DevOps - Group M

Casper Wassar Skourup - caws@itu.dk
Lukas Hjelmstrand - luhj@itu.dk
Mads Piriwe Risom Andersen - mpia@itu.dk
Niclas Søgaard Hjortkjær - nihj@itu.dk
Villum Nils Robert Sonne - vson@itu.dk

24/05-2023

DevOps, Software Evolution and Software Maintenance, BSc
Course code: BSDSESM1KU

Github link: https://github.com/NiclasHjortkjaer/itu-minitwit
Webite link: http://jasonderulo.live
Simulator link: http://jasonderulo.live/simulator

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 to use ASP.NET Core was made based on the team's prior experience, which enabled us to implement the legacy Flask application's features quickly. Another reason for our selection is that with the performance enhancements of .NET 7, this is a very well performing framework[1]. This can also be seen in our group's solution being one of the tops 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 with using Razor pages to replicate the original Flask application as much as possible. Further explanation is given in section 3.1.

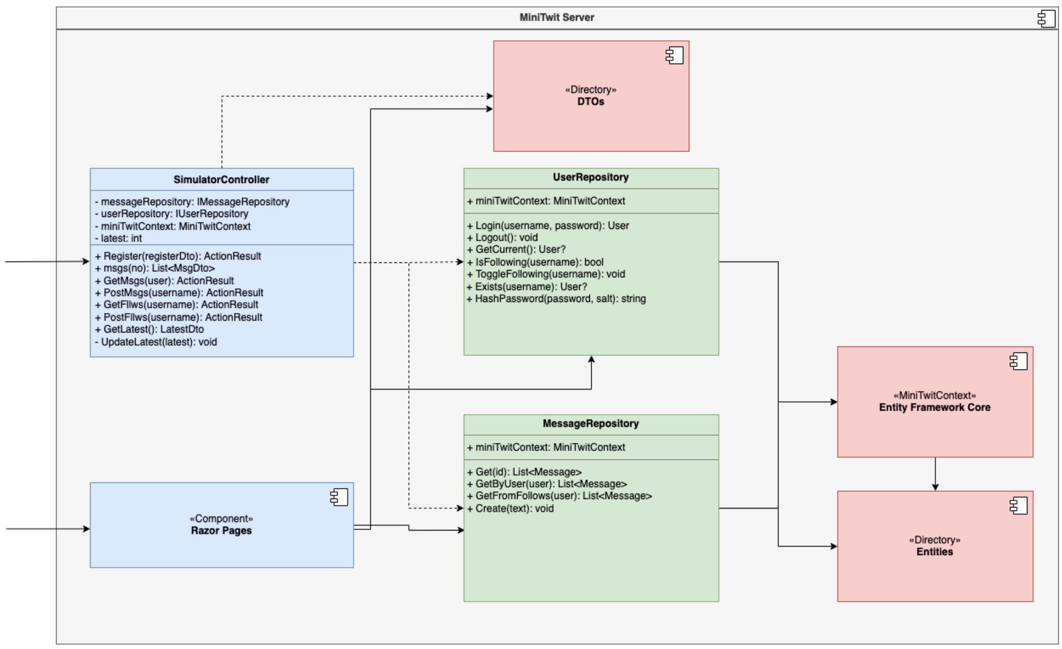


Figure 1: Architecture of the MiniTwitApi component

For the illustration of the activity on our Minitwit website, we created an API that the simulator could communicate with. Over four weeks, the simulator provided our system with over 4 million messages/posts and nearly 13 million API calls.

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

Figure 2 shows the architecture of the final version of the system that was deployed, using three droplets (virtual machines) 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, while the manager includes load balancing, logging, and monitoring services. We decided to keep all these services on the same node to be 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, which caused approximately 12 hours of downtime. We decided to switch to a managed database in our deployment to avoid worrying about this. Running our *infrastructure.sh* script, which deploys the entire system and provisions the infrastructure on DigitalOcean, will make the database a container in the swarm. Rolling updates are implemented through the docker swarm.

