IT UNIVERSITY OF COPENHAGEN

DevOps:
Software Evolution and Software Maintenance
Final Exam

GitHub Repository

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

BSDSESM1KU

1 System's Perspective

1.1 Design and Architecture

Our version of MiniTwit uses Docker Swarm with 6 virtual machines (vms) and a separate vm for the database. The API and back-end are both built in Golang. The web framework is built using Gin. PostgreSQL is used for our database. Prometheus is used in conjunction with Grafana for monitoring, and Loki is used for logging.

1.2 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 front-end, 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 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 our Docker containers 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 the Loki logs and the Prometheus metrics.

Infrastructure:

- **Docker** - Builds our application
- **Docker swarm** - management and scaling of Docker containers
- **Linux** (Ubuntu?)
- **Digital Ocean** - Our choice of cloud service provider

CI/CD:

- **Vagrant. Working on converting to Terraform** - 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
- **ShellCheck and govulncheck** - analyses code/commands checks for faults

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** - Chatbot, helps us getting quick answers

1.3 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 all the other containers.

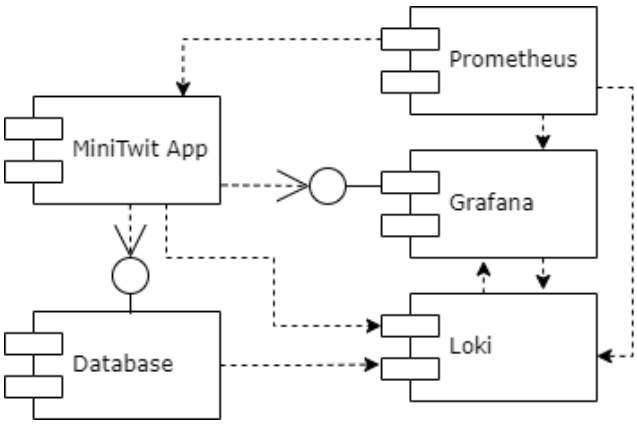

Figure 1: Subsystem interaction

1.4 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 2.5). Static code analysis, shown in Fig. 2 does not, however, not detect many large reliability issues.

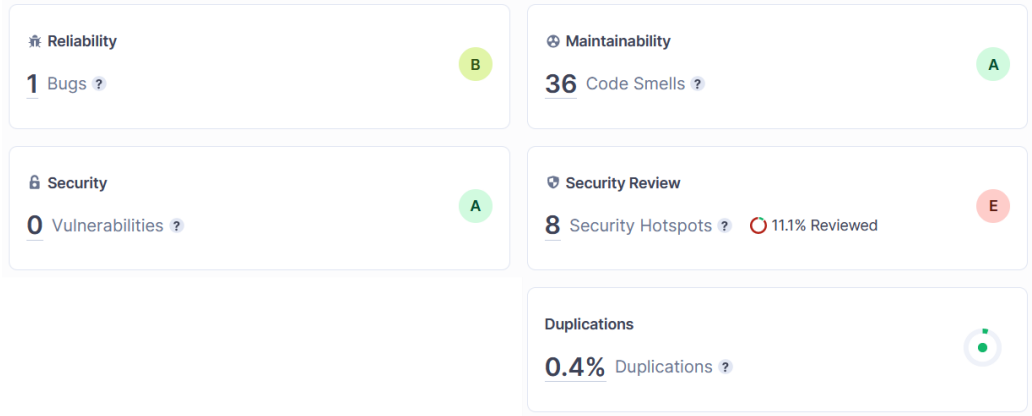

Figure 2: SonarCloud scan results of the development branch.

2 indicates that the code is *maintainable*. The low complexity of the system also reduces the threshold of when the system is unmaintainable.

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 digital ocean as IaaS provider.

1.5 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 1).

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.

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 1: Code dependencies and their licenses

Permissive licenses have minimal requirements for usage and redistribution and typically only ask for the inclusion of the original copyright notice and a copy of the license text when distributing the software. They are designed to be accommodating, allowing developers to use the code in various projects, including proprietary ones.

