DevOps Project 2024
Group n - Exam Report

Name	Email
Daniel Sølvsten Millard	dmil@itu.dk
Frederik Lund Rosenlund	frlr@itu.dk
Jacob Pærregaard	jacp@itu.dk
Natalie Clara Pedersen	natp@itu.dk
Rasmus Lundahl Nielsen	raln@itu.dk

Submission: 23/05 - 2024

Courses: DevOps 2024
Course code: BSDSESM1KU

Contents

1 Introduction

The following report is written by Group N, MacOnTop, as a part of the course *DevOps, Software Evolution and Software Maintenance.* The report will be focused on the final state of our program as well as the lessons learned through the process of maintaining it during the semester.

2 System Perspective

2.1 Design and Architecture

Program Architecture

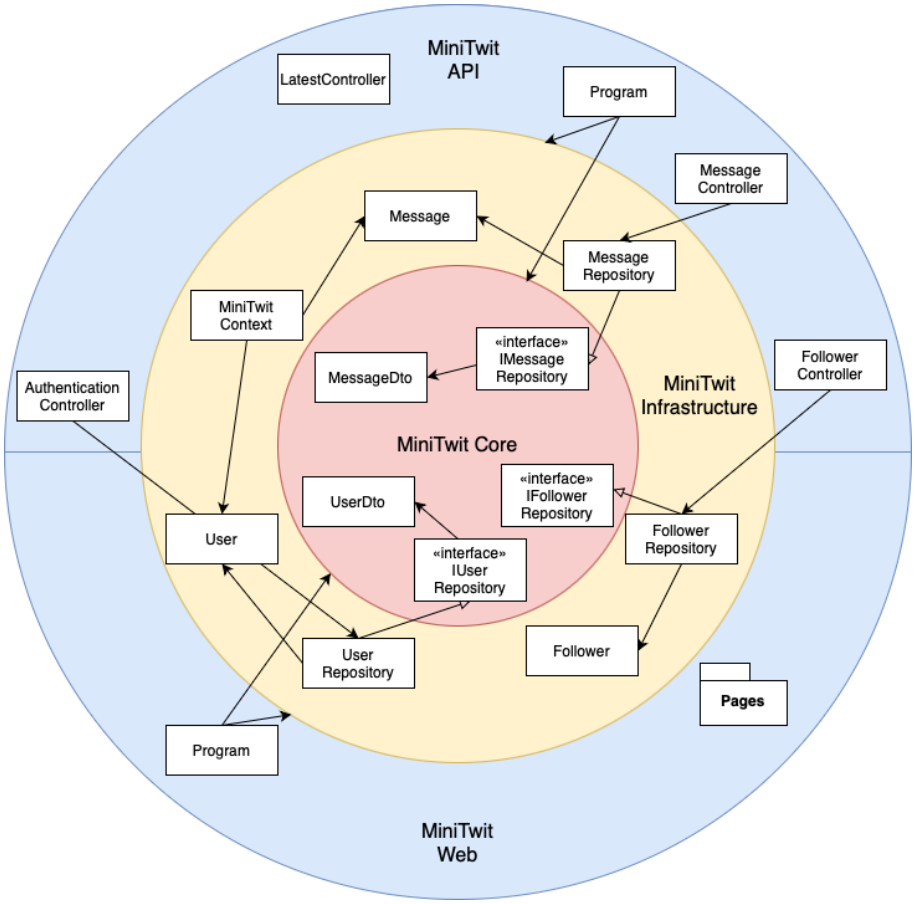

Figure 1: Figure showing the architecture of our program.

