ITU DevOps Project Report

ITU DevOps 2025 Group a

Christoffer Grünberg gryn@itu.dk
Rasmus Rosenmejer Larsen rarl@itu.dk
Mathias Labori Olsen mlao@itu.dk
Anthon Castillo Hertzum acah@itu.dk
Jacques Puvis de Chavannes japu@itu.dk

Contents

1 System

1.1 Current Deployment
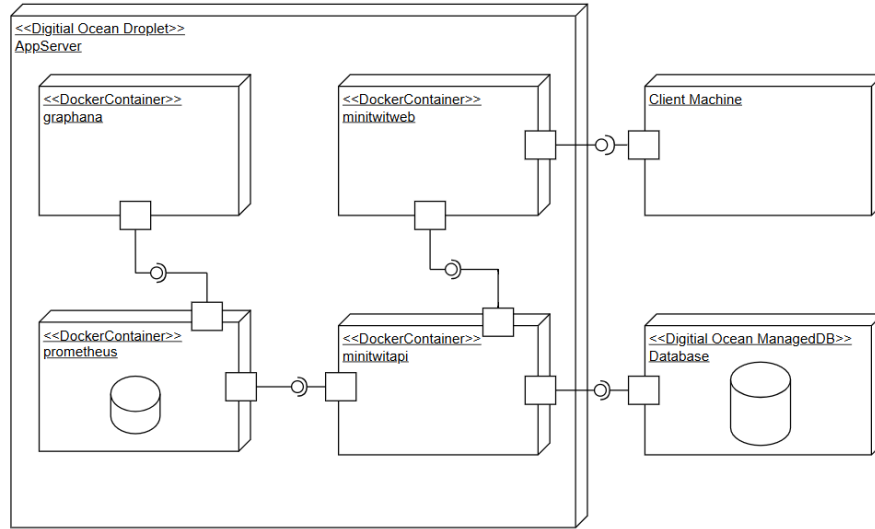

Figure 1: Current deployment of *itu_minitwit*
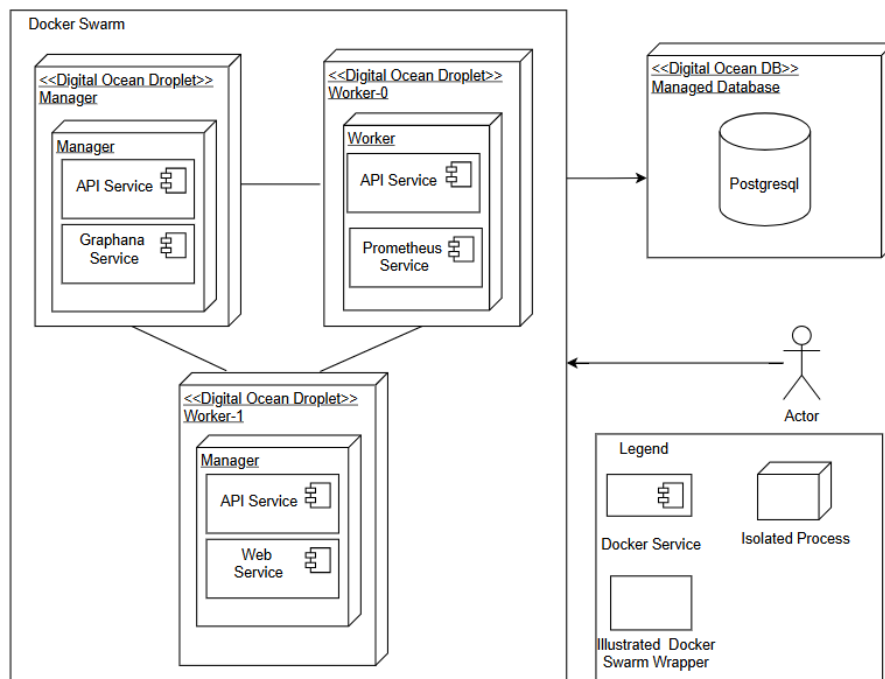
1.2 Swarm Deployment

Figure 2: Docker swarm deployment of *itu_minitwit*

The system is in the process of being converted to use Docker Swarm instead of a docker network. This was to increase the crash resilience by replicating the services, so that if the application ever did crash, it would still function and be up and running. Currently, the Swarm can be set up by using terraform, but it only deploys an empty Swarm. The idea was to populate the Swarm as a step in the deployment chain, where we would SSH into the manager node and then deploy it from there. However, there were complications with the Prometheus config file. It can't mount to the swarm in the same way as it can to a single machine with docker compose.

One thing missing in our terraform configuration is to correctly open for the firewalls. We are currently not opening the correct ports for the internal DNS network to route between docker services.

