DevOps - Group M

VORES NAVNE

May 2023

1 System Perspective

1.1 Design and architecture

The technologies we selected to recreate the legacy flask application ITU-MiniTwit, were C# with the ASP.NET Core framework for the server and Postgresql as the database. The decision of using ASP.NET Core was made based on that the team had prior experience using it, which enabled us to quickly implement the features from the Flask application. Another reason for our selection is that with the performance enchancements of .NET 7, this is a very well performing framework[1]. This can also be seen in our groups solution being one of the top performing in this chart: `http://104.248.134.203/chart.svg`.

We used Razor pages to render the web pages on the server side. We also considered creating a single page application as a frontend, such as with Blazor or React, as this would be the more modern approach. However we stuck to using Razor pages as we wanted to replicate the original Flask application as much as possible.

skriv om simulator api'en og lav diagram over appen

Figure 1 shows the architecture of the final version of the system that was deployed, using three droplets (virtual machine) on DigitalOcean. Docker containers are within a docker swarm and communicate via the internal overlay network. There are three replicas of the MiniTwit server in the swarm, one on each droplet, to ensure availability. The two worker nodes only contain an instance of the MiniTwit server, the manager contains the services for load balancing, logging and monitoring. We decided to keep all these services on the same node so they are accessible on the same IP address. Nginx is used for load balancing between the MiniTwit instances. The database was hosted as a DigitalOcean managed database so that it was guaranteed to be reliable. Initially, we managed the postgres database ourselves, however one day it suddenly crashed causing approximately 12 hours of downtime. To avoid worrying about this we decided to switch to a managed database in our deployment. Running our *infrastructure.sh* script, which deploys the entire system and provisions the infrastructure on DigitalOcean, will make the database as a container in the swarm though. Rolling updates are implemented through docker swarm.

[1] 14th in this benchmark: `https://www.techempower.com/benchmarks/#section=data-r21`