System Infrastructure

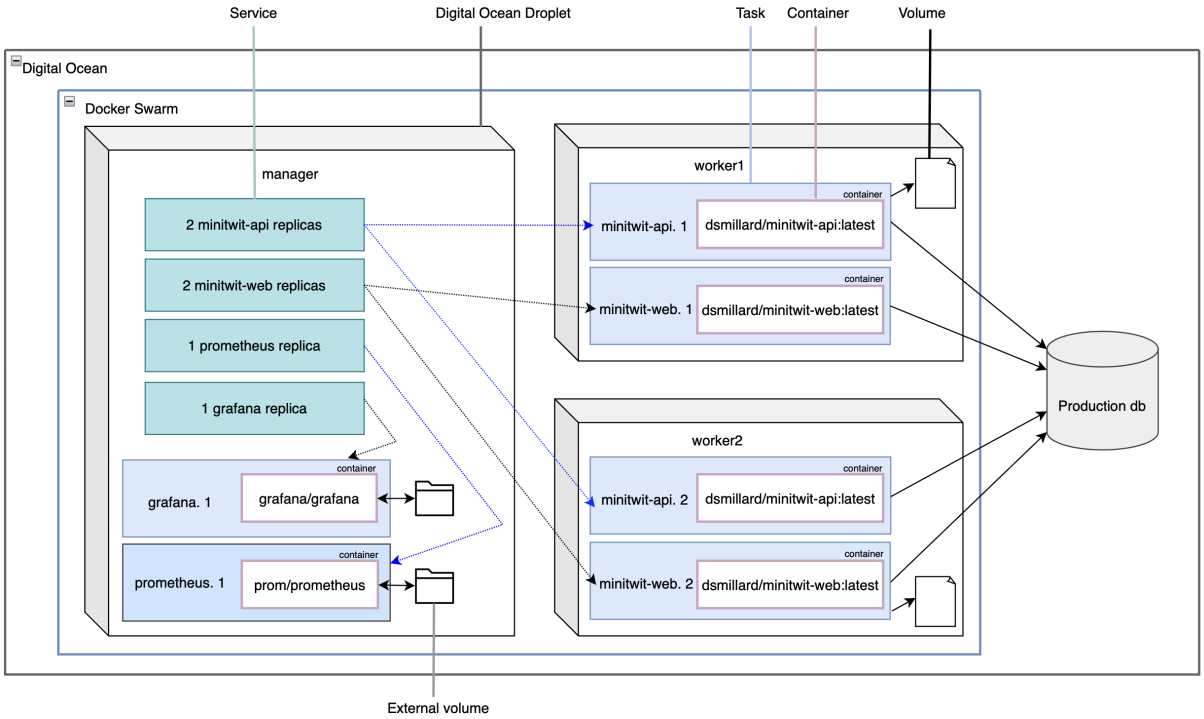

Figure 2: Overview of our system infrastructure.

The system has 3 droplets; a manager and 2 worker nodes that form the application. The manager hosts the prometheus- and grafana-application, and the workers host the web- and api-application. The database is a managed database hosted on DigitalOcean.

Interaction of Subsystems

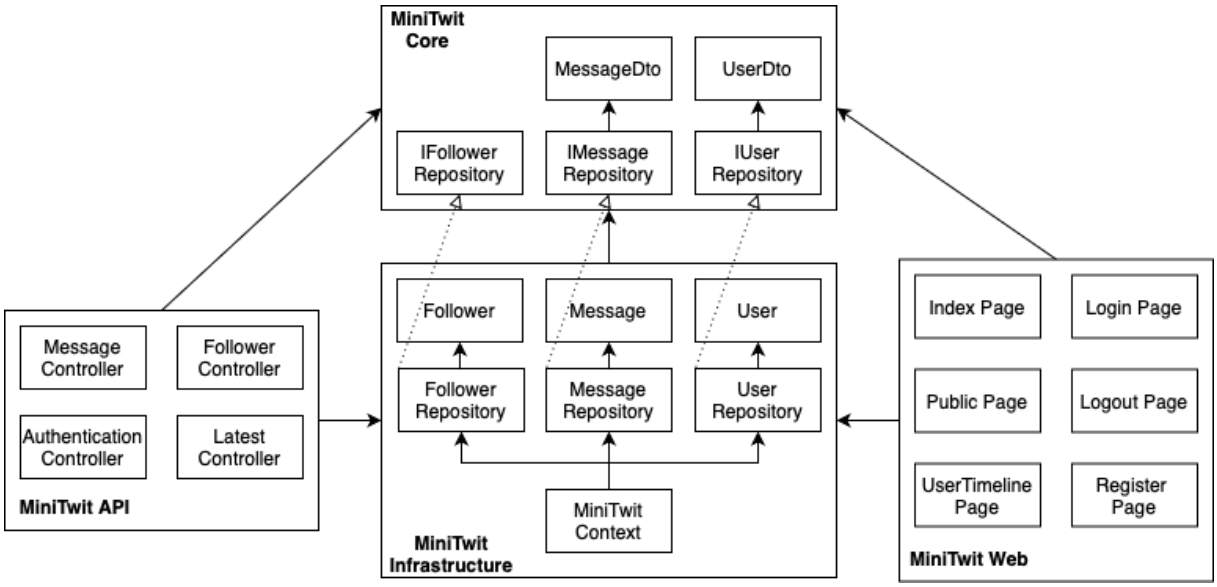

Figure 3: Overview of how our subsystems interact with eachother.

Core is the smallest abstraction of our application. It only holds the data-transfer-objects and the interfaces for our repositories. Then comes the infrastructure that implements the interfaces from the Core and is the layer that is responsible for how we query the database. The API- and Web-application is the front of the application. They use the infrastructure to query the database and then either give something back to the user (API) or show the user something (Web).

