Group J - DevOps Report

May 23, 2023

Email:	Name:
behv@itu.dk	Bertram Kosmo Hviid
paab@itu.dk	Patrick Wittendorff Abarzua Neira
tbru@itu.dk	Tobias Skovbæk Brund
siar@itu.dk	Silas Roien Arildsen

Contents

1 System Perspective

1.1 Design and architecture

1.1.1 Design

The first step of taking over the Minitwit system required a refactoring to a different technology stack. As a group we chose a full .NET stack because every group member felt comfortable and experienced with both backend and front-end development using C#. The group wanted to focus on learning the new tools and working in a "DevOps" manner, and we recognized that experience would simplify the process.

For the front-end we went with Blazor-WASM and for our backend we decided used a traditional ASP.NET API. When refactoring the Flask application we did it in small steps making sure to preserve the functionality and style of the original.

1.1.2 Deployment architecture

The overall architecture of our deployed MiniTwit service can be seen on figure 1. As seen on the figure all our nodes in the system are run as virtual machines (or droplets) on DigitalOcean, we even decided to self-host the database to experience the challenges that go along with doing so.
As scaling/load balancing was one of the weekly tasks, we introduced it into the architecture in the form of a Docker swarm. We deployed our simulation API to a Docker swarm with a total of three nodes, one manager and two workers. Our Docker swarm would replicate NetData globally, a tool we use for monitoring hardware, and only replicate our simulation API on a single node. When deciding on a scaling strategy, we also discussed the possibility of creating a swarm for our front-end. Ultimately, we decided to keep it a single node, since the cost of instantiating a swarm was too high compared to the amount of front-end users.

To do monitoring and logging of the system, a *utility node* was set up. The utility node would host tools like Grafana, Kibana, postgres, and be in charge of collecting the monitoring data through Prometheus. The utility node as such communicates with most other nodes in the system.

1.1.3 Code architecture:

Our system is fully contained in the .NET environment, that is our front-end, backend, and simulator API are all built using .NET. At compile-time every module/project depends on domain model which exists in our MiniTwit.Shared library, as can be seen on figure 2.

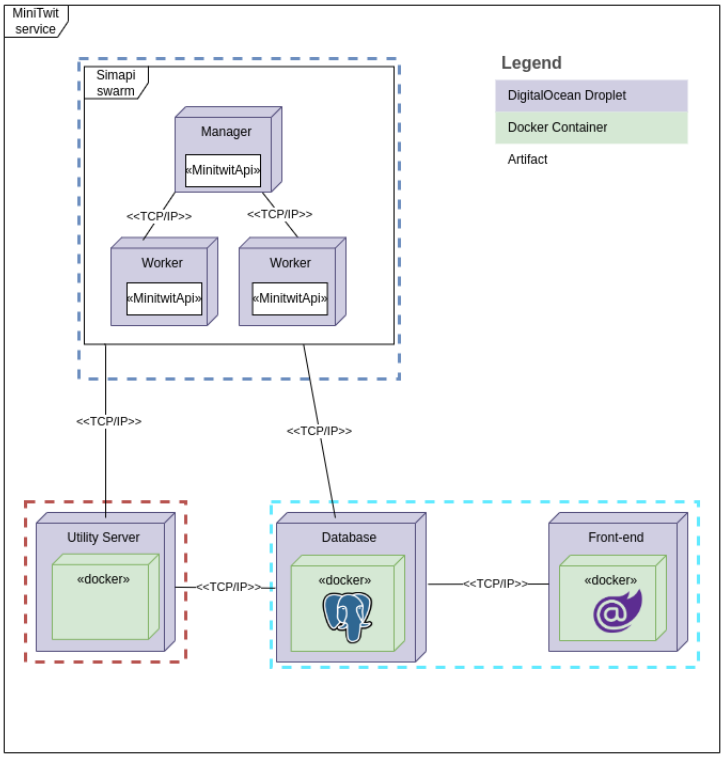

Figure 1: UML Deployment Diagram—Shows the overall architecture of our deployed MiniTwit service. Dashed colored lines indicate different zooms of the architecture. Simulator zoom(4.1), Front-end zoom(4.2), Utility zoom(4.3).