2 Process' Perspective

2.1 Team Interaction and Organization

2.1.1 Meeting times

The way we worked is that we treated each week as a "sprint" where we would mainly work asynchronously. Once per week, 3 hours before the lecture would 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 where workload would be especially high.
The mediums used for interaction was:

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

2.1.2 Work delegation

Tasks would be created from users, taken from weekly course tasks, or created internally based on status of implementing different tools or functionality.
As all of us were new to the concepts introduced throughout the course, we weren't able to pair up someone experienced with someone inexperienced, which would have been the ideal scenario for us. Therefore, we would typically try to assign 2 people, but as technical debt began piling up, we sometimes had to assign 1 person to a tasks, which was not ideal.

2.2 Description of Stages and Tools Included In the CI/CD Chains

Git version control, hosted on github. Repository setup and branching Kanban Docker-compose file

Digital Ocean as IaaS vendor.

2.3 Organization of Our Repository

The way our repository is set up is that we have one repository with all of our services. We had some issues with folder structure in the beginning, but managed to figure a structure out that worked nicely for us. 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 identifying where a specific service would be.

2.4 Applied Branching Strategy

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

The **development** branch is for testing our code. Here is where every newly developed feature is merged into as per default when creating a pull requests. This is also where we would discover if any new features were to 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 to 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 a track of what each branch is for.

This was all agreed on internally, but later decided needed to be set in stone and so the "CONTRIBUTING.md" file was pulled in to our repository.

2.4.1 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 to each person. 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 as serving 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.

2.5 Monitoring

Metrics we monitor:

-

Early into monitoring we were at the manual stage [1]. As shown in Fig. 3 we experienced downtime twice. The first occurrence of downtime was longer as a consequence of monitoring manually, as this was during Easter holiday. We learned that the system was down by manually checking, and subsequently fixed the issue each time.

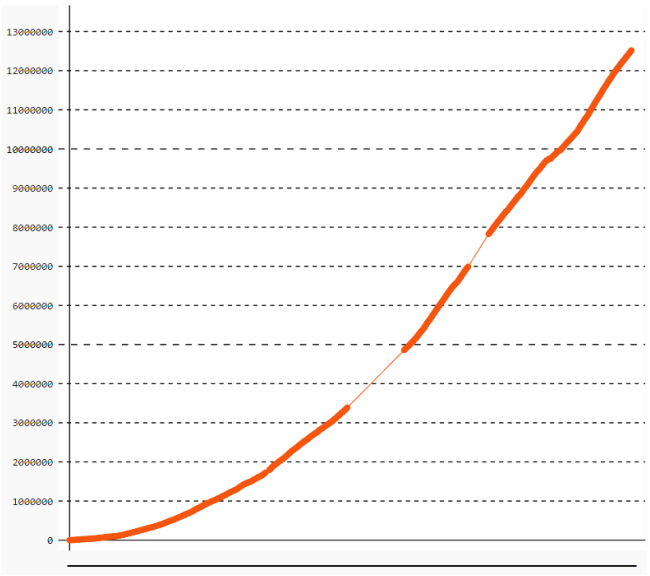

Figure 3: A plot showing the latest IDs sent and reported by the simulator. The two thin spots in the line indicate the two times our system was down. The x-axis is the duration of the simulator, and the y-axis is the ID.

The first metrics we set up for monitoring were the number of 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 then 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. As per the way the subsystems interact as described in section 1.3, since Prometheus

is added as a data source to Grafana, 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" that will track if the system is down, and added the corresponding panel 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.

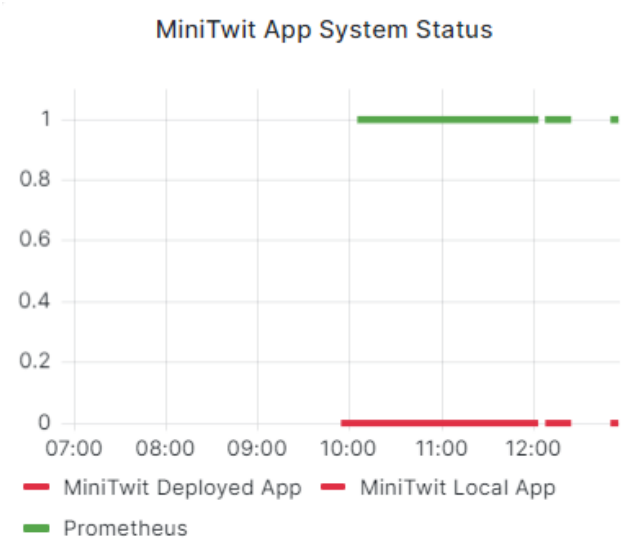

Figure 4: A screenshot showing the panel on our dashboard that shows the system status of our application.

