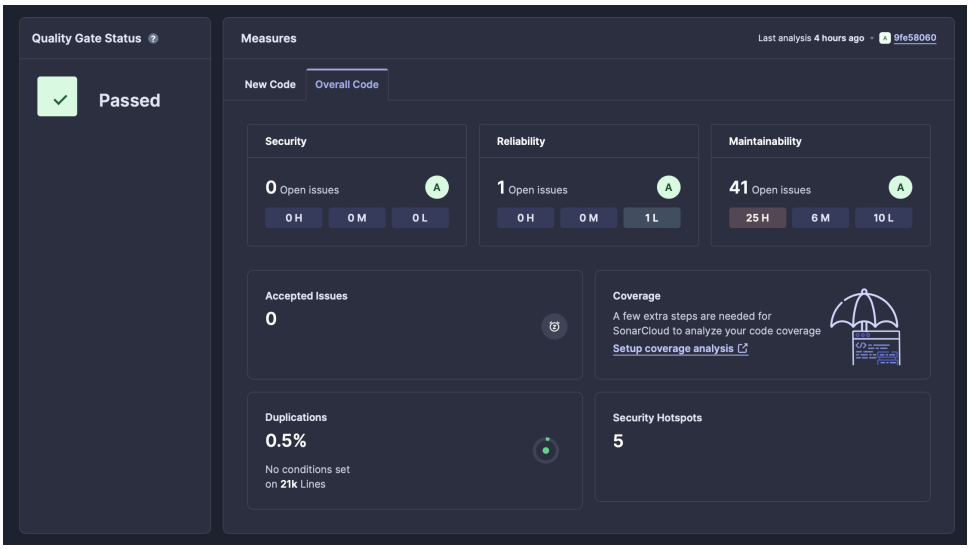

Figure 3: SonarCloud's dashboard after running an analysis on our repository.

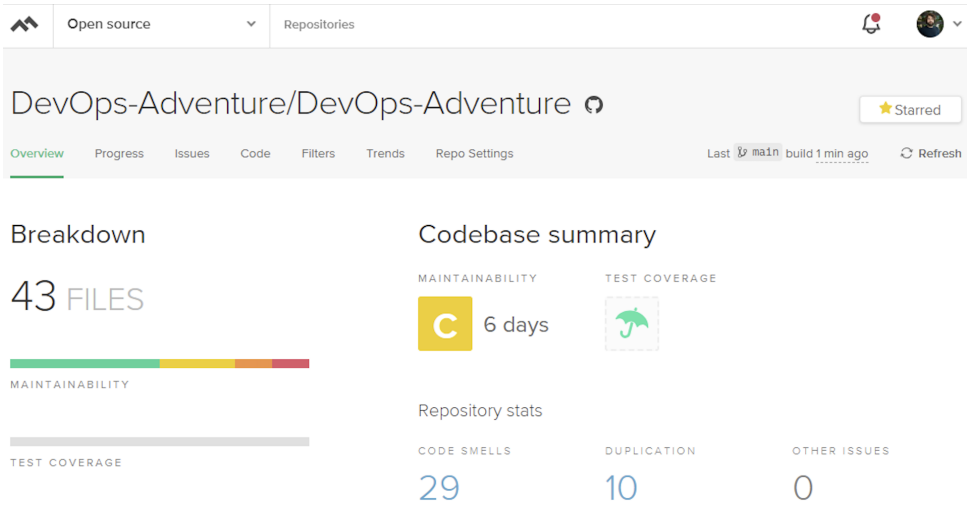

Figure 4: Code Climate's dashboard after running an analysis on our repository.

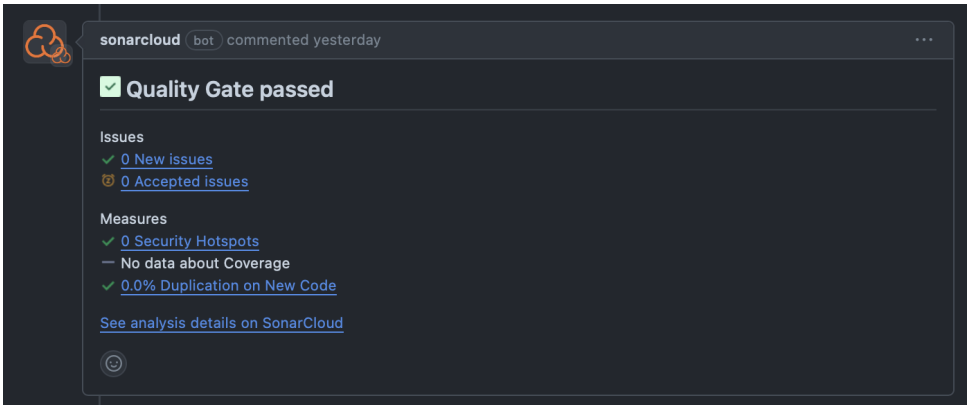

Figure 5: SonarCloud's bot adding comments on our pull requests.