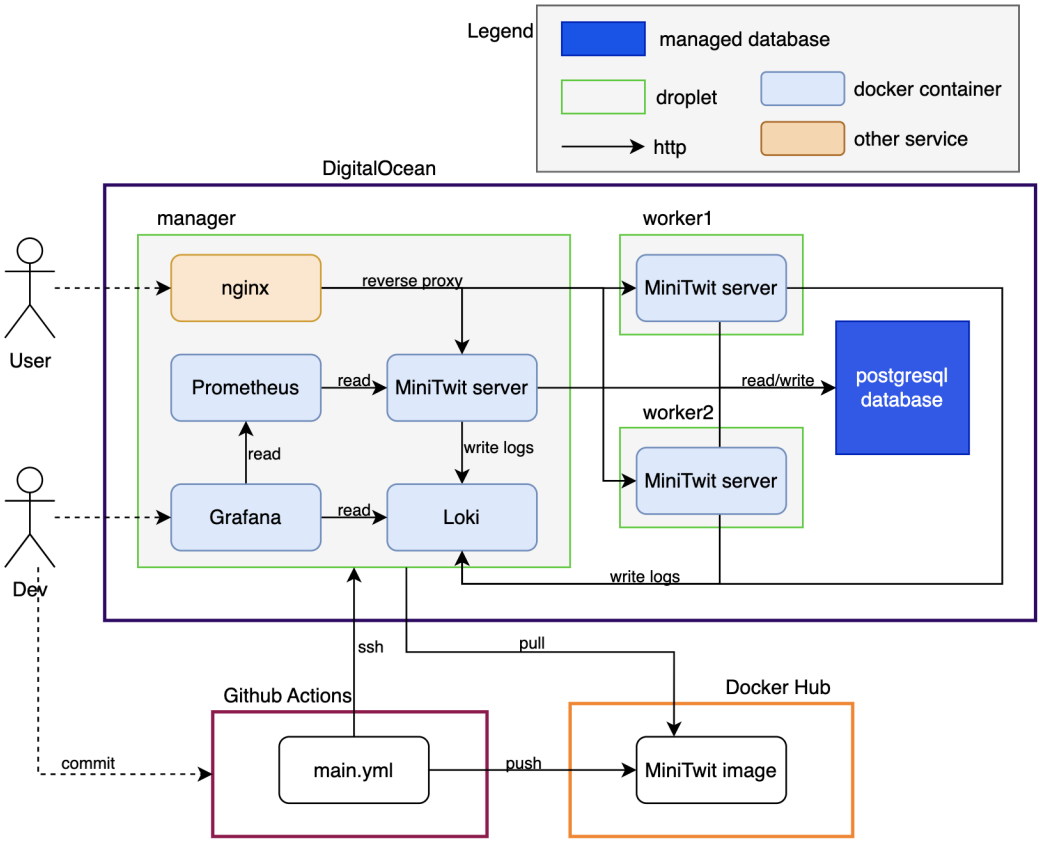


Figure 1: Architecture of the final system.

1.2 Technologies and Dependencies

The technologies we have decided on for the system to rely on are listed below. All versions used were the most recent stable releases as of start-2023:

- **ASP.NET Core**, for the web application.
- **PostgreSQL**, for data storage.
- **nginx**, for reverse proxy and load balancing.
- **Grafana**, for view metrics and logs.
- **Loki**, for storing logs.
- **Prometheus**, for storing metrics.
- **Docker**, for containerization.
- **Docker Hub**, for container image registry.
- **Ubuntu**, for the server OS.
- **Python**, for the tests.
- **Nuget**, for package management.
- **Github Actions**, for CI/CD.
- **DigitalOcean**, for infrastructure.
- **SonarCloud**, for static analysis.

The libraries which the ASP.NET Core MiniTwit application depends on are:

- **Entity Framework Core**, library for database abstraction layer.
- **Npgsql**, library for connecting to postgres from .NET.
- **Serilog**, library for logging.
- **Serilog.Sinks.Grafana.Loki**, library for shipping logs to Loki.
- **Prometheus-net**, library to instrument .NET code with Prometheus metrics.
- **SignalR**, library for realtime communication between client and server.

1.3 Important interactions of subsystems

Figure 1 shows the interactions of the subsystems. Docker containers communicate via the internal network within the swarm.

A shortcoming of the architecture have chosen is that Prometheus only reads from one of the MiniTwit server instances, the one on the same node. Since the load is balanced this means that only 1/3 of the traffic is monitored by metrics. This is not optimal, since it does not give all of the information, however it does present a sort of snippet of the systems' monitoring.

Er der mere til interactions mellem subsystems?

1.4 Current state of systems

Due to our project utilizing infrastructure as code, it is possible to easily and quickly turn the system on/off. At the time of writing this report, we have turned the system off to minimize payments to Digital Ocean. When turning our system on, everything works as expected. However, there is little to no traffic, since the simulator has stopped.

The current code quality can be seen on our SonarQube page, which is showcased in figure 2.

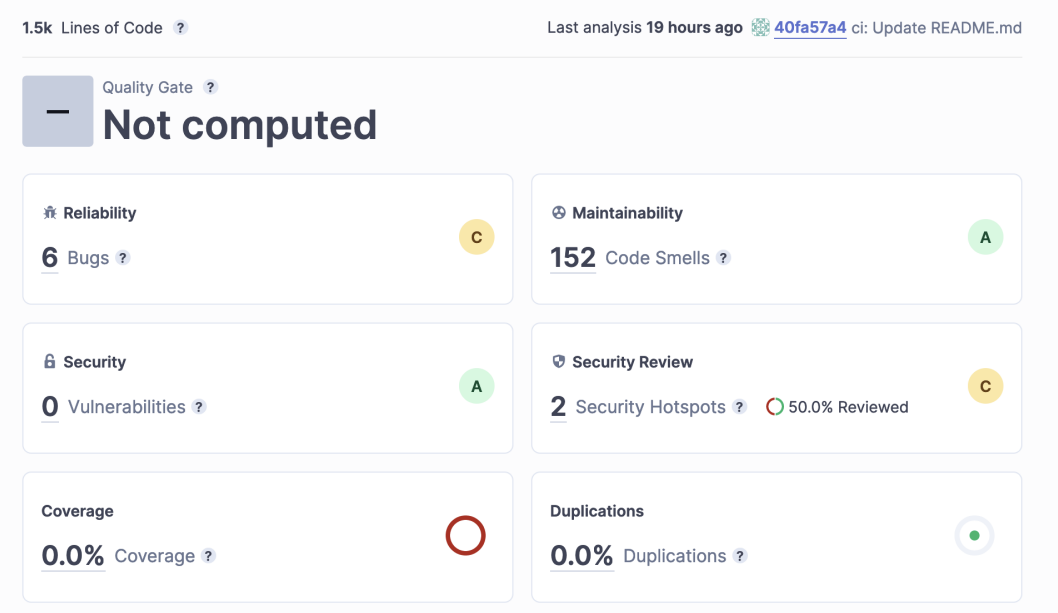


Figure 2: An overview of the code quality of the project in SonarQube.