1.3 Used Technologies

1.4 Used Technologies

- Programmed in C# .NET 9 with ASP.NET as web framework and Blazor frontend.
- Containerised with docker.
- Onion architecture for code structure
- Dependencies:
  - EFCore
  - PostgreSQL
  - Digital Ocean
  - Docker
  - Prometheus
  - Grafana
  - Serilog
  - Seq
  - Terraform

1.5 Interactions of subsystems

Below are two sequence diagrams, showing the flow when a user posts a message. It was the simulator, that would post the message instead, it just communicates directley with the api, instead of the web. Otherwise they are the same from the api controller and onwards.

JohnDoe   web   api.controller   api.service   api.repository   database

Post message "Hello"

CreateMessage("JohnDoe", "Hello")

PostMessage("JohnDoe", "Hello")

PostMessage("JohnDoe", "Hello")

«get user»("JohnDoe")

alt   [user exists]

user

<<add messages> "Hello"

messages created

messages created

null

«exception» UserDoesntExist

break   [user doesn't exist]

«create user»("JohnDoe")

«create user»("JohnDoe")

«create user»("JohnDoe")

user created

«user created»(user)

«add message»("JohnDoe", "Hello")

<<add message> "Hello"

message created

message created

true

«HTTP no content» 204

add new message to feed

notify UI of changes

updated message feed

JohnDoe   web   api.controller   api.service   api.repository   database

---

sim   api.controller

send messsage "Hello"

same as for web

«Http no content» 204

sim   api.controller

1.6 Current state of System

As seen on the image below, the reliability and maintainability score is as high as can be. The same goes for code-duplication. For explanation of security, see the security section in the process overview.
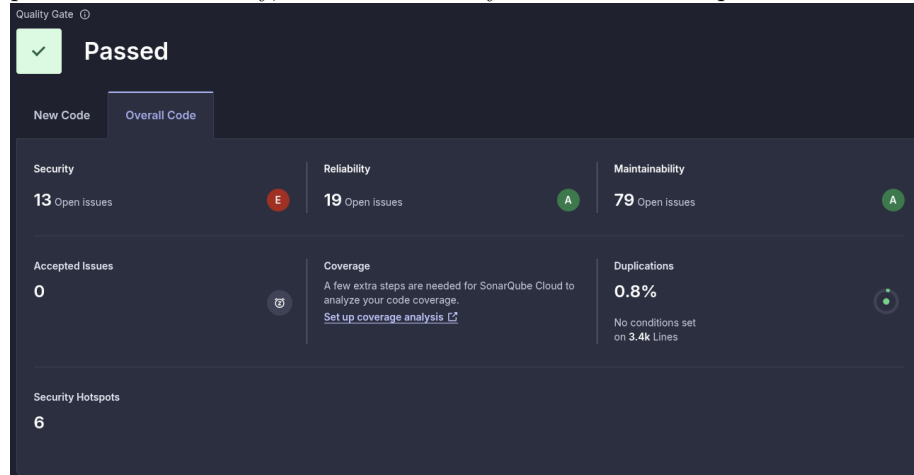
2 Process

2.1 Provisioning

Vagrant was used to provision virtual machines, specified with a Vagrantfile. In the Vagrantfile, you're able to provision several virtual machines at the same time (fx the web app and the database), define and install their dependencies. This allows for an easy, streamlined way to always provision VM's without having to rely on configuration a specific user interface from various VM providers. This means that we are able to use the Vagrantfile with several providers, only having to change the vm.provider.

We are changing from Vagrant to Terraform. Terraform is infrastructure as code. This allows us to define how we want our infrastructure to look, and let terraform figure out how to get there. Terraform is currently configured to provision DigitalOcean, having 1 manager and 2 workers connected in a Docker swarm.

We have a folder called remote_files which contains files that needs to be pushed to the server. This folder contains:

- Deploy script
- Docker compose
- Prometheus configuration

The folder also contains the Grafana Dashboards. These Were added during report writing and are currently not being utilized. The dashboards are also

stored in a volume on the server for persistence.

2.2 Workflow

For our entire developing process we've used trunk-based development with each feature being developed in a separate branch. We use GitHub actions for CI/CD and GitHub issues for task management. Our workflows include building, testing and deploying the code. On each pull request to the main branch, we first run the 'changes-to-pr-to-main' that checks if the pull request has a label, followed by 'commit-pr-to-main' which runs a handful of jobs:

- Check-for-warnings
- Build
- Test
- Run-simulation test script
- Sonarqube-analysis

These jobs are there to ensure that the codebase still works as intended on the branch that the developer has worked on. The important note is that the "run-simulation test script" could have http requests that could time out, however, we ensured that if there was only a couple of timeouts we could deduce that the codebase still worked as intended. This was primarily to confirm that if we had 10's or 100's of timeouts, we could be sure that the codebase was broken.