3 Process' Perspective

3.1 Initial Resource Partitioning

Before deploying our system to DigitalOcean, we partitioned resources with the help of Vagrant.

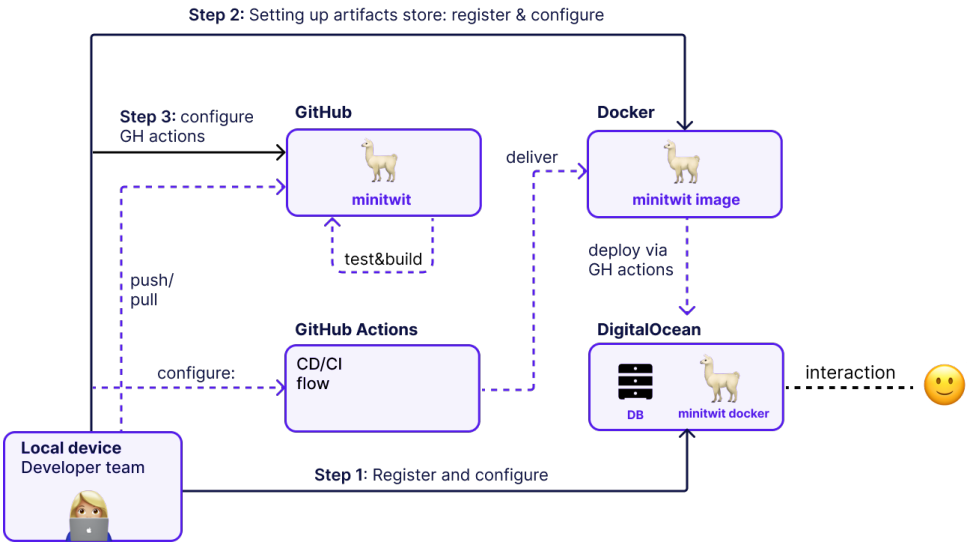

Figure 6: Resources Setup

3.1.1 Creation of the Virtual Machine and Server Setup on Digital Ocean

We designed a Vagrant file to programatically create the droplet, start the virtual machine (VM), connect to the droplet, install Docker container and test that docker was functioning correctly.

3.1.2 Setting Up Artifacts Store

We registered in DockerHub to store our ITU-Minitwit image and be able to easily push and pull changes from it. We also DigitalOcean to deploy our app to a managed server, rather than manage a server ourselves to minimise maintenance and risk failure, ease of use and access, and multiple possibilities for integration for our system like Docker Swarm and hosting our database on cloud. We chose DigitalOcean over other providers as the GitHub student developer pack gave us credits to use on the platform.

3.1.3 Configuration of Secrets on our GitHub Repository

We chose GitHub to store our codebase and to manage our CI/CD jobs. This required us to store secrets directly on the repository as environment variables so that they could be piped into our CI/CD actions. We also set up each member of our team with SSH key-pair access to the droplets to easily connect to the server.

3.2 Development, Integration, Deployment & Release

3.2.1 Development & Pre-commit

We utilized Git heavily during the course of this project. We also set up a few pre-commit hooks to facilitate development which would run linters such as golangci-lint, gofumpt and shellcheck. Finally, the pre-commit hooks also ran the tests that were provided to us.

3.2.2 Continous Integration

We created a continuous integration (CI) pipeline to enable us to test our code and new features frequently. We also created this pipeline to protect our main branch, where our ITU-MiniTwit was deployed, from having buggy or malicious code merged into it, as any merge into merge into main would automatically be deployed into our live system by our continuous deployment pipeline. Finally, we also set up our CI pipeline to run linters against code being added to the codebase require PR's to be reviewed and approved before it could be merged into our main branch and production environment. Figure 7 describes the steps in our CI pipeline and its main stages.