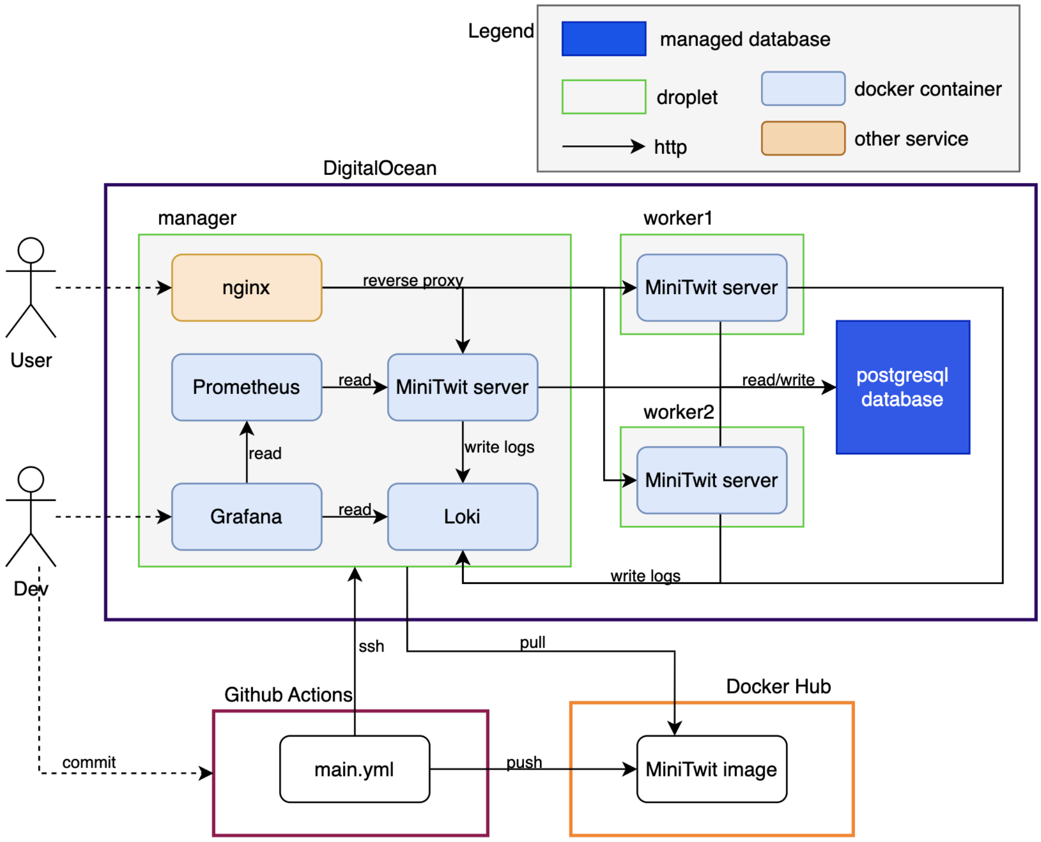


Figure 2: Architecture of all subsystems in the final system.

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 the start of 2023:

- **ASP.NET Core**, for the web application.
- **PostgreSQL**, for data storage.
- **nginx**, for reverse proxy and load balancing.
- **Grafana**, for viewing 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.
- **Snyk**, for security assessments.

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 real-time communication between server and client.
- **code-cracker**, library for analyzing code.

1.3 Important interactions of subsystems

Figure 2 shows the interactions of the subsystems. Docker containers communicate via the internal network within the swarm.

When a developer pushes to the main branch on GitHub, the *main.yml* workflow is run. First, the system is built on the workflow server via Docker-Compose, then the Python tests interact with the system via HTTP. We have refactored and expanded the Python tests from the exercises. If the tests all pass, the workflow pushes the new MiniTwit image to Docker Hub. It then utilizes ssh to execute a script on the manager node, which pulls the image and updates the workers. Finally, a GitHub release is created by the workflow.

A shortcoming of our chosen architecture is that Prometheus only reads from one of the MiniTwit server instances, the one on the same node. Since the load is balanced, only 33% of the traffic is monitored by metrics. This is not optimal since it does not give all of the information. However, it presents a snippet of the system's monitoring. The same problem appeared with the real-time updates implemented with SignalR. Because of the load balancing, only every third message will, on average, appear for the user without reloading the page.

1.4 Current state of systems

Due to our project utilizing infrastructure as code, it is possible to turn the system on/off quickly. When 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 3.