1.5 License of our systems

As we use the MIT license, and most of the direct dependencies use AGPLV3, we do not have any license conflicts.

2 Process Perspective

2.1 Teamwork

2.1.1 Team agreement

Within the period where the project was worked, the team formed some rules and guidelines based on some of the points presented in the "Three Ways" to characterize DevOps from "The DevOps Handbook": Flow, Feedback and Continual Learning/Experimentation. This was to enhance efficiency, and to ensure transparency within the group.

Lets get a citation here

To ensure flow we did the following:

- Made work visible by using the GitHub kanban board.
- Reduced batch sizes by focusing on getting on feature done before moving focus to another.
- Reduced the number of handoffs by only working on a few features at a time,
- Continually identified and evaluated constraints by using GitHub issues if problems occurs.
- Eliminated hardship and waste in the value stream by implementing automatic releases in our pipeline.

To ensure feedback we did the following:

- Discovered problems as they occur and make sure to communicate them to the team.
- Swarmed and solved problems to build new knowledge and share that knowledge with the rest of the group.
- Kept pushing quality closer to the source by using a trunk-based strategy that tested each push to the main branch.

To ensure continual learning and experimentation we did the following:

- Institutionalized the improvement of daily work by keeping an open discussion about what we can do better.
- Injected resilience patterns into our daily work by using GitHub actions to constantly ensure quality of the work done.
- Ensuring a learning culture by by sharing knowledge at weekly meetings.

2.1.2 Working schedule

The team's interaction has been physical at ITU and online through Discord. The primary working day was Tuesday after the exercise session. The group would try to solve the exercise session first together, and then afterwards implement the feature to our own system. If there happened to be leftover work, we would split into subgroups, as different schedules made it difficult to meet physically more than once a week. Since most of the features and tools were new to the group the overall focus

was to implement and use them in a correct manner. This led us to use pair programming, where one team member would share his screen and other team members provided input. In case the workload was trivial, the other team members would also write code individually. Furthermore, the team members who were not sharing their screen, would under many circumstances try to test or solve things on their own machines. This was to speed up the process, especially when we were working with something that we did not initially understand.

We incorporated retrospectives from Scrum every Tuesday before we began working on next weeks tasks. This meant that we discussed what had been implemented since last Tuesday as a way of sharing our individual knowledge with the group.

2.2 CI/CD

We used Github Actions to implement our CI/CD. Our CI/CD consists of test, deploy, release, and code quality analyzer. We used Trunk Based Development where every developer shared the same trunk branch. Every developer pushed to the main branch for efficiency. Upon push Github Actions will run the following steps:

2.2.1 Code quality analyzer

SonarCloud was used for code quality analyzer to analyze the technical debt of the code base. Our repository contains badges with information about the technical debt, security, vulnerabilities, and bugs based on the latest commit to SonarCloud. We also used the C# linting tool *code-cracker*, but this was only used when building on developer machines and not a part of the workflow. Furthermore, we have used Snyk for scanning the project, which er not a part of the workflow either.

2.2.2 Test

We have different tests as a quality gate before deploying and releasing. First, we have smoke test to verify that the most important elements of the application is functioning. Second, we use Selenium for end-to-end tests. Last, we use integration test in order to test the API.

2.2.3 Deployment

If the tests are successful the push will be deployed. We have omitted code reviews and branching in order to push new code faster. This is to prevent potential dead local code that is not being used, since we value that our code should be used, once it works.

2.2.4 Release

After deploying the application we use Versionize and Husky for automatic releases on Github.