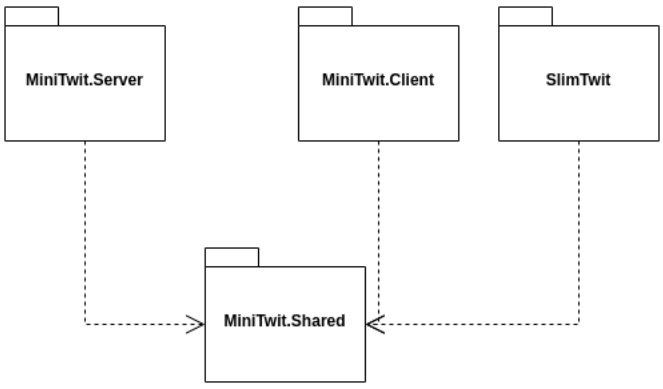

Figure 2: UML Module Diagram - shows the compile time dependencies within our own modules

1.2 The System's Dependencies

1.2.1 Nuget

	Server.Test	e2e	SlimTwit	MiniTwit.Shared	MiniTwit.Client	MiniTwit.Server
Microsoft.EntityFrameworkCore.InMemory	x		x			x
Npgsql.EntityFrameworkCore.PostgreSQL			x			x
Microsoft.AspNetCore.Components.WebAssembly.Server						x
Microsoft.AspNetCore.Mvc.Testing	x					x
Microsoft.EntityFrameworkCore	x		x	x		x
Microsoft.EntityFrameworkCore.Design						x
Newtonsoft.Json				x	x	x
Serilog.AspNetCore	x		x			x
Serilog.Sinks.Console	x		x			x
Serilog.Sinks.Elasticsearch	x		x			x
SQLite					x	x
Microsoft.AspNetCore.Components.WebAssembly.DevServer					x	
Bogus				x		
Microsoft.EntityFrameworkCore.Sqlite	x			x		
Microsoft.AspNetCore.OpenApi			x			
prometheus-net			x			
prometheus-net.AspNetCore			x			
Swashbuckle.AspNetCore			x			
coverlet.collector	x	x				
FluentAssertions	x	x				
Microsoft.AspNetCore.TestHost	x					
Microsoft.Data.Sqlite	x					
Microsoft.NET.Test.Sdk	x	x				
xunit	x					
xunit.runner.visualstudio	x					
Microsoft.Playwright.Nunit		x				
NUnit		x				
NUnit.Analyzers		x				
NUnit3TestAdapter		x				

Table 1: The full list of all 29 Nuget dependencies in our system, and what dotnet projects they belong to.

1.2.2 Tools & Other Dependencies

- Docker for containerization & Swarm
- DigitalOcean for hosting
- Terraform for infrastructure as code
- Prometheus for monitoring
- Grafana for visualizing data from Prometheus
- Dependabot for updating dependencies automatically
- SonarCloud as code review tool
- Serilog for logging

- Kibana for visualizing data from Serilog
- NetData For monitoring health of hardware
- Playwright / nUnit for E2E testing
- xUnit for unit testing
- ~~Vagrant for VM configuration~~

1.3 Important interactions of subsystems

A very important interaction for service is the interaction with the database which both the front-end system and simulation API need to do. The interaction is similar for both systems, both systems use our *TwitContext* to query and save entities in the database, this interaction can be seen on figure 3.