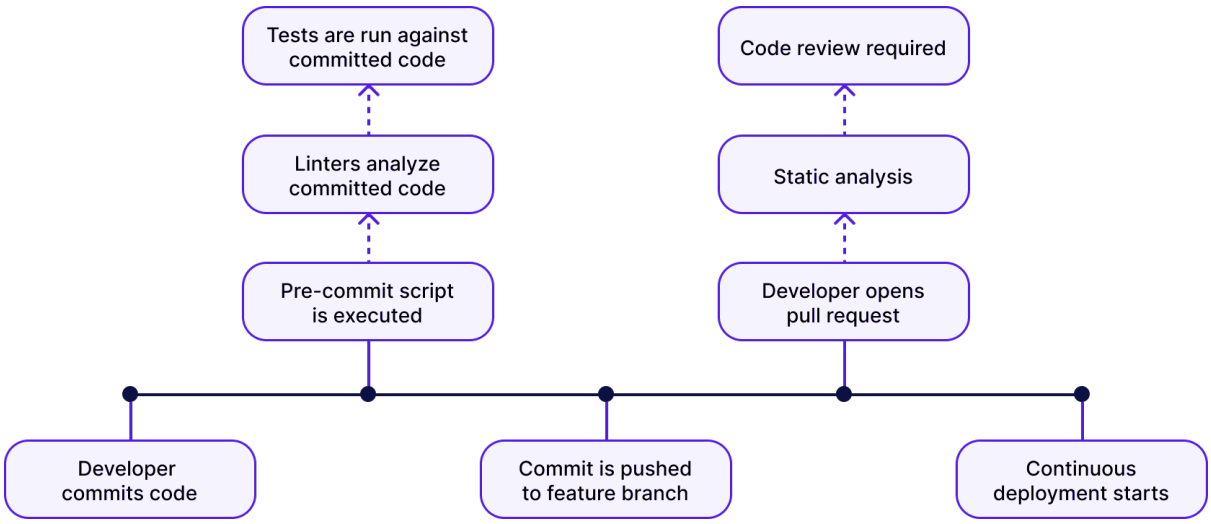

Figure 7: Our continuous integration pipeline.

3.3 Continuous Deployment

We also established a continuous deployment (CD) pipeline to automatically deploy the updated version of our ITU-MiniTwit. Once a new feature was incorporated into the main branch of our repo after the successful merge of a pull request, our CD pipeline would automatically trigger and deploy the new version of our application. Our CD pipeline was executed with a GitHub action and had many responsibilities:

1. Build the new version of ITU-MiniTwit and run tests against it.
2. Build a new Docker image of our app and push it to DockerHub.
3. Upload the deploy script from our repo to the DigitalOcean server

4. Stop current containers running on the servers and download the new images from DockerHub.
5. Start the containers with the new images.
6. Release a new version of the ITU-MiniTwit on GitHub.

If the CD pipeline successfully terminated, the new feature was deployed to our live system. It is important to note that the workers from the swarm were brought offline and back online using a rolling update scheme, to ensure that our system would always be available to access. This was easily implemented using Docker Swarm. The exact steps of the continuous deployment pipeline can be found in figure 8.