Example of Interaction - API

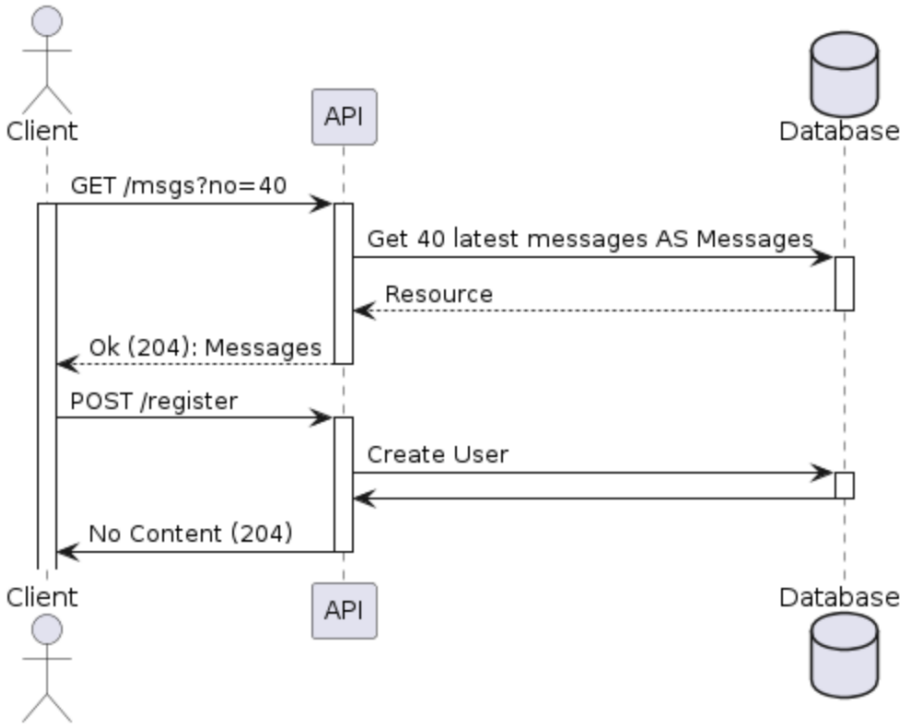

Figure 4: Example showing 2 interactions to the API; getting messages and registering a user.

2.2 Dependencies & Tools[1]

Operating System

- Ubuntu 22.04 (LTS) x64

Development Environment

- .NET SDK 8.0

Containerization

- Docker v24.0.5

Database

- PostgreSQL 16

Cloud Infrastructure:

[1] All version numbers are as of 23/5-24

- **DigitalOcean** Used for hosting our application and managed databases.

Docker images:

- **mcr.microsoft.com/dotnet/sdk:8.0**: Image containing the entire .NET SDK used for building the application. This image is currently also being deployed.
- **prom/prometheus@2.52.0**: Image containing a Prometheus instance configured by a .yml file.
- **grafana/grafana@10.1.10-ubuntu**: Image containing a Grafana instance configurable through a UI.

NuGet-packages:

- **Npgsql.EntityFrameworkCore.PostgreSQL v8.0.2**: Entity Framework Core provider for PostgreSQL enabling interaction with PostgreSQL databases
- **prometheus-net.AspNetCore v8.2.1**: .NET library for instrumenting applications and exporting metrics to Prometheus.
- **Swashbuckle.AspNetCore v6.4.0**: Swagger tooling for generating Swagger UI for API endpoints, making them easier to test.

2.3 Current State of The System

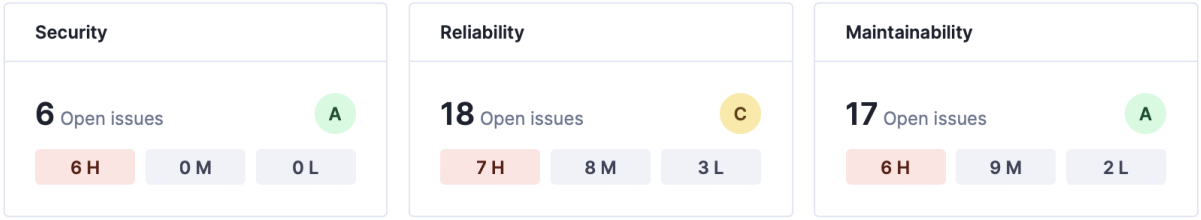

Figure 5: Latest summary from SonarQube showing our security-, reliability-, and maintainability-issues.

Security: relates to us not checking if the model state is valid when using controller actions.
Reliability: relates to some async-problems, frontend-problems, and one unreachable code-problem.
Maintainability: relates to commented out code and some related to the tests file provided to us.

Figure 6: Latest summary from Code Climate showing our code smells and duplication issues.