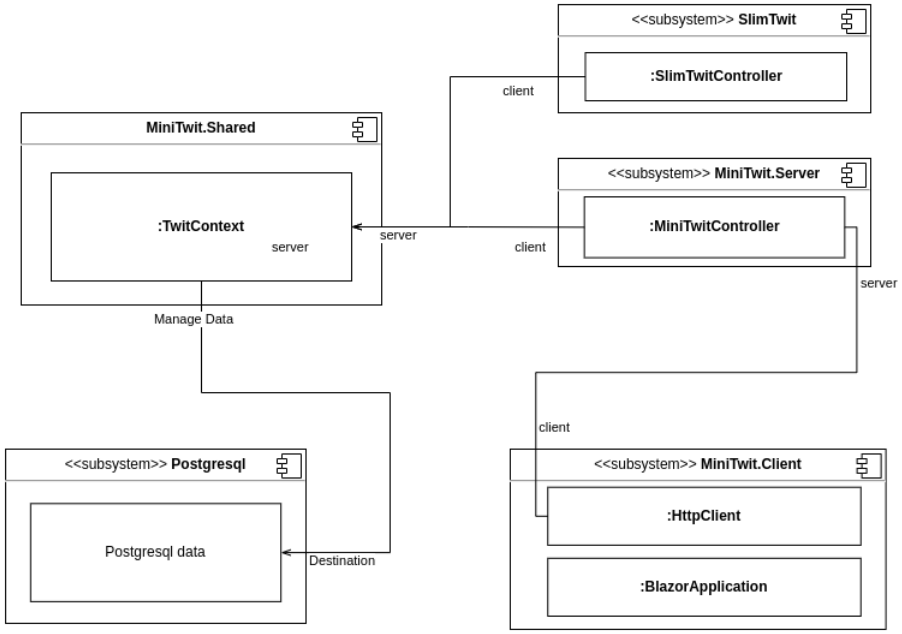

Figure 3: Components and Connector diagram showing how our system reaches the database through EFCore

1.4 The current state of our systems

The system is fully containerized, meaning it is independent of the development and deployment environment. Similarly, the containerization allows deploying the system to a Docker swarm increasing availability. To increase the reliability, the CI/CD pipeline now includes quality gates in the form of static code analysis and tests. Since taking over the Minitwit system it has been refactored to a statically typed language and the system receives an *A* in the maintainability category in the Sonarcloud static code report [1] (which should be taken with a grain of salt, as there are still 15 code smells and at least two known bugs).
For the current state, it is important to note that the database runs incredibly slow. Some often done SQL joins take upwards of seconds and regularly going beyond 10 seconds.

We would have liked to improve the system by:

- Implementing the repository pattern for the database interaction to reduce the amount of duplicated logic for front-end and simulation API.
- Properly running end-to-end tests in the CI pipeline.
- Successfully creating indexes for the database.
- Making the front-end design more modern

1.5 License

We did not discuss licenses while the project was going on, but we would likely have gone with GPL 3.0 or Apache 2.0, as these are the more restrictive out of the ones that occur in our dependencies, and we want to keep it open source. We used a tool called choosealicense.com to reach this conclusion.

2 Process Perspective

2.1 Organization of the Team

We have primarily met on Tuesdays at ITU before the lecture to work on the project together. We usually stay throughout the exercise session to do the exercises and some project work. At the end of the session, we usually distribute the task that should be worked on. For communication, we have used Facebook Messenger for quick day to day coordination, and a Discord server for knowledge sharing, planning, and discussions. We've used GitHub projects as a kanban board for the tasks that needs to be done, but slowly drifted away from using this towards the end of the project, mostly because the amount of tasks got more manageable to the team size.

[1] Sonarcloud report can be here: `https://sonarcloud.io/summary/overall?id=bhviid_GroupJ_Dev_Ops23`

2.2 Organization of your repository

The whole system is stored in a mono repository on Github. We never found a reason to split our repository up. We did not reach a size of the codebase, where we lost track of the structure, and our layered architecture did not make much sense to split up into multiple repositories.

2.3 Branching strategy

During this project, to avoid risking downtime during deployment, we used the GitLab Flow branching strategy. However, after implementing Docker swarm with rolling updates, we've been less concerned with downtime. For this reason, had we continued work on this project for longer and given our team size, we would have used Trunk-Based branching strategy. We think having small lived branches that get merged frequently to the mainline is the most effective way to incrementally and quickly deploy new features, which is the whole point of DevOps.