There are addition metrics we would consider monitoring. We initially monitored the CPU gauge, but did not find the information relevant for our needs at the time. There were a couple of weeks when requests to each endpoint in our system could take many seconds. Therefore we would like to include having a panel displaying a load gauge, indicating the time needed for endpoints to load. For the business perspective we would like to monitor activity level of the app based on users, for example the inflow of users, how often a user tweets, and how users interact with each other.

2.6 Logging

We set up Grafana Loki to aggregate logs. We had initially installed the Loki plugin within the droplet on DigitalOcean that runs our MiniTwit. Then, in our *docker-compose.yml* we specified the logging driver to be Loki. The first time we ran this we logged into Grafana, connected the Loki data source, and configured a dashboard to display the logs.

Since our system is built with containers, when releasing a new version of the app and recreating new containers the logs would be lost. By aggregating the logs with Loki we were able to keep logs for analysis even if the containers are deleted or recreated.

Then 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.

After switching to Docker Swarm, the logging seems to be broken, therefore this version of our

MiniTwit app does not contain logging as of yet.

Further logging...

2.7 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 Digital Ocean 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 2: Identified risks and set up scenarios for each risk

After identifying the risks, we analysed how big of a deal it would be, if they were to happen. We visualised this with a Risk Matrix.

Risk Matrix

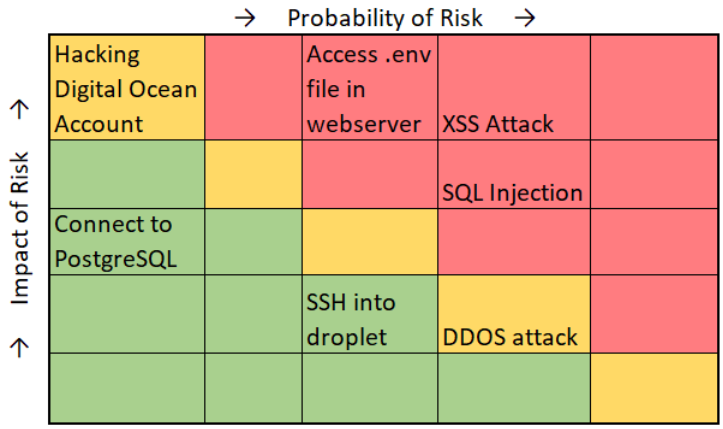

Figure 5: Risk Assessment

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.

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.

2.8 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 / main branch. Auto-scaling is not a feature of the system. Our scaling/load balancing implementation is based off of this repository.

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 has the minitwit service (see **??**) replicated 9 times, the load balancer replicated 2 times, with the rest of the services only having 1 replica.

There are 3 manager nodes (one of which is a leader), and 3 worker nodes. The manager nodes orchestrate the cluster and delegate work to worker nodes.