Code Smells: relates to some methods having more than 35 lines, one method having more than 4 arguments and some methods having too many return-statements.
Duplication: relates to some code-blocks showing up more than once.

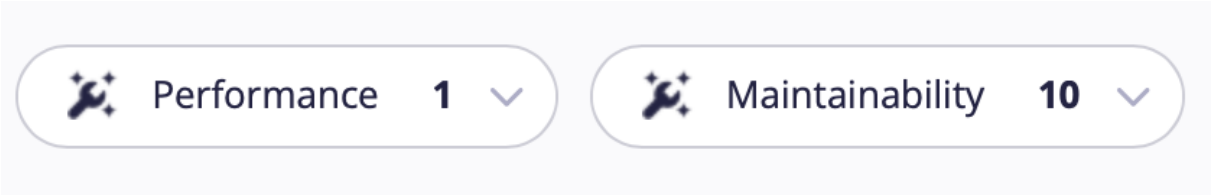

Figure 7: Latest summary from CodeFactor showing our performance- and maintainability-issues.

Performance: relates to us allocating an array with zero-length.
Maintainability: relates to us having empty lines and some closing braces being preceded by a blank line.

3 Process perspective

3.1 CI/CD Pipeline

We have set up our CI/CD pipeline through GitHub Actions consisting of a total of 4 (5 if we count Dependabot) different workflows that trigger throughout the development process.

Autorelease

The autorelease workflow is triggered on push to main as long as there have been made changes to the application (changes to the src-folder). First, it creates a new release tag, using the semantic versioning scheme (Major, Minor, Patch)[2]. By default, it will bump the Patch version on merges to main but can be modified by writing *#major* or *#minor* in the commit message, which then will bump the respective number. When the tag has been created, a release with the version tag and auto-generated release notes will be made.

Continuous Deployment

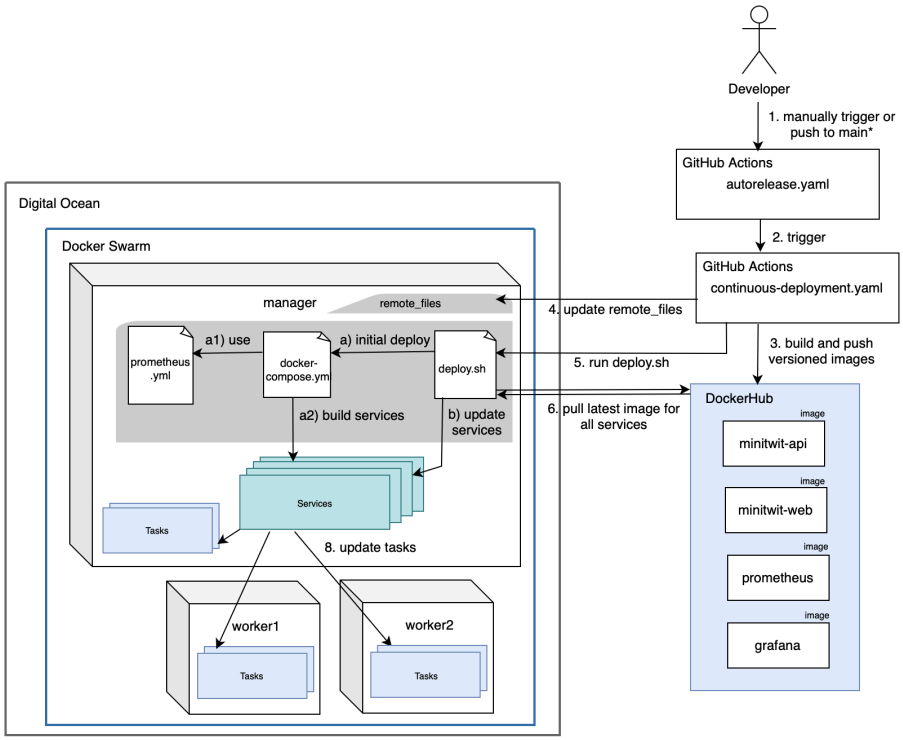

Figure 8: Deployment. Step 7a: These steps run only during the first deployment. Step 7b: For all subsequent deployments, this step is executed. The workflow does not depict the volumes for the services, for this level of detail see the figure of Docker Swarm.

In addition to the steps shown above, the deployment process also tags the Docker images pushed to DockerHub with the version number created in the Autorelease (Section 3.1) as well as with a "latest" tag. This ensures that all versions of our application are preserved on DockerHub.