2.4 CI/CD

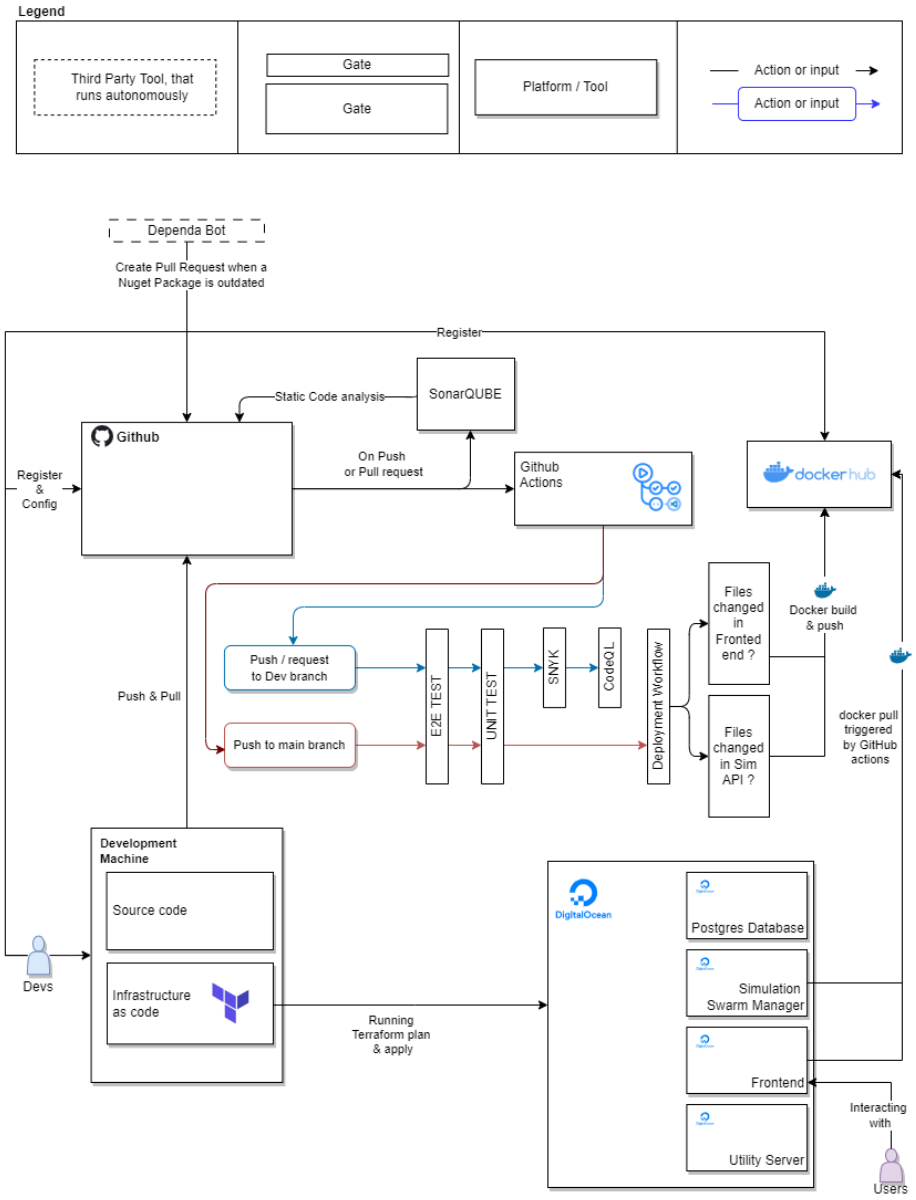

Figure 4: Our current pipeline as of finishing the project

Our CI/CD relies heavily on GitHub Actions, and we actually ended with a pipeline similar to our planned pipeline.8

2.4.1 Continuous deployment

It happens when a push has been made to the main branch, which upon agreement (meaning no enforced restrictions on GitHub) happens when we successfully push to the main branch, usually by creating a release.
When a successful push has been made, we run our test suite, which acts as a gate for the following steps. Because we are using a mono repository, we then check whether a change has occurred in the frontend folder or in the backend folder and build and push the docker images to docker hub accordingly. This led to us having two different docker images for the frontend and for the simulation API, respectively. If an image is successfully build and pushed, we then use a workflow to ssh into the correct machine and do a docker pull. The idea behind this workflow was that we didn't like that changes to the frontend killed the API server and vice versa, and ultimately meant that we split the API and frontend into two different services hosted on different servers.

2.4.2 Continuous integration

It happens whenever a push or a pull request occurs on the development branch. Just like our CD workflow it runs our unit test suite, and you could argue that it makes the testing suite in the deployment workflow redundant, but given that we didn't have any enforced rules on pushing to the main branch, we felt that it was best to keep it there as a safety. We also don't run the E2E test in this workflow, as the E2E in playwright didn't work nicely with GitHub Actions, but would have worked similar to the gate for the docker build & push in the CD where it would only run on changes in the frontend folder. It then runs linting for our docker files as well as image security using Snyk. We then perform code security analysis on our CSharp code using CodeQL. We also use SonarQube on a pull request to check for code smells, vulnerabilities, security Hotspots, Code smells. Our CI/CD might seem disconnected from each other in our current workflow and this is a product of our current branching strategy. When we eventually would have made the switch to trunk based branching, we would have liked to combine the two as a single workflow. This workflow would run on changes on the main branch as we believe that CI is a step to reach CD, and it therefore doesn't make sense to have them as separated as we currently do.