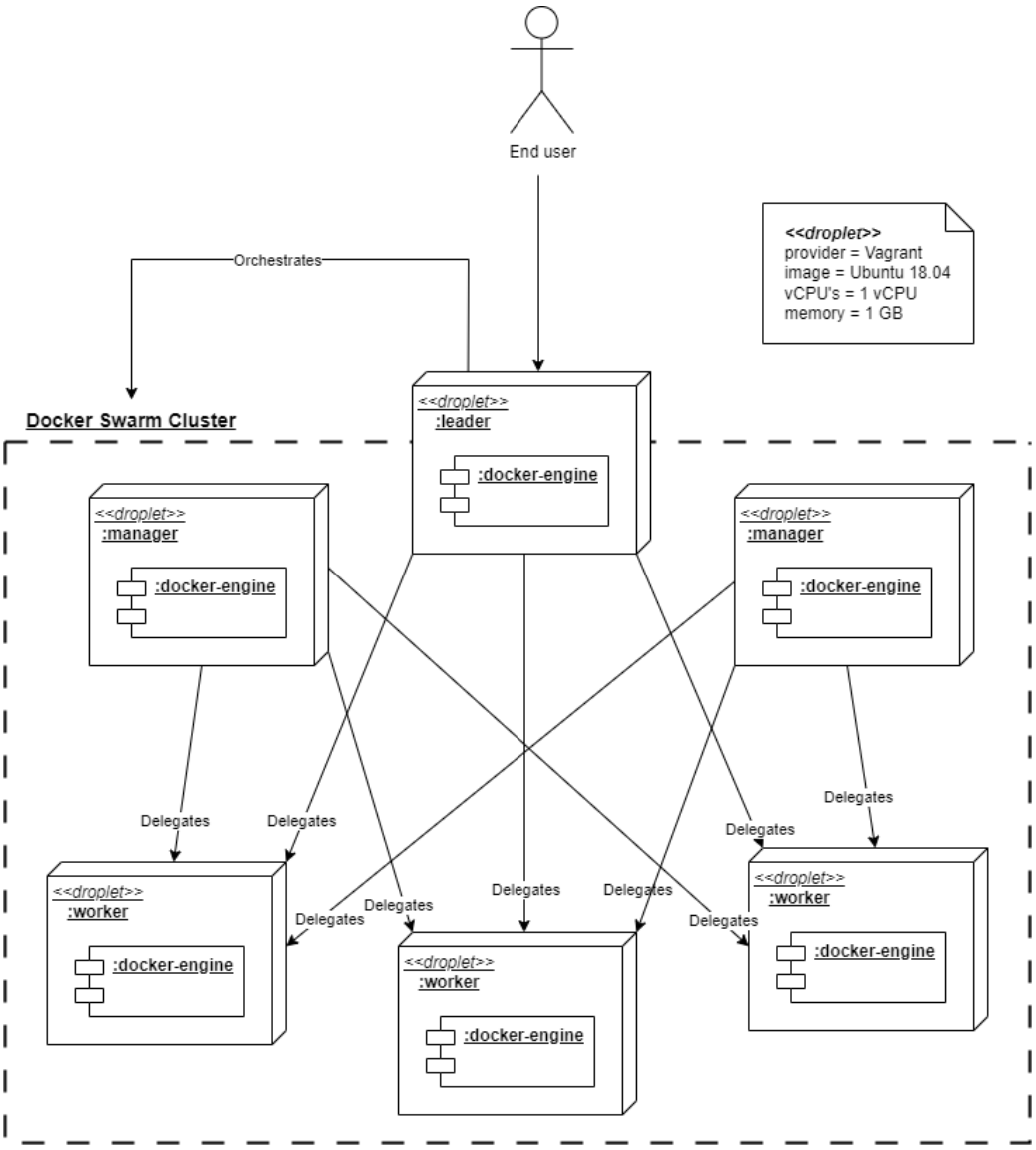

Figure 6: Interactions between nodes in our swarm cluster.

2.9 Using AI Assistants

Assisted in writing the SLA.

Attempted to use for how to implement "cookies" using the Gin framework, due to lack of sufficient examples in documentation. We inquired ChatGPT for examples of code, but did not end up using it. So we do not know if this is a good use case.

3 Lessons Learned Perspective

3.1 Evolution and Refactoring

Over the course we created a fairly high technical debt. Due to deadlines, we tended to get each job done "good enough", with the idea to go back and make it perfect later. But later never happened, due to continually more tasks, issues, or feature development. It would have been better to spend the extra time and complete tasks well the first time.

3.2 Operation

3.3 Maintenance

With more time we would like to implement additional monitoring so that certain qualities of our system are automatically checked. For example, query time to all endpoints to ensure our system is not lagging.

3.4 Our "DevOps" Style

For now just some raw notes on what to include:

We are missing: Pen-test of other teams system? Security report. SLA review of other team. Certain kinds of tests Static code analysis Docker swarm Adherence to the 'three ways' 'Rolling updates strategy'?

References

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