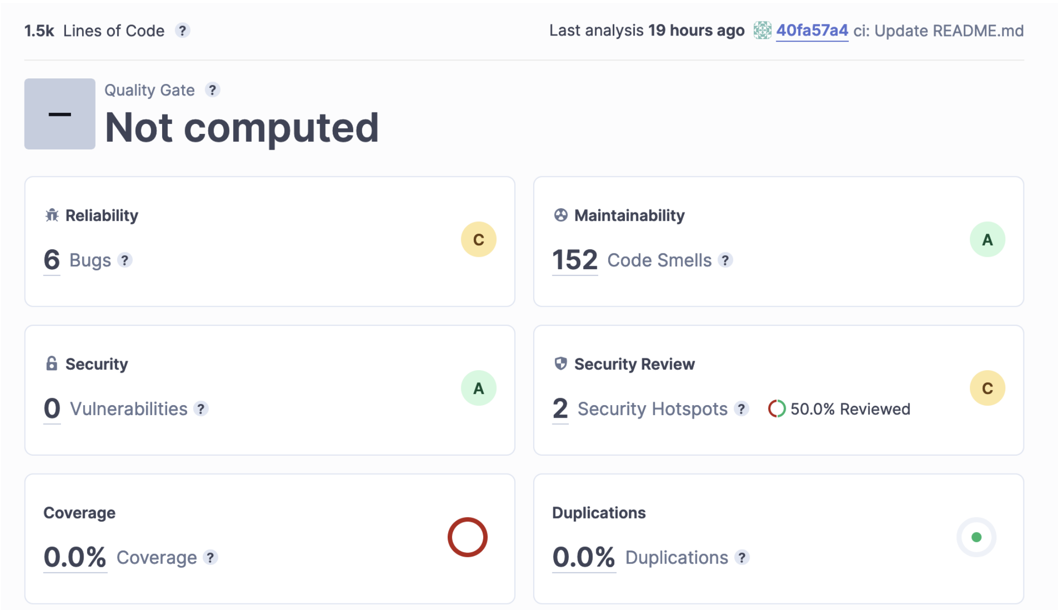


Figure 3: An overview of the code quality of the project in SonarQube.

For future work on the project, it would make sense to perfect the load-balancing aspect of the dockerized swarm. For example, the current setup means that Grafana only shows 1/3 of metrics, and when a user is connected to a MiniTwit server via WebSockets, this user would only receive 1/3 of new tweets. Currently, the WebSockets only send messages to clients when a change happens on the server instance, not when a change occurs in the database.

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. Our API also has an SLA.

2 Process Perspective

2.1 Teamwork

2.1.1 Team agreement

Within the period where the project was worked on, 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"[2]: Flow, Feedback and Continual Learning/Experimentation . This was to enhance efficiency and ensure transparency within the group.

To ensure flow, we did the following:

- Made work visible by using the GitHub Kanban board.
- Reduced batch sizes by focusing on getting one 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 using GitHub issues if problems occurred.
- 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 occurred and communicated 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 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.
- Ensuring a learning culture 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. First, we solved the exercises together and then implemented the feature in our system. If there were 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 on implementing and correctly using them. This led us to use pair programming, where one team member shared his

[2] https://www.oreilly.com/library/view/the-devops-handbook/9781457191381/

screen, and others provided input. The other team members would write code individually if the work were trivial. Furthermore, the team members who were not sharing their screens would, under any circumstances, try to test or solve things on their machines. This was to speed up the process, especially when working with something we did not initially understand.

We incorporated retrospectives from Scrum every Tuesday before we began working on the following week's tasks. This means that we discussed what had been implemented since last Tuesday to share 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 as a code quality analyzer to analyze the technical debt of the code base. Our repository contains badges about 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 to scan the project, which is not part of the workflow.

2.2.2 Test

We have different tests as a quality gate before deploying and releasing. First, we have a smoke test to verify that essential elements of the application are functional. Second, we use integration tests to test the API. Last, we use Selenium for end-to-end tests.

2.2.3 Deployment

If the tests are successful, the push will be deployed. We have omitted code reviews and branches 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 when 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 mono repository for our source code, both frontend, and backend. As mentioned, we have a trunk-based development, meaning developers push directly to the main branch. If the commits pass the GitHub Workflow, then they 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. However, since we used 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, which is why we used trunk-based development.

2.4 Development Process

To track issues and which tasks need 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 and when. The primary usage has been when we split our group into many parts (individual work) and needed to keep track of it. Mostly development has happened in group work, either all together or split into two groups. Since our project and group size are manageable, it is still possible to hold developers accountable by reviewing the commit history. However, a more rigorous use of the Kanban board would have made it possible to measure implementation progress retroactively.

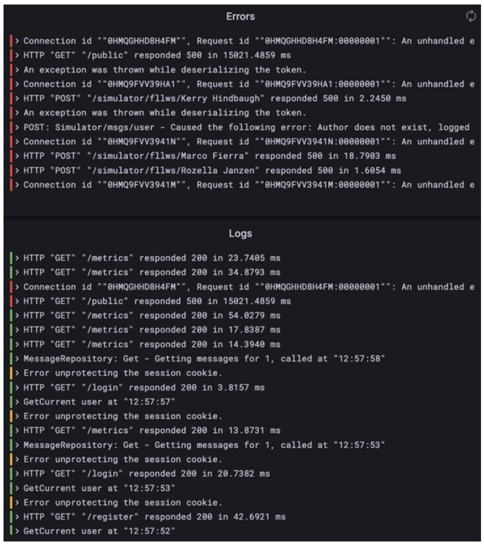
2.5 Monitoring & Logging

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. 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 behavior, it was a sign that our virtual machine in the cloud required scaling - or that poorly performing code had been pushed recently. When the second dashboard showed irregular behavior, it was a sign that our website was not deployed properly and was inaccessible to users or that some recently pushed code had broken core functionality.

We have implemented a logging tool to track how many requests our system has in real-time. 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 allows us 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 4: 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 were wrong in our system, we