2.3 Repository organization

We use a monorepository for our source code, both frontend and backend. As mentioned we have a trunk based development, which means that developers push directly to the main branch. If the commits passes the Github Workflow, then it will be released. We omitted code reviews because we thought it took too much time to do them, and since we often worked together physically, we mostly already knew what each other had written. Initially we considered using it for knowledge sharing, but since we were already set on using pair programming, we already had a way of knowledge sharing. Since we did not have code reviews we wanted to be able to release code as fast as possible.

2.4 Development Process

To track issues and which tasks needed to be done, we have to some degree utilized Github Projects, which provides Kanban boards. We have not used this feature rigorously, since the course structure already provided some structure of what needed to be done when. The main usage has been when we have split our group up in many parts (individual work), and needed to keep track of it. Mostly development has happened in group work, either all together or split in two groups with knowledge sharing. Since our project and group size are manageable, it is still possible to hold developers accountable by looking at commit history. A more rigorous use of the Kanban board would however have made it possible to retroactively measure system progress.

2.5 Monitoring & Logging

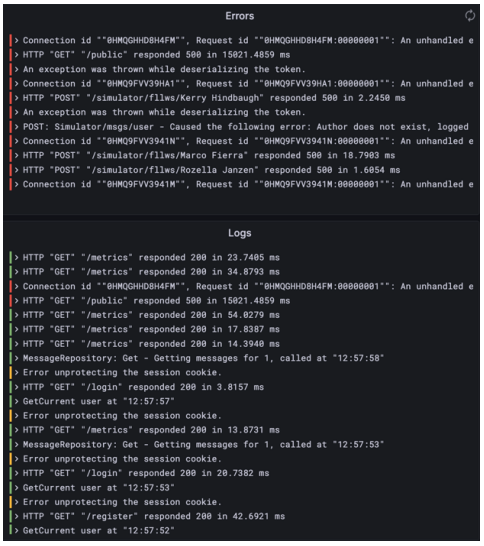
During the lifetime of a system, it is rare for everything to work smoothly and without any problems. System components will break down, the system will crash in production, server response times will grind to a halt, and so on. In order to (1) be able to estimate when the system is not operating within the expected limits, and (2) be able to track down the source of this disturbance, it is necessary to have monitoring and logging setup in the project.

Our monitoring is done via Grafana dashboards, where we have created two dashboards: One for system metrics (RAM, CPU usage, response times) and one for business logic (active users, timeline requests, number of tweets made). When the first dashboard showed irregular behaviour, it was a sign that our virtual machine in the cloud was in need of scaling - or that poorly performing code had been pushed

recently. When the second dashboard showed irregular behaviour, it was a sign that our website was not deployed properly and inaccessible to users, or that some recently pushed code had broken core functionality.

In order to accurately track how many requests our system has in real time, we have implemented a logging tool. Each time a new user is created, an existing user logs in, a user makes a post, or a user follows another user, the API call is registered and saved in the logs. This gives us the ability to determine whether or not something is wrong, in case these API calls fail.

(a) A snippet of our logs showing the API calls registered



(b) A snippet of our logs when we took down the database

Figure 3: 2 Figures side by side

When we had to change the database, we first had to take it down. This is visible in our logs, since we began getting a lot of errors when this was the case. This was ultimately a good sign, since it meant that if something was wrong in our system, we would be able to see it immediately, and therefore fix it as fast as possible.

2.6 Scaling And Load Balancing

Our load balancing is separated for a 'normal' user and the course simulator. When a request is made through the simulator API, we have decided to use a round robin approach to load balancing, which essentially means that the request is just sent to some container. However, when a connection is established through the website, we are using an IP hash approach. This means that the same user is always directed to the same container, which ensures that we do not have to worry about the user changing session states when the connection is renewed.

It has not been necessary to establish a scaling strategy, due to the scope of the course. Our monitoring has never indicated this to be necessary. If our cur-

rent system would be overloaded, it would be necessary to either scale vertically or horizontally. Our system makes it easy to scale horizontally, the only real change being having to change our infrastructure as code scripts. Scaling vertically is easy through Digital Ocean.

2.7 Security Assessment

To discover potential security flaws in our system implementation, we decided to do a security assessment. This was done through the following process:

1. Identifying the different components of the system.
2. Identify which component posed potential security breaches, and what these were for each component.
3. Constructing risk scenarios on each potential security breach.
4. Performing a risk analysis on the different scenarios.
5. Running a preemptive pen-test on our own system.