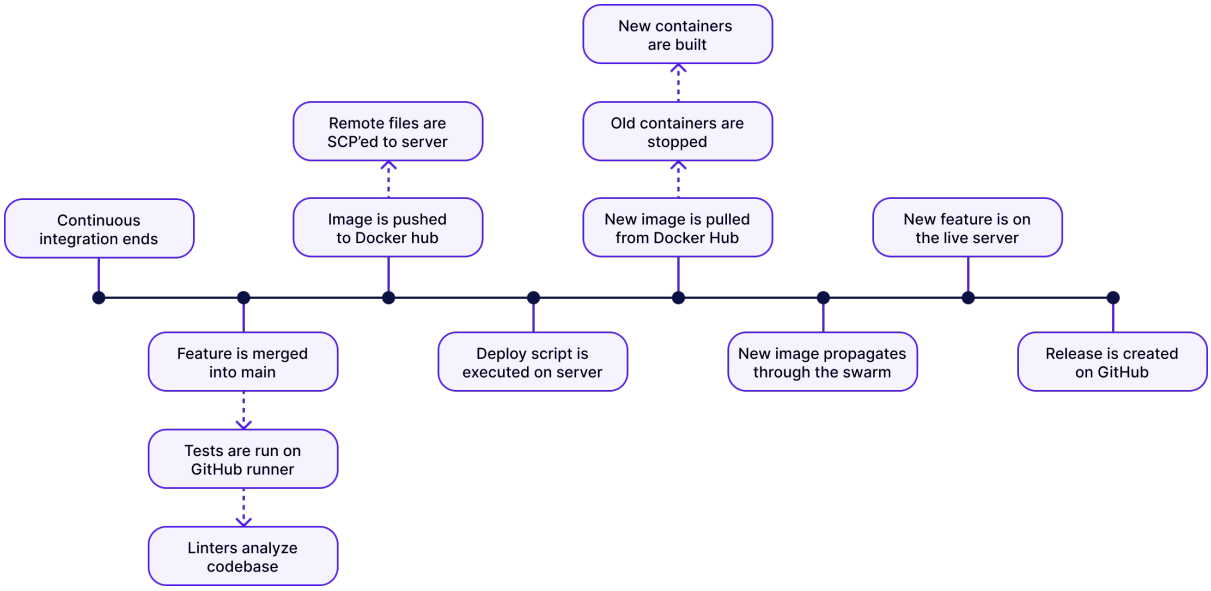

Figure 8: Our continuous deployment pipeline.

3.4 Monitoring

We used white box monitoring, which focuses on the processes within our application, providing insights for the internal state and performance of our application. We also used active monitoring, where we tracked interactions within our application, such as metrics for active logged users (note: without tracking their location).

3.4.1 White Box Monitoring

Figure 9 depicts the configurations we chose for Prometheus, and how and where it collects information. We chose to monitor metrics on two levels: application and system level.

1. CPU load in the application. A Grafana dashboard helped us diagnose a bug where the operating system of our virtual machine would starve our containers randomly. This is further described under section (note: todo).
2. Monitoring CPU, Memory, and Network metrics in the hosting system helped us determine whether issues originate locally within the application or at the server level. To be informed about overall health of the hosting environment was of a crucial importance as it could affect our application performance.

Figure 10 shows that after data was scraped from our application, it was stored in Prometheus container and then displayed using Grafana's visualization dashboards. We chose a Grafana and Prometheus stack for multiple reasons:

1. These free and open source software (FOSS) helped keep our costs for the overall project low.

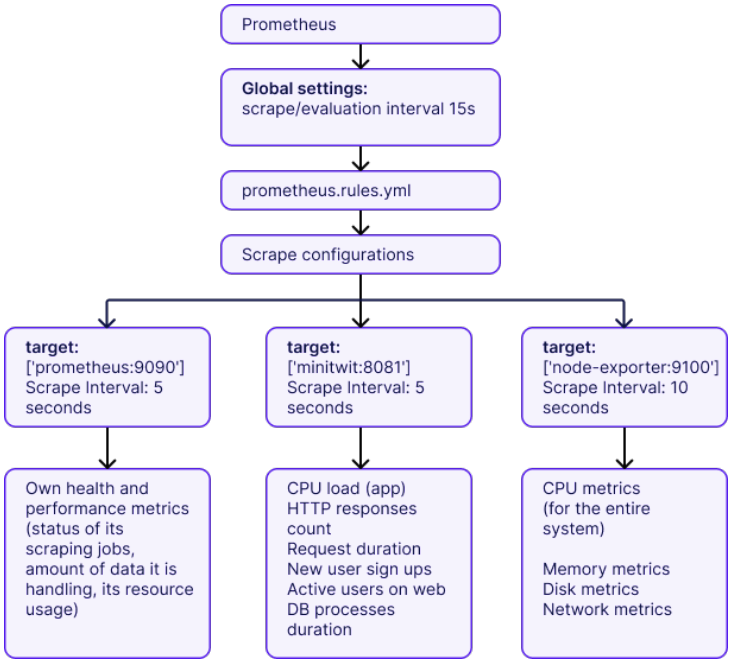

Figure 9: Prometheus configuration.

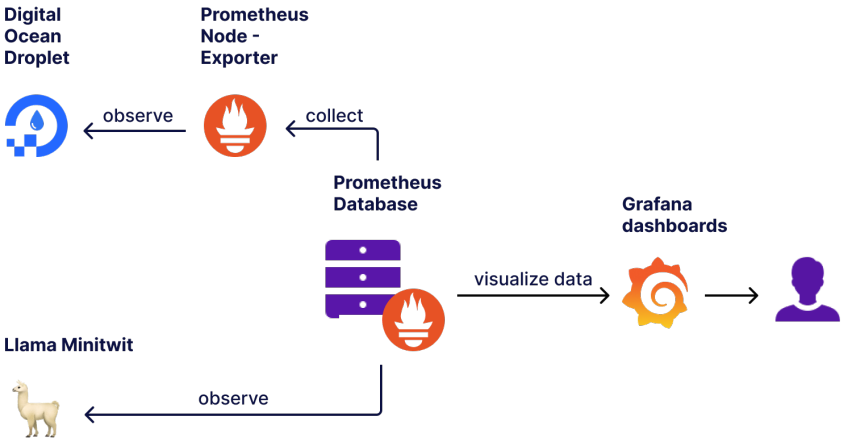

Figure 10: Visualization flow.