[2]https://interrupt.memfault.com/blog/release-versioning [2]

CodeQL

The CodeQL workflow is triggered on pull requests to main and has a cron job every Wednesday at 16:20. It scans the code for vulnerabilities and quality issues by using a predefined set of queries. If issues are found, it provides detailed feedback, allowing us to address problems before merging changes into the main branch. In hindsight, we should've decided between either running the workflow as either a cron job *or* on pull requests. As it is now the cron job is irrelevant because the code gets checked every time before merged into main.

Linter

The linter workflow is triggered on all pull requests and can be triggered manually also. It runs a super-linter for C#, GitHub Actions, and Dockerfile, that checks for code quality and best practices throughout all the files in the codebase.

Dependabot

The Dependabot workflow runs weekly, where it checks for updates throughout the codebase and create pull requests when updates are needed. Also to minimize bot spam every week, every Nuget update is getting grouped into one single pull request.

3.2 Monitoring

For the monitoring of our MiniTwit project, we utilize Prometheus and Grafana to store and visualize custom metrics from our system. Prometheus efficiently pulls data from various components and stores it as metrics, such as the current number of registered users and the rate of errors per second/minute. Grafana serves as our visualization tool, providing a range of customizable options for creating dashboards that offer a quick overview of the system's current status.

At the top section of our Grafana dashboard, we monitor the real-time status of the API and Web service. This includes tracking HTTP Requests per second and monitoring error codes. We also monitor the average CPU usage for the minitwit-api- and minitwit-web-services. Additionally, we track average page load times and API response times for various POST requests. The cumulative count of HTTP requests received provides insights into system usage over time, while error response codes highlight areas needing optimization. A pie chart summarizes the distribution of HTTP actions, offering a clear view of user interaction trends. This monitoring setup enables timely identification and resolution of potential issues, ensuring the project's health and performance.

Figure 9: Dashboard monitoring

3.3 Logging

Logging for the application in production has not been set up yet, due to difficulties in implementing it. Read more about that in the lessons section 4.4. The intended approach was to implement the ELK stack (Elasticsearch, Logstash, and Kibana), which are open-source tools designed to help us establish an easy-to-manage platform for logging, with the ability to modify and customize visuals as needed. The intended use of the different tools in the stack is as follows:

- *ElasticSearch* is an efficient search and analysis engine designed to handle the large text data generated by the logs.
- *Logstash* is intended to aggregate and collect logs throughout the codebase.
- *Kibana* utilizes the Elasticsearch indexes, search, and analysis capabilities to create visualizations of the data.

3.4 Security

In the Security and Risk analysis (see Appendix A.1) we have identified a variety of possible security issues. The identified risks highlight valid concerns and several safeguards could be implemented to enhance the application's security posture. Here are some potential changes that could be made:

- Implementing rate limiting and input validation mechanisms would mitigate DDoS attacks and injection vulnerabilities.
- Securing cookies with the `Secure` and `HttpOnly` flags, along with robust authentication and authorization measures, would protect against unauthorized access and cookie manipulation.

- Running containers as a non-root user and securing database connections would reduce the impact of potential container escapes or data breaches.
- Implementing CAPTCHA/reCAPTCHA would protect us against bot spam.

3.5 Strategy for Scaling & Upgrades

Load balancing - Docker Swarm

In our deployment of the MiniTwit application, we employ Docker Swarm to ensure high availability and efficient distribution of workloads across our infrastructure. Docker Swarm is a native clustering and orchestration tool for Docker containers, which allows us to manage a cluster of Docker nodes as a single system. By leveraging Docker Swarm, we can automatically distribute our services across multiple nodes, providing load balancing, fault tolerance, and scalability.

Our Docker Compose file on the manager droplet (remote_files/docker-compose.yml) sets up the necessary services, networks, and volumes. We utilize external volumes to persist data for Prometheus and Grafana.

When defining the services, we use the update strategy `start-first`, which ensures that new instances are started before old ones are stopped during updates, minimizing downtime.

We use placement preferences and constraints, to ensure that exactly one instance of both the web and API service runs on each worker.