2.3 Monitoring

We monitor through the use of Prometheus and Grafana. Our application expose an endpoint using the OpenTelemetry nuget package for exporting telemetry data that Prometheus can understand. Prometheus then scrapes the endpoint with an interval of 5 seconds, configured in the Prometheus.yaml file. Prometheus saves the data in a times series database. This database is queried by Grafana which visualises the data in a custom dashboard. Our custom dashboard has been built on top of the "ASP.NET Core" dashboard published by the .Net Team (https://grafana.com/grafana/dashboards/19924-asp-net-core/). We have added a few custom panels. The most interesting being a table that shows total amount of request per status code for each endpoint. Another useful panel we made plots the request duration of different endpoints.

Status code panel

```
label_join(
  http_server_request_duration_seconds_count{
    job="$job",
    instance="$instance",
    http_route!=""
  },
  "method_route",
  " ",
```

```
  "http_request_method",
  "http_route"
)
```

Status code panel

```
rate(
  http_server_request_duration_seconds_sum{
    job="$job",
    instance="$instance", http_route!=""
  }[5m]
) / rate(
  http_server_request_duration_seconds_count{
    job="$job",
    instance="$instance",
    http_route!=""
    } [5m]
)
```

2.4 Logging

We rely on serilog for generating and sending logs to our log visualiser Seq. Our logging strategy is quite extensive, since we have had a lot of troubles with our application, we thought it was better to have more, and then not keep them for as long, to see if they could help us sort out our errors/bugs. It is as follows We log when we raise exceptions and when exceptions are caught, this to help us see how erros where propecated through the system. We log execution time of methods called, this was done as to help us see if there wehere methods bottle necking us. We log the input and output of methods called, this way we can observe if they behave like we expect them too.

2.5 Security

We had set SonarQube up to comment on every pull-request we had made, to ensure that the pull-request had passed the quality gate. A segment of this quality gate, was to ensure that there were no security hotspots. Not only did SonarQube show where the hotspots were, but it also explained why this is a hotspot, and how to fix it. In our last release, we still had security hotspot, more specifically an "Open Redirect Vulnerability". This vulnerability results in an "E-score", but the rest is rate B or above. This could have been fixed, by creating a "allow-list" of safe relative paths to redirect to.

2.6 Scaling

Regarding the scaling of our application, we are in the transition of moving from docker compose to docker swarm. However, we are using docker compose that composes an API and MiniTwit dockerfile. Our intentions are to set up a

declarative IaC using Terraform and with that set up a docker swarm cluster that can handle the scaling of our application. Unfortunately as of now, we haven't fully integrated this structure because of some complications with the implementation.

2.7 AI-assistant

The use of Chat-GPT has been used to some extent. Fx to more easily understand (at the time) complex notions in the topic of operations. Various Dockerfiles were created with the assistance of AI, in order to more effectively "learn-by-doing". Furthermore, the assistance of AI proved efficient when having to translate the python api into C#. Problems did arise from this, though. As an example, the AI rewrote the returned status codes, which meant that it wasn't compliant with the simulator. This ended up being an achilles heel, since we spent a lot of time trying to diagnose the problem with the simulator.

3 Reflections

The difficulties of the project were primarily to translate the simulation_api from python to C#, but also in implementing new features as well as learning new and unfamiliar technologies that should be integrated with the project.

3.1 Evolution

3.1.1 Technical Debt

The group experienced first hand, how much technical debt can slow down the development process. When we first shipped our code, we had not made sure that all the simulator tests passed. Because of this, our database was missing initial users, which gave us simulator errors. This was likely due to our implementation of the simulator api that would handle the data to insert into our database. Because of these errors, we later attempted to insert the missing users into the database. However, this created a new error where our VM would crash and was never resolved. As of now (28-05-2025), the API will still create any users who aren't registered in the database.

3.1.2 'Dev' and 'Ops'

In the beginning of the project we had a lot of work that needed to be done. This was for instance translating the simulation whilst also setting up the CI/CD pipelines. We also had to start using containerisation with docker and vagrant. This led to a split in the group between the developers, who started translating the simulation to C#, and the operators, who started setting up the CI/CD and containerisation. The difficulties lied in the communication between these two groups and how both parties could get up to speed with their respective work they've made. A logbook was created to keep track of any work that has been

done, but was unfortunately disregarded by other tasks and hasn't been updated since 07/02/25 (according to the log.md file).

3.1.3 Refactoring

When rewriting the code to C# and adhering to the 'minitwit_sim_api.py' from session3, we weren't thorough enough when analyzing the specs. This resulted in us pushing code to production, which 'seemingly' followed specs from the aforementioned file. After long contemplation on why it didn't work, we took a step back and properly analyzed the api specs. Lesson: Do it right the first time.