2.5 Development process and tools

We organized our work with a Github Project which allowed us to easily keep track of issues and their progress. Dragging these issues around in a Kanban board gave a nice overview of what needed to be worked on. We also used some automation that would add issues on GitHub to the kanban board automatically.

2.6 Monitoring

2.6.1 NetData

NetData helps us monitor the performance and health of the hardware of our system's nodes, however the usefulness of NetData got diminished when we switched our simulation API to be distributed through docker swarm as we only received performance metrics for the current main worker.

2.6.2 Prometheus

We used Prometheus to monitor our application's performance, collect metrics from various components and gain insights into the health and behavior of our infrastructure. We collect service specific metrics from our simulation API like number of registered users and the amount of twits sent.

2.6.3 Visualization

Grafana has allowed us to take the data collected from Prometheus, and visualize it in a dashboard.

2.7 Logging

We log all the request being made by our users, and sort the logs by assigning them levels such as Information for when things go as planned, Error when there occurs an error and Fatal it there occurs a critical error. We use Serilog, a Nuget package, as an alternative to Logstash, for sending logs to Kibana.

2.8 Brief results of the security assessment.

Using a Kali Linux Docker container as described in the Security Exercise[2], we scanned our site with WMAP in Metasploit, which yielded no warnings for our system at the time.

2.9 Strategy for scaling and load balancing

The Docker Swarm manager balances requests for our simulation within the cluster.[1] We still have to manually change our docker compose to do more replications but would have loved to have some kind of system in place to do this automatically like Kubernetes. We could easily have expanded to use docker Swarm to the frontend and if need be the utility server, but felt like this was unnecessary.

[2] `https://github.com/itu-devops/lecture_notes/blob/master/sessions/session_09/README_EXERCISE.md`

2.10 AI-assistants

2.10.1 Explain which system(s) you used during the project.

We have used ChatGPT lightly as an alternative to asking a TA and mostly for trivial questions as it has a tendency to lie when asked non-trivial questions. We have also used for generating skeleton files for things like docker-compose etc. this works fine, but you need to know what it is doing as it is very capable of making mistakes.

2.10.2 Reflect how it supported/hindered your process.

I would say it aided in our process. It is great at giving more precise information whenever a tools' documentation is lackluster, it also is good at writing boilerplate code / configs, but you have to review the output as it is often faulty, outdated, or straight up wrong.

3 Lessons Learned

3.1 Evolution and Refactoring

We learned a lot about Blazor's pros and cons. On one hand, it works really well with the rest of the .NET ecosystem. On the other, we experienced significant performance issues with Blazor, and sometimes found it to be difficult to work with. In the future, we would consider using a different technology for our front-end.

3.2 Operation

3.2.1 Terraform

Terraform was a breath of fresh air when working with a larger amount of nodes, compared to Vagrant. It was way easier to define a larger infrastructure in Terraform, since that's the main purpose of the tool. We would have liked to have a way to store our state, such that every member could have done a "terraform destroy" instead of it being only local to the machine that launched the infrastructure. We could presumably have used some kind of bucket to store this, but didn't reach a point where we implemented it.

3.2.2 Containerizing

Defining infrastructure as code becomes considerably more challenging without the use of containers. This became evident when we transitioned from Vagrant to Terraform, as we had to refactor the entire "Utility" stack to ensure that all services were running within containers(Postgres and Kibana was installed via ssh prior to this). In hindsight, it would have been beneficial and future-proof to have containerized the stack right from the beginning. Doing so would have

provided streamlined management and enhanced scalability, making the overall process more efficient and adaptable.

3.2.3 Pushing secrets to our repository

We had an issue where some secrets were exposed to the public through GitHub. We had a clean record of not pushing any secrets, but when we implemented Terraform, unbeknownst to us, created some files where our secrets were exposed in plain text. We only know this because a tool called GitGuardian wrote us an email and warned us. This tool is not something we opted in for, but we were very grateful that it send us a DM.[3]

3.3 Maintenance

3.3.1 Keeping an eye on the state of deployed machines

We ran into an issue where one droplet ran out of storage due to Docker logs being stored indefinitely. Our simulation API container had accumulated 13 GB of docker logs since it had been started, which turned out to be a little problematic because the droplet only had a total of 23 GB of storage. We realized something was wrong with the droplet when we were no longer able to log into our Grafana dashboard, entering a correct password and username would yield errors. The solution we found was to limit the size of the log file for the container [4].