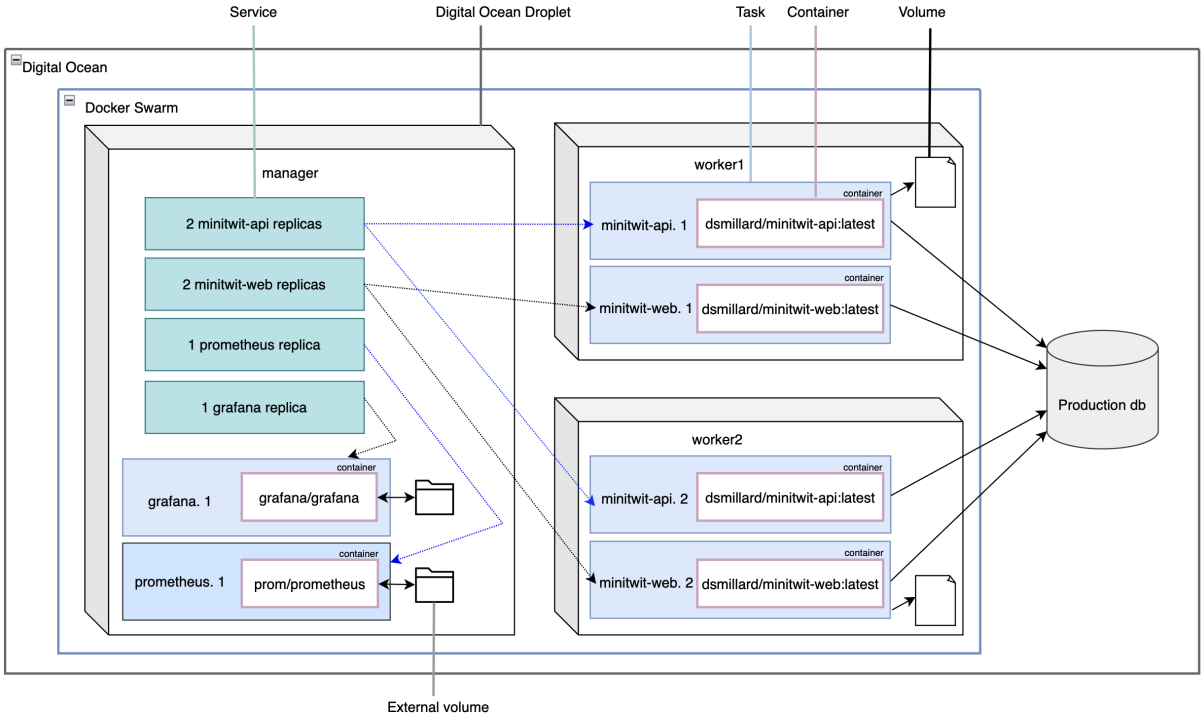

Figure 10: Docker Swarm service overview (coloring of the dotted lines are only for visibility).

Ansible

We have set up Ansible for Infrastructure as Code to create and configure droplets, then initialize and couple them together into a Docker Swarm. Although it should be usable in production, we currently only use Personal Access Tokens (PATs) from other Digital Ocean projects, to use them as test environment for infrastructural changes.

Our first Ansible playbook, "create droplets," uses a Digital Ocean PAT and a fingerprint to create three droplets. Based on their IP addresses and names, it auto-generates an inventory.ini file using a Jinja2 template. This inventory file allows us to run tasks for "all", "manager", or "workers". You can even limit tasks to a specific worker using the `--limit <hostname>` option. Additionally, the "create droplets" playbook includes a feature to delete the droplets and the inventory file by running it with the `--tags teardown` option.

Our second Ansible playbook, "configure_swarm," uses the generated inventory file to configure the created droplets. It installs Docker and Docker Compose, opens necessary ports, syncs files, initializes a Docker Swarm on the manager and joins worker nodes to the Swarm.

We implemented these playbooks later in the project, which allowed us to experiment with our deployment scripts, droplet configurations, and Swarm setups in a test environment. Given additional time, we would have aimed to implement Ansible in production, automate the processes, and utilize it to develop a comprehensive restore strategy.