2. It is widely used as industry-standard way of monitoring, has good integration with cloud-origin environments and variety of plugins, provides detailed visualizations and is very user-friendly.

3.4.2 Active Monitoring

Despite our existing use of Grafana, we selected the EFK (Elasticsearch, Fluentd, Kibana) stack over Promtail + Loki + Grafana (PLG) for qualitative logging (active monitoring). The EFK stack has a mature ecosystem, well-suited search capabilities, and flexible data processing, which makes it a good choice for our qualitative logging. Figure 11 presents an overview of our EFK stack architecture.

Our minitwit application represents the source of the data we aim to analyze. We utilized Logrus for structured logs within our go application. Fluentd is the log aggregator in this stack. It collects logs from Minitwit, processes them, and forwards them to Elasticsearch. Elasticsearch serves as both a search and analytics engine. It indexes and stores log data forwarded by Fluentd, enabling flexible and fast search capabilities. Kibana is the user interface where we can view and interact with the log data stored in Elasticsearch. The logging stack was initially tested using local Docker containers with various logging scenarios. An example of the log output from the simulator is shown in Figure 12:

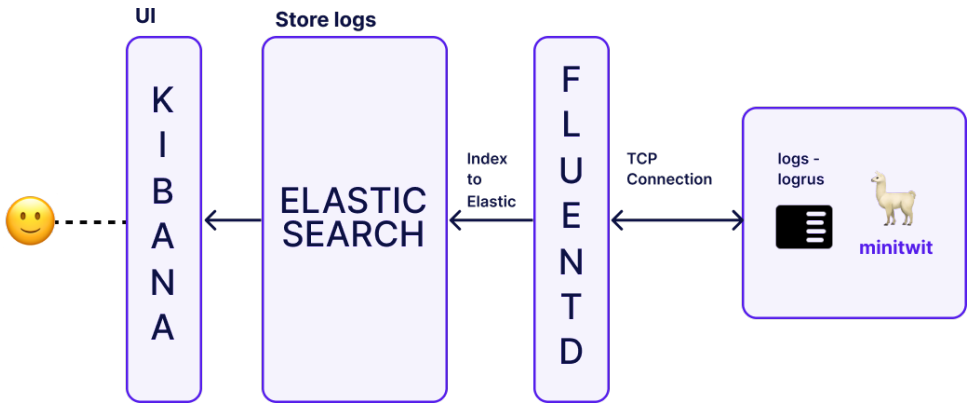

Figure 11: EFK Stack Architecture

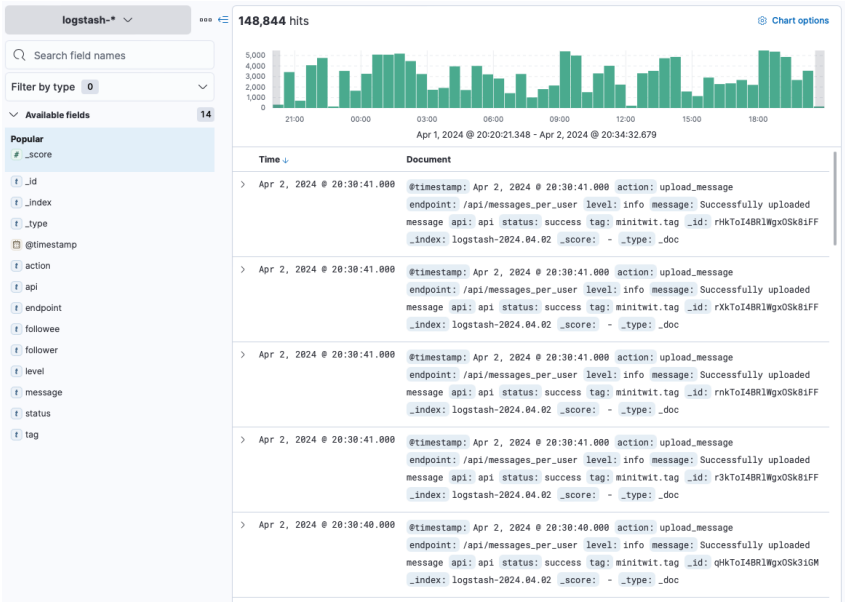

Figure 12: Logs Generated by the Simulator, Displayed in Kibana

3.5 Security Assessment

3.5.1 Risk Identification

The following assets have been identified and considered in the context of the security assessment:

1. **User Data**: it includes any information directly connected to users.
2. **Production VMs**: servers used to run the Llama MiniTwit application.
3. **Logging Data**: generated logs, especially the qualitative ones.
4. **Source code**: Llama MiniTwit application source exposition.
5. **Private Keys and Access Credentials**: keys and credentials used to access parts of the system.