3.3.2 The danger of having a key person with primary access to all infrastructure

We had an issue where a person with keys to all our infrastructure wasn't available, and we needed to deploy some infrastructure to our Frontend server and because the person with access to the server was unavailable how we had to abuse Github actions to ssh into a machine and try to fix it that way. [5] An issue occurred where our simulation API became unavailable afterward, and we thought it was an issue with how we deployed Grafana and Prometheus. It turned out later to be an issue that occurred when we switched from a managed database to self-managed database.

If we had established a team in Digital Ocean instead of relying on a personal account, all team members would have had access to information such as graphs and IP addresses, enabling them to make direct changes. If we were to redo the project, our approach would involve setting up a Digital Ocean team right from the start.

[3] `https://github.com/bhviid/GroupJ_Dev_Ops23/commit/1f5951c2ce47ed4c367a7ffd01a405f878afeb3a`

[4] `https://github.com/bhviid/GroupJ_Dev_Ops23/commit/38ad93a2724a494d399a140a3fe22fd8038c39d3`

[5] `https://github.com/bhviid/GroupJ_Dev_Ops23/blob/d5614f8a94428a7a0ddd10e24fe6addffa367702/.github/workflows/deploy-without-vagrant.yml`

3.3.3 Being Careful with Infrastructure Changes

Changes can have significant consequences for the infrastructure of the system. When we made the move from a managed database to a self-managed one, we lost Postgres serial counter, which broke the system for some time.

3.4 DevOps

The major difference from previous projects has been the way of working way more continuously. The automation of many tedious processes in regard to testing and deployment has made a big positive difference in the way we've worked. It gives freedom to actually spend time developing instead of focusing our efforts on less productive manual tasks. We believe that adopting a collaborative approach between development and operations, rather than focusing solely on one aspect, has made it easier to create seamlessly integrated solutions. By considering the perspectives of both developers and operators during our work, we have been able to design and implement solutions that effectively cater to the needs of both roles.

References

[1] Docker Documentation. Swarm mode key concepts. https://docs.docker.com/engine/swarm/key-concepts/.