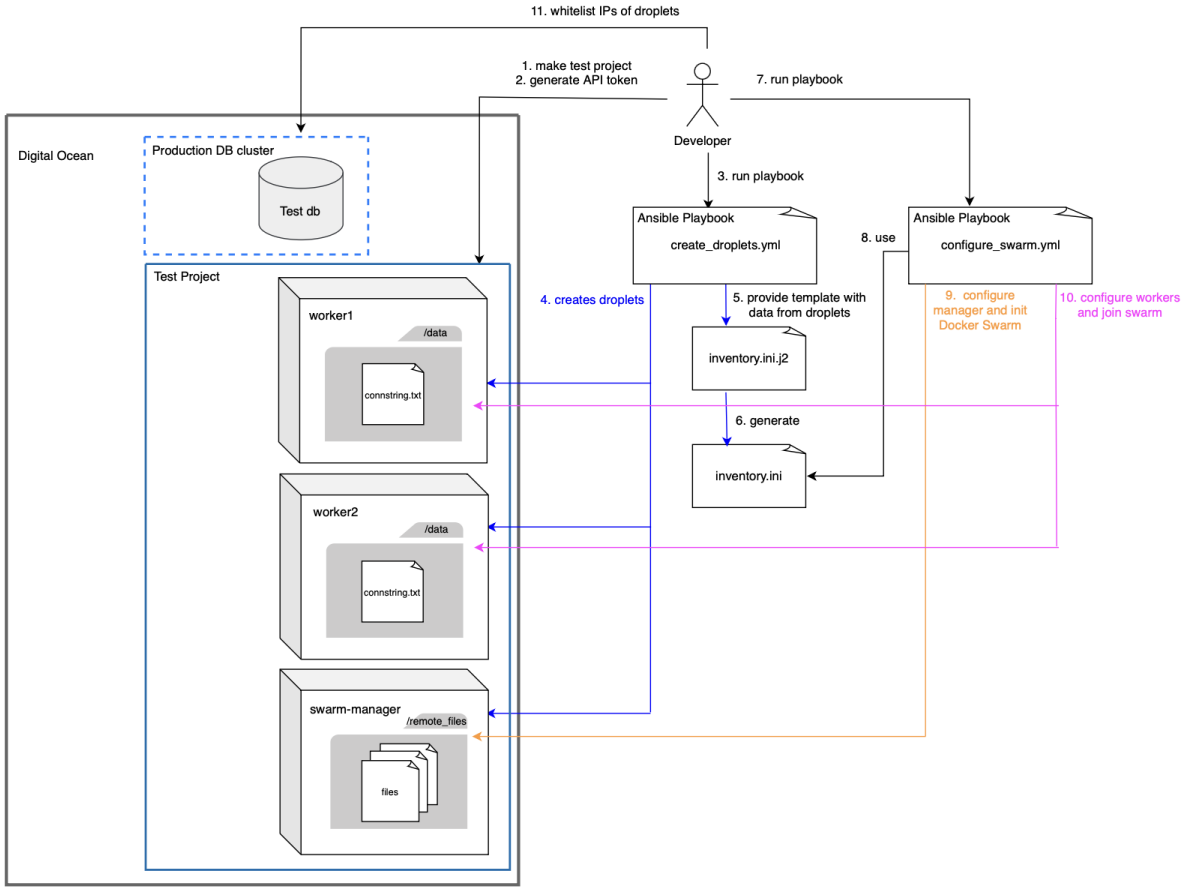

Figure 11: Our Ansible test environment setup, that initializes the docker swarm and makes sure that all droplets are configured with the right files and packages. It takes 2 minutes to create droplets and 2,5 minutes to configure the Swarm.

ansible > inventory.ini

```
1   [manager]
2   swarm-manager ansible_host=64.226.103.131 ansible_user=root
3
4   [workers]
5   worker1 ansible_host=138.68.70.52 ansible_user=root
6   worker2 ansible_host=161.35.20.222 ansible_user=root
7
8   [all:children]
9   manager
10  workers
```

Figure 12: Example of an auto-generated inventory file after setting test-project using Ansible.

3.6 Use of AI

We have been assisted by Copilot. It was primarily used to help with LINQ in our infrastructure as it is easier to ask Copilot to create the query. It did, however, at times give

queries that could be improved **a lot**, and when asking about it Copilot would go ahead and fix the issue.

Copilot was also been great at explaining certain things that we were either unsure of or did not know about, and it often times provided us with good auto completions that we could use (and modify to our liking).

Lastly, with us not knowing much about Grafana and Prometheus, Copilot provided great support in this helping us make our PromQL queries for our dashboards.

4 Lessons perspective

4.1 Refactoring code into .NET

Refactoring the original Python Flask application to an ASP.NET Core application proved challenging. Initially, we scaffolded an entire Razor Page application[3], but due to confusion about the course requirements, we scrapped it and pivoted to a minimal API approach[4]. This attempt to create a one-to-one copy of the single Python file resulted in complications due to the differences in the programming languages, including static versus dynamic typing and handling database connections and queries. In the end, we decided to import our Chirp application from the BDSA course due to time limits. The lesson learned is that refactoring to another language requires ample planning of the necessary frameworks and the architecture of the new application.

4.2 Migrating from SQLite to PostgreSQL

We faced challenges migrating from SQLite to PostgreSQL due to differences in the databases' queries. Initially, we attempted to use pgloader to convert our database into a PostgreSQL dump file and execute it on a test database, but this approach failed. Consequently, we created a SQLite dump file and manually converted it into a PostgreSQL dump file by adjusting specific queries that were incompatible between the two systems. We then used the PostgreSQL interactive terminal to execute the modified dump file into our database. Simultaneously, we deployed the latest application image to support PostgreSQL instead of SQLite, which was a minor change because we were already using EFCore in our infrastructure[5].

4.3 Implementing Docker Swarm

When we initially attempted to implement Docker Swarm, we encountered issues with the firewall. We had not realized that specific ports needed to be allowed, which prevented the application from functioning correctly, as it could only be accessed through the workers' IP addresses. After identifying our mistake, we researched firewall settings, which made us more mindful of the ports we needed to open. This also resolved the issue with Docker Swarm.