3.5.2 Risk Scenarios

- ***Risk Scenario 1***: *Data Breach via SQL Injection*
  Description: An attacker exploits a SQL injection vulnerability in the Llama MiniTwit application to gain unauthorized access to the database.

Impact: Exposure of sensitive user data (emails, passwords, blog posts).

- ***Risk Scenario 2***: *Compromised Production VM*
  Description: The production VM is compromised due to weak SSH credentials, allowing an attacker to gain control over the server.
  Impact: Unauthorized access to application and data.

- ***Risk Scenario 3***: *Distributed Denial of Service (DDoS) Attack*
  Description: The application is targeted by a DDoS attack, overwhelming the servers and causing service disruption.
  Impact: Unavailability of the application.

- ***Risk Scenario 4***: *White-Box Exploits*
  Description: The source code of Llama MiniTwit is accidentally made public on a version control platform like GitHub.
  Impact: Exposure of application vulnerabilities and sensitive information (e.g., hardcoded credentials)

- ***Risk Scenario 5***: *Third-Party Software Breach*
  Description: A third-party software component used in the application is compromised, leading to security vulnerabilities or data breaches.
  Impact: Software vulnerabilities and data leaks

- ***Risk Scenario 6***: *Exposure of Sensitive Information in Logs*
  Description: Qualitative logs containing sensitive user information and operational details are exposed due to misconfigured logging settings.
  Impact: Leakage of user behaviour patterns eventually personal information.

- ***Risk Scenario 7***: *Brute Force Attack*
  Description: An attacker that uses trial and error to gain access to passwords, login credentials, and encryption keys.
  Impact: Unauthorized access to user accounts, data leaks, data manipulation.

3.5.3 Risk Analysis and Mitigation

Following we have determined the severity of each threat and placed them in a risk matrix to prioritize appropriate mitigation strategies. Furthermore, the outcomes are discussed and potential solutions are presented. The severity categories are defined as follows:

	Insignificant	**Marginal**	**Critical**
Likely			
Possible			1, 5
Unlikely	7	4, 6	2, 3

	Low risk
	Medium risk
	High risk

High Risk

- **1** *Data Breach via SQL Injection:* Our Llama MiniTwit has an abstraction layer in place, so it is highly unlikely to be attacked by a SQL injection. To further mitigate the risk of data loss, regular backups could be implemented.
- **5** *Third-Party Software Breach:* The impact of a third-party software breach can vary. Fortunately, this risk can also be mitigated e.g. checking the security posture, compliance with industry standards, and historical performance.

Medium Risk

- **2** *Compromised Production VM:* DigitalOcean offers the capability to add a firewall that serves to safeguard access to our virtual machines. However, no additional firewall rules have been implemented to protect the VMs themselves yet.
- **3** *Distributed Denial of Service (DDoS) Attack:* Our application is not that popular, so it is unlikely to be under a DDoS attack. However, it is still possible since our application is public. If such a thing happens it will probably experience some downtime.

Low Risk

- **4** *White-Box Exploits:* Our Git repository is public, meaning anyone can view our code and potentially create exploits based on it. However, our code review processes provide a layer of protection in such scenarios. Additionally, we have been diligent in ensuring that no credentials are stored in the repository.
- **6** *Exposure of Sensitive Information in Logs:* The implementation of a proxy server plus firewall can render the logs inaccessible from external sources. Even if the logs are accessed by a malicious actor, no behavioural patterns or private user information is being displayed.
- **7** *Brute Force Attack:* This scenario is unlikely to happen and would only affect one or a few users. Two-factor authentication could be implemented as a potential solution.