4 Appendix

4.1 Simulation API deployment diagram

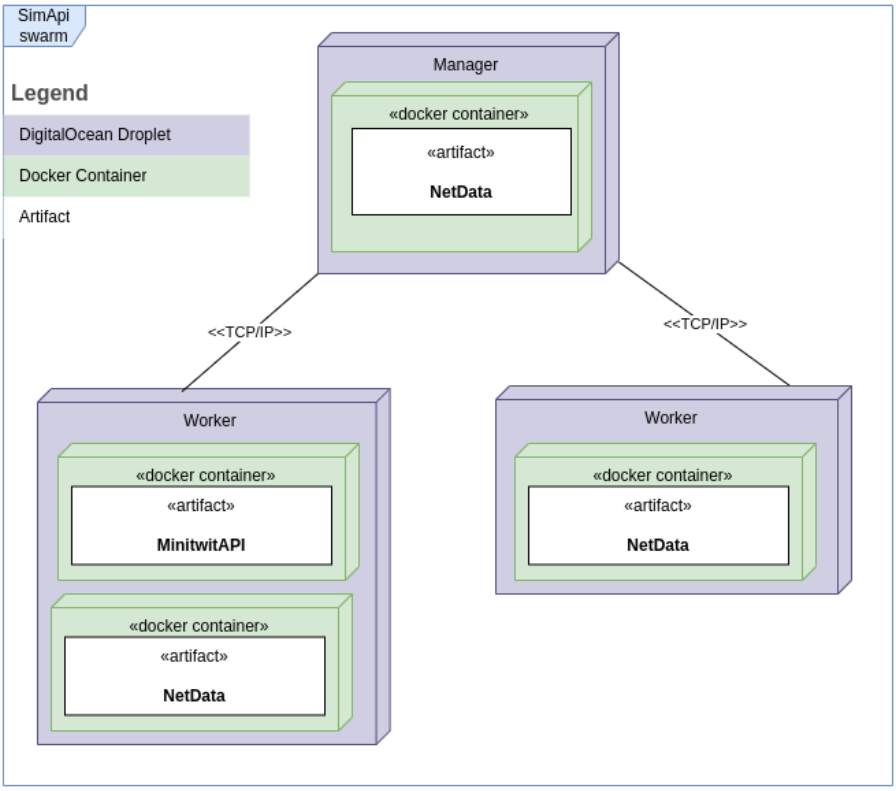

Figure 5: UML deployment diagram - zoom of the simulation API swarm

4.2 Front-end deployment diagram

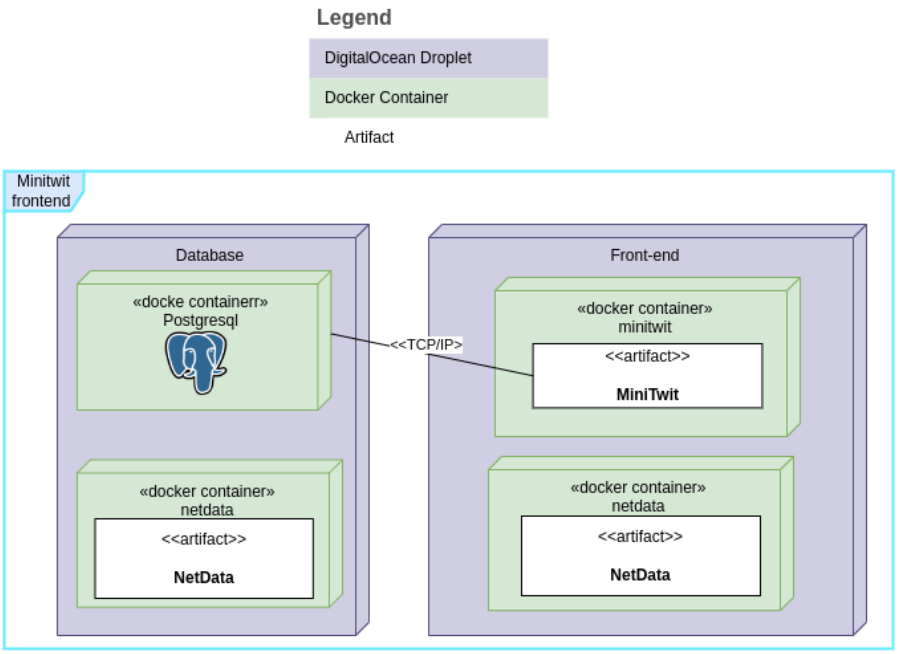

Figure 6: UML deployment diagram - zoom of the front-end setup

4.3 Utility Server deployment diagram

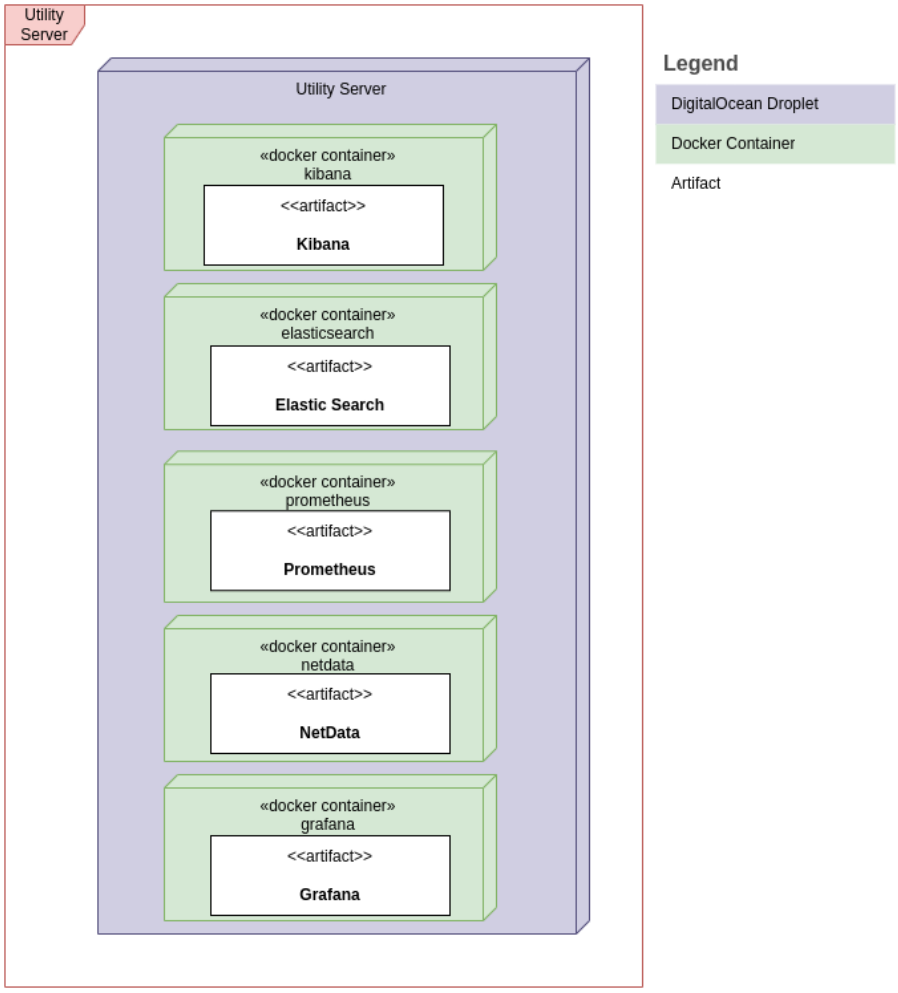

Figure 7: UML deployment diagram - zoom of the utility server

4.4 Pipeline from guest lecture

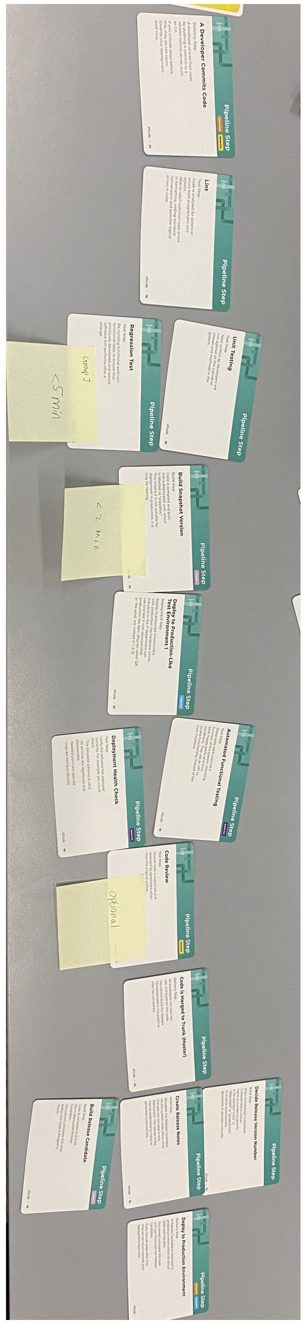

Figure 8: Initial ideas for CICD Pipeline

4.5 Licenses of dependencies

.NET license: MIT https://github.com/Microsoft/dotnet/blob/main/LICENSE
EntityFramework Core: MIT https://github.com/dotnet/efcore/blob/main/LICENSE.txt
ASP.NET CORE: MIT https://github.com/dotnet/aspnetcore/blob/main/LICENSE.txt
Npgsql: PostgreSQL https://github.com/npgsql/npgsql/blob/main/LICENSE
Newtonsoft.Json: MIT https://github.com/JamesNK/Newtonsoft.Json/blob/master/LICENSE.md
Serilog: Apache 2.0 https://github.com/serilog/serilog/blob/dev/LICENSE
Bogus: MIT https://github.com/bchavez/Bogus/blob/master/LICENSE
Prometheus-net: MIT https://github.com/prometheus-net/prometheus-net/blob/master/LICENSE
Swashbuckle.AspNetCore: MIT https://github.com/domaindrivendev/Swashbuckle.AspNetCore/blob/master/LICENSE
coverlet: MIT https://github.com/coverlet-coverage/coverlet/blob/master/LICENSE
FluentAssertions: MIT https://github.com/fluentassertions/fluentassertions.json/blob/master/LICENSE
xunit: Apache 2.0 https://github.com/xunit/xunit/blob/main/LICENSE
Playwright: Apache 2.0 https://github.com/microsoft/playwright/blob/main/LICENSE
NUnit: MIT https://github.com/nunit/nunit/blob/master/LICENSE.txt