4.4 Implementing logging

We attempted to implement logging throughout development using the Elasticsearch, Logstash, and Kibana (ELK) stack. However, we never got it fully operational for production use. Early on, we recognized the value of logging when our program crashed randomly after deployments. Without logs, we had to guess and apply quick fixes. Eventually, examining the Docker container logs revealed the crashes were due to deadlocks caused by high request loads. Understanding this allowed us to implement asynchronous methods to handle tasks better and prevent future deadlocks.

[3]https://github.com/itu-devops-groupn/itu-minitwit/commit/deaa7d4d4cfa54782cb72cd7417c7f033fc5aac5
[4]https://github.com/itu-devops-groupn/itu-minitwit/commit/b892da1d2f537e4c542e64fdcd49ad428b58e3b2
[5]https://github.com/itu-devops-groupn/itu-minitwit/commit/c721dfab96e5fd7ba91f71cdb948357a693a86c2

Initially, the setup seemed robust, and we had a functional local project within the first week of trying to implement logging. But transferring this to the MiniTwit-project proved problematic, with issues related to ports, networks, and authentications preventing proper read/write operations. We then discovered a setup script to configure the ELK stack, which resolved these issues theoretically.[6] However, implementing this solution late in the process made us hesitant to integrate it into production, hence it remains in a separate branch.[7]

4.5 Reflection on DevOps-workstyle

The DevOps philosophy has required every developer in the group to be able to do "Operations" work. We have facilitated this through our CI/CD workflow. This differs from a lot of our earlier projects since they have not required deployment to a server and have usually only involved a single final build and hand-in. The only course that has required continuous deployment was BDSA from our 3rd semester and for that, we did do DevOps style deployment with automation. During the project, we focused on automating as much of the program as possible, as well as keeping it as "clean" as possible to use. Not having a proper test environment, in the beginning, slowed our development process down. Partly due to the public error tracking, we were overly cautious about making changes to our deployment process. A stable testing environment is crucial for smooth development cycles and fostering innovation. Without it, there's a risk of accepting solutions that work without optimizing them.

[6] https://github.com/itu-devops/itu-minitwit-logging/tree/new-elk-stack[1]
[7] https://github.com/itu-devops-groupn/itu-minitwit/tree/feature/setup-logging

5 Literature

References

[1] ITU-DevOps GitHub. https://github.com/itu-devops/itu-minitwit-logging/tree/new-elk-stack, 2024.

[2] Tyler Hoffman. https://interrupt.memfault.com/blog/release-versioning, 2021.

A Appendix

A.1 Security & Risk Matrix

A. Assets

- Web application
- Web API
- Grafana dashboard
- Prometheus instance
- Database

Threat Sources

- DDoS attacks due to no request limits
- Cookie manipulation
- Root-user on Docker images
- Automated bot registrations due to lack of human verification

Risk Scenarios

- Attacker performs DDoS attacks on our application, bringing the application down.
- Attacker sends a lot of requests, bringing our application to a halt due to many requests and potentially breaking the application or API.
- Attacker registers a user and changes the username cookie to someone else, which changes the user's identity and could lead to unauthorized access to personal information.
- Attacker manages to break out from the Docker container, giving the attacker root access to the host machine, potentially bringing down the application or breaking the infrastructure.

B. Risk Analysis

Likelihood

- DDoS attacks and sending a lot of requests are likely to happen should an attack occur. They are the easiest to perform as we have no limit on requests to our application.
- Cookie manipulation could be likely to happen should an attacker know
- Cookie manipulation could be likely to happen should an attacker know the possibility of it, and if they know the format for the username.
- Breaking out from the container and exploiting root-user access is very unlikely to happen. It would require an attacker with a high level of skill and knowledge to perform such an attack.

Impact

- Bringing down our application is troublesome and would require the service to be restarted, but due to us using Docker Swarm, this would happen automatically if a container crashes. However, should the host machine be brought down, it would have a higher impact because it would need a manual restart.
- One could potentially use another user's identity on our application with cookie manipulation, which could mean that someone could send messages as someone else and harm someone's reputation.
- Breaking out from the container could lead to us losing ownership of our application and could mean that secrets could be exposed. One could also access our database if they broke out from a container due to us storing the connection string on the host machines in plain text.

Risk Matrix

	Low Likelihood	Medium Likelihood	High Likelihood
Low Impact	Incorrect data entry	Incorrect user input validation	Sending many requests
Medium Impact	Unauthorized access	Cookie manipulation	Bot spam
High Impact	Breaking out from the container	Data breach	DDoS attacks