3.5.4 Pen Testing

We conducted a penetration test using Metasploit and the WMAP plugin. Our main objective was to target the production URL (`http://134.209.226.165:8081/public`) to identify potential vulnerabilities. The WMAP scan reported no vulnerabilities, as shown in the terminal output below:

```
[*] Done.
msf6 > wmap_vulns -l
[*] + [134.209.226.165] (134.209.226.165): directory /log/
[*]  directory Directory found.
[*]  GET Res code: 404
[*] + [134.209.226.165] (134.209.226.165): directory /login/
[*]  directory Directory found.
[*]  GET Res code: 301
[*] + [134.209.226.165] (134.209.226.165): directory /logout/
[*]  directory Directory found.
[*]  GET Res code: 301
[*] + [134.209.226.165] (134.209.226.165): directory /public/
[*]  directory Directory found.
[*]  GET Res code: 301
[*] + [134.209.226.165] (134.209.226.165): directory /register/
[*]  directory Directory found.
[*]  GET Res code: 301
[*] + [134.209.226.165] (134.209.226.165): directory /static/
[*]  directory Directory found.
[*]  GET Res code: 404
[*] + [134.209.226.165] (134.209.226.165): file /login
```

```
[*]  file File found.
[*]  GET Res code: 200
[*] + [134.209.226.165] (134.209.226.165): file /logout
[*]  file File found.
[*]  GET Res code: 302
[*] + [134.209.226.165] (134.209.226.165): file /public
[*]  file File found.
[*]  GET Res code: 200
[*] + [134.209.226.165] (134.209.226.165): file /register
[*]  file File found.
[*]  GET Res code: 200
msf6 > vulns

Vulnerabilities
===============

Timestamp  Host  Name  References
---------  ----  ----  ----------

msf6 >
```

3.6 Scaling

Our strategy for scaling mainly relies on the services provided to us by DigitalOcean and our system architecture. Should we need it, we could scale vertically by purchasing droplets with more resources on DigitalOcean. We could also purchase more droplets and create additional swarms around the world. Finally,

3.7 Our use of AI assistants

AI-tools such as ChatGPT proved to be very helpful for:

- Debugging and handling errors.
- How different functions could be implemented in different languages.

However, we also found that AI assistants were not helpful for:

- Direct translations of code from one language to another.
- Providing correct answers for higher complex issues such as adding GORM as database abstraction layer or indexing or database.

In general, throughout our project we did not rely heavily on AI assistant as we were able to solve most issues reading documentation and using each other.

4 Lessons Learned Perspective

In this section, we describe some of the technical difficulties we ran into during the semester, the lessons we learned from them and reflect on how we these difficulties could have been avoided or how we could have recovered more gracefully.

4.0.1 Bad Database Management

Initially, we had successfully running flow that was able to update the docker image, however we were failing to do so because the script which was responsible for rebooting the docker was not running smoothly. as a result we have had a new image, but we were still updating our database with the old one for about two weeks. After the script was fixed we were able to reboot our DockerHub and update the Minitwit image with the latest version.

4.0.2 Resource Starvation

During the 4th session of the course, we were tasked with deploying our ITU-MiniTwit system to the cloud so that the simulator could eventually interact with it through our API. We partitioned a droplet on DigitalOcean and deployed our app through a Docker container. We had some additional services running on other containers but on the same droplet. Once the simulator started, our Docker container would seemingly randomly be shut down every few hours. After a few weeks, we realized that there was a direct correlation between our container being shut off and a spike in traffic on our ITU-MiniTwit. We realized that the sum of resources we had allocated to each container running on the virtual machine installed on our droplet was higher than the resources we had purchased from Digital Ocean. More specifically, our containers were allowed to consume up to 6GB of RAM, but our droplet only provided us with 4 GB. Thus, when the traffic spiked, the container consumed more RAM than it physically had access to and the OS would kill it.

While it's easy for us to look back and recognize that this was a simple oversight, we had seen many tutorials that recommended capping the resources each Docker container could consume and had simply glanced over the information.

4.0.3 Slow ITU-MiniTwit

After the simulator had been running on our live system for a few weeks, we noticed a significant increase in the dataset size, which resulted in much longer query times. To address this issue, we implemented indices in the database using SQL queries. To determine which indices to create, we focused on the processes that were performing the slowest: accessing the public timeline, retrieving messages, and managing followers. For these, we implemented B-tree indices and we managed to optimize the performance of our database queries and ensuring efficient data retrieval.
For user table (unique):

```
CREATE INDEX idx_username ON user(username(255));
CREATE INDEX idx_email ON user(email(255));
```

For message table (non-unique):

```
CREATE INDEX idx_author_id ON message(author_id(255));
CREATE INDEX idx_pub_date ON message(pub_date(255));
```

For follower table:

```
CREATE INDEX idx_who_id ON follower(who_id(255));
CREATE INDEX idx_whom_id ON follower(whom_id(255));
```

4.1 DevOps Style

4.1.1 Good Git Practices

We established a routine where we worked separately on feature branches, testing our code locally before creating a pull-request to the main branch. To merge into main another developer would then have to review. This greatly reduced the risk of bugs finding their way onto our deployed environment.