	Insignificant	Negligible	Moderate	Extensive	Significant
Almost certain	Scenario E	Scenario I			
Likely				Scenario B	
Possible		Scenario C			
Unlikely				Scenario A	
Rare			Scenario G	Scenario F	Scenarios D, H

Figure 4: Security assessment of different risk scenarios (see appendix 4.1 for details on each scenario).

A report of our entire process can be found in appendix 4.1. The security assessment, which this table stems from, can be seen in figure 4. The table shows us that the most critical scenario is (B) Attacker performs a DDOS attack to make our server unresponsive. This would make our service useless, and could be very bad for the business. There is however no imminent way to mitigate this, other than scaling the application, having a load balancer, or pay for DDOS protection. Our system has a load balancer, but the other options were deemed to be outside the scope of this project.

While the security assessment has helped us realize which parts of the system is most vulnerable, and how they might be attacked, it does in no way guarantee that our system is unexploitable, or that every possible scenario is considered. The most normal way to discover security flaws is when an attacker takes advantage of them. The security assessment did however give us confidence that the most normal tools for discovering weaknesses didn't find anything on our system. Based on our

findings, it was not necessary to change any parts of the system.

2.8 AI assistants

In this project we used ChatGPT. It did not write any of our source code, but it turned out to be helpful for error handling. When googling an error, it can sometimes be difficult to find the correct stack overflow response, whereas when using ChatGPT, we could paste the error message and often it could point us in the right direction. It has also been useful for setting up nginx and docker swarm. For instance, ChatGPT provided help for how to store session ids from users with ID hash.

We tried not to rely on it too much in order to keep the work as authentic as possible. While getting help from it can be useful, we never directly copy/pasted anything that it gave us. The way we used ChatGPT was essentially as an extra TA, for when TAs were not available.

3 Lessons Learned Perspective

3.1 Evolution and refactoring

Initially, we wanted to use Blazor for developing the application. However, because of the limited time for refactoring we switched to Razor pages, as we estimated that it required a third of the code. Furthermore, the idea of choosing Razor pages instead of Blazor, was because the code base would be a lot more similar to the project we had to refactor. Blazor is notoriously known for splitting up model, view and controller, which is not the case with Razor.

3.2 Operations

We learnt the importance of knowledge sharing. We quickly found that when we did not share what we had done to the project since last week, it was difficult for all members to have an overview of the entire system. To solve this issue we started using retrospectives as a way of knowledge sharing every Tuesday before next weeks task.

3.3 Biggest issues

One of the biggest issues the group faced, was the limited work time that we had. Since the project was rather extensive, and the amount of new tools that should be incorporated was large, it required a lot of time of patience from the group members. This was an issue, since we primarily only had Tuesday as the day where we met physically, and otherwise only online days. We all agreed that we work better when we are in the room together, and this was difficult to do every time. The result of this issue was that we sometimes had to work until the late evening hours on Tuesdays, but it was a price that we were willing to pay, so that we did not have to work more online than what we already did.

3.4 Major lessons learned

1. We learnt a lot of new tools such as Grafana, Prometheus, Vagrant, Digital Ocean, among others. We also reinforced our learning, and learnt more about tools such as Bash, Docker, and Github.
2. We learnt how to use the DevOps theory in practice, on a project, which helps a lot with understanding the concepts more thoroughly.
3. We learnt what it felt like to work with a DevOps approach, which many companies use in real life.
4. We learnt and understood how one should think when working with DevOps, since this is a very different way of thinking about the project and the code, compared to not using a DevOps approach.

5. The value of automating your work: Being able to turn the entire application on/off with a single script. While the initial work needed to write the automation scripts was demanding, it proved a very valuable and cool feature of our project. Being able to automate the many, complicated steps in deploying a new version of the code to a live server is also a completely new experience.
6. The practical sides of writing code meant to be published. It is not enough for a system to run on just our own computers. Normally we just hand in the source code, but in this project we had to make sure our code was deployable in the real world.

3.5 The differences with DevOps

Many of the previous projects we have worked with on ITU have not used a DevOps approach. This resulted in many of the things we did, technologies that we used, and the way we had to think about the project changed substantially. All of the group members were writing their bachelor thesis, while having this course, and none of us are using DevOps in that. This gives a very clear comparison between the applied work ethics in the two respective courses.