3.2 Operation and Maintenance

3.2.1 Logging

We had an experience, before we were introduced to logging in class, where our VM crashed due to extensive (and redundant) logging. We logged to console, docker logs and files. This resulted in the bloating our Digital Ocean droplet with sizeable logs. What could be done differently, was to make to only log once and automatically delete old ones which weren't needed anymore.

3.2.2 CPU overload

We experienced a CPU overload in our droplet. The CPU would spike to 100% and sometimes exceeding that (due to Digital Ocean limiting the CPU size of the droplet). This resulted in a crash of the droplet. Unfortunately, as of now (9/5/2025) we haven't found the reason for why this is happening. For future reference, we should have a more thorough testing suite.
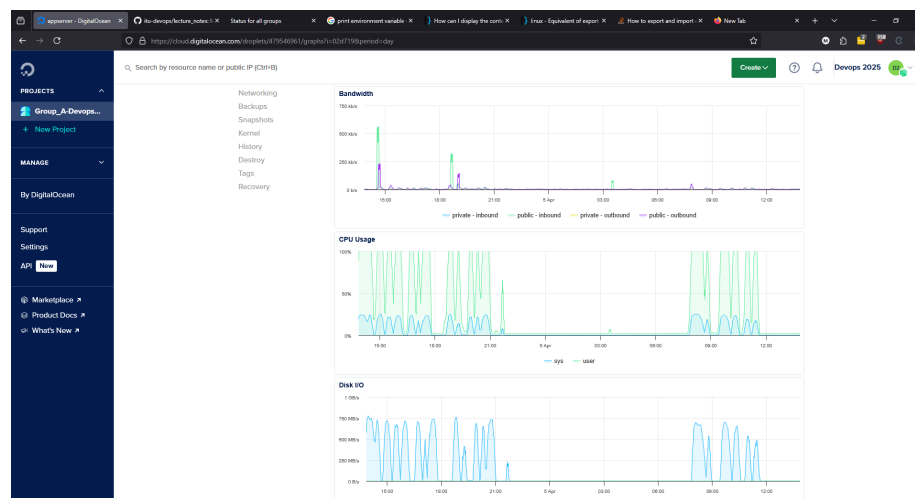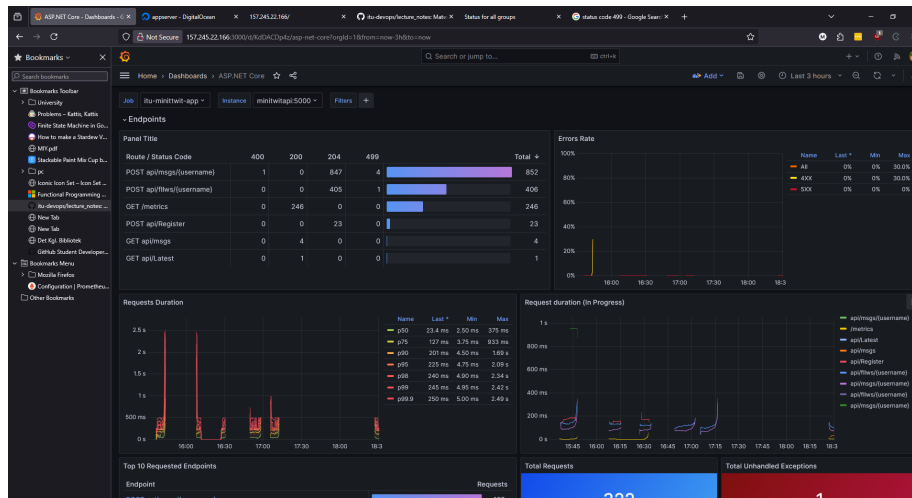
3.3   Grafana

During session 6 we were tasked to add monitoring to our application through prometheus using grafana for visualization. This included monitoring over the requests for different endpoints in the application. While we did add monitoring to the application, we didn't use it as much as we should have. There wasn't much time to monitor the application with our schedule this semester. This was made worse by having to use digitalocean's dashboard as well, as we struggled to add CPU usage to Grafana. One thing that could have ameliorated this issue was alerts.

3.4   DevOps Reflections

Despite the fact the the group did not solve all of the issues, we still learned a lot about the DevOps work. When comparing the workflow to other projects like *Chirp!* in BDSA, one big difference was the deployment strategy. Using Docker Images to deploy instead of just copying the entire codebase was a great learning experience. We could have used the built in versioning in Docker Hub to better be able to roll back to working versions of the codebase. This would have been helpful, when faulty code was deployed to production.

3.4.1   Workflow Learnings

The group also learned a lot working with workflows. In previous projects most of the logic was in one large workflow. In this course we worked with splitting up the responsibility into smaller workflows, then these where called by composing workflows. This way of working made it easier to find errors and read the intended functionality of the workflows.

Another learning was incorporating static analysis into the integration chain.

This gave better clarity of the code quality and helped reduce errors being merged with the main deployment.