4.1.2 The three DevOps ways

1/ The Principles of Flow

- Visibility: We were sharing our work by defining tasks in a Notion board to help us to keep track of what we needed to do.
- Limit Work in Progress: We sat a limit on the tasks that could be in each column in the Notion board. In case we were waiting on a task to be completed, we asked the team member what was causing the delay and help him/her instead. We we meeting regularly in person for our lectures and we make sure to have at least one meeting in person per week. When we are not co-located we are distributing our tasks accordingly.

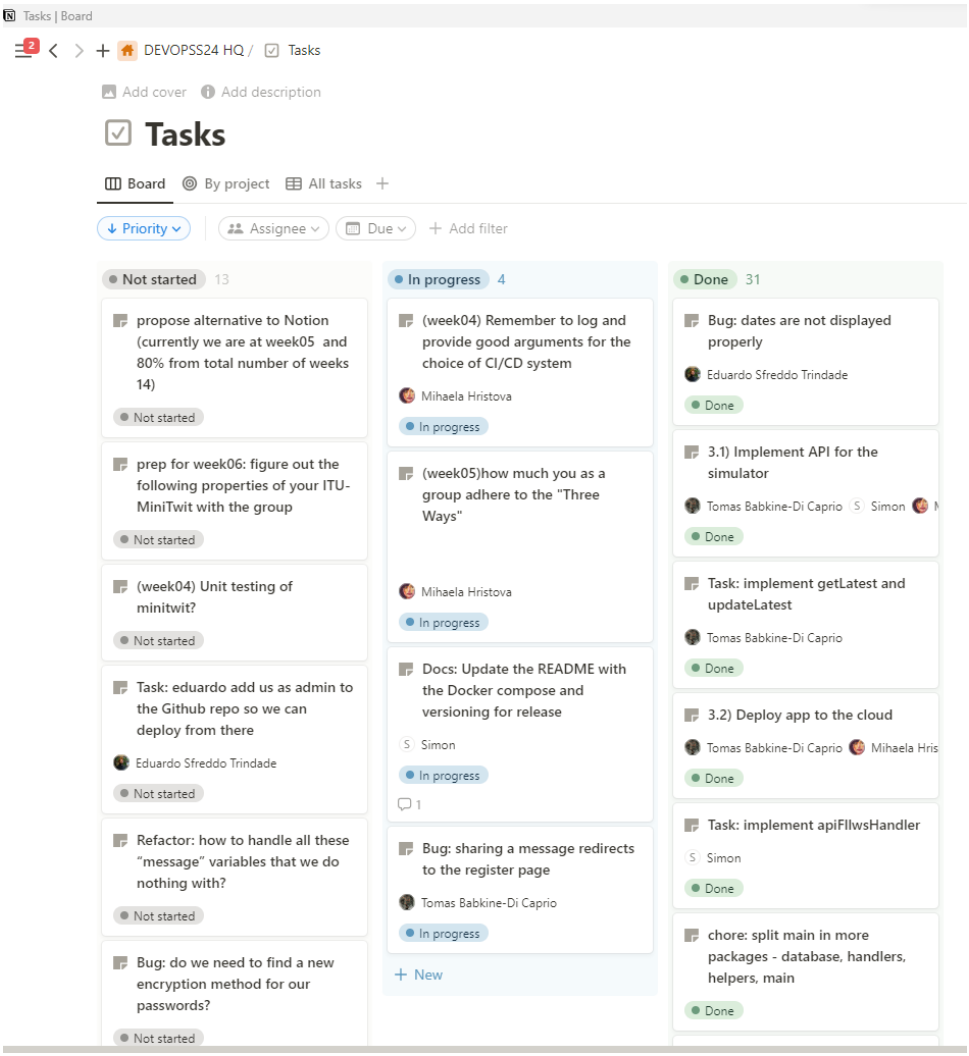

Figure 13: Notion dashboard where we keep track of our tasks.

- Reduce batch sizes: In our case we looked at each different feature as a single unit operation being refactored, tested and deployed. Instead of implementing all features at once (post messages, follow user, register etc) we develop and test one at a time. This shortens the lead time significantly and most importantly help us to detect and act on bugs much faster.

2/The Principles of Feedback

- We were revising our code

3/The Principles of Continual Learning and Experimentation we acknoledge that

5 Dependencies

6 References

1. Official website of Go https://go.dev/ -for installation and tutorials
2. LinkedIn learning "Learning Go" by David Gassner https://www.linkedin.com/learning/learning-go-8399317/explore-go-s-variable-types?u=55937129
3. Reference for indexing the database https://github.com/dbeaver/dbeaver/wiki/Creating-Indexes