Normally, we do not prioritize committing and pushing code as soon as it works in small steps. Instead, we usually only push when we have to merge the code we have written, with what the others have written. This is in sharp contrast to the DevOps approach we have had in this course, where we would always commit, push, (and, when the pipeline was set up correctly, release), when we had something that worked. The DevOps approach led to fewer merge conflicts, compared to our previous projects, and it also resulted in our code being live faster. If one has code locally on their device without it being released, it is more or less dead code, since it is not being used whatsoever.

4 Appendix

4.1 Security Assessment

A. Risk Identification Identify assets (e.g. web application)

1. Webserver (deployed via DigitalOcean)
2. Prometheus
3. Managed Database
4. Grafana service
5. Loki

Identify threat sources (e.g. SQL injection)

1. Webserver (deployed via DigitalOcean)
   (a) Unsanitized input (sql injections)
   (b) DDOS
   (c) Brute force attacks on login
   (d) SSH brute force attack on login
2. Prometheus
   (a) Publicly exposed metrics
3. Managed Database
   (a) No/bad authorization/authentication
   (b) Entire database could be exposed/deleted
4. Grafana service
   (a) No/bad authorization/authentication
   (b) Public dashboards
   (c) Editable dashboards (not read-only)
5. Loki
   (a) Can get information about username of users

Construct risk scenarios (e.g. Attacker performs SQL injection on web application to download sensitive user data)

A Attacker performs SQL injection on web application to download sensitive user data

B Attacker performs a DDOS attack to make our server unresponsive

C Attacker brute forces a user's password and gains control over their profile

D Attacker brute forces SSH credentials and gets full access over the web server

E Attacker reads the public Prometheus metrics, and gains business insights on our service

F Attacker gets control over Prometheus and is able to misrepresent metrics. This could mask spikes or irregular patterns in our monitoring.

G Attacker is able to get access to our Grafana service, and can see/delete all our monitoring

H Attacker is able to get access to our managed database, and downloads/deletes

all our data

I Attacker gets access to metrics

B. Risk Analysis Translated from english to danish: Vurderinger er lavet baseret på sandsynligheden for, at et angreb ville lykkedes Certain, likely, possible, unlikely, rare Insignificant, Negligible, Marginal, Critical, Catastrophic Determine likelihood and impact

A Likelihood: Unlikely, Severity: Extensive

B Likelihood: Likely, Severity: Extensive

C Likelihood: Possible, Severity: Negligible

D Likelihood: Rare, Severity: Significant

E Likelihood: Possible, Severity: Insignificant (ITU IP is whitelisted)

F Likelihood: Rare, Severity: Extensive

G Likelihood: Rare, Severity: Moderate

H Likelihood: Rare, Severity: Significant

I Likelihood: Almost Certain, Severity: Negligible

Use a Risk Matrix to prioritize risk of scenarios in current state of program Discuss what are you going to do about each of the scenarios

Ud fra vores matrix kan vi se at de vigtigste scenarier at gøre noget ved er B

For at fikse B kan vi: Load balancing

4.2 Logs

4.2.1 Session 2

- We have decided to write in C# and dotnet since that is the language most of us are comfortable with.
- The folder structure of the legacy project is not that good. We therefore set up a new folder structure following the MVC pattern.
- We struggled a lot and needed to make a lot of changes to make the program work.

4.2.2 Session 3

- We are not taking small steps. We took a big step in trying to implement it in a different way. Lukas has made it just using razor pages instead. This makes an exact copy of the project. This is the correct way to do it. Therefore we will build our main branch based on that.
- We have made the Simulator and tested it. It should work. For the "latest" endpoint we have created a static variable that can be used. Maybe this is a bit scuffed since it can lead to race conditions if multiple people try to access the resource. But let's see.

- We have changed the database to a postgres database so it should be better when a lot of users start using the program. The database also runs in another docker image now.
- We have tested the simulator with the test program provided in the project work, and the program seems to run as it should.

4.2.3 Session 5

Consider how much you as a group adhere to the "Three Ways" characterizing DevOps (from "The DevOps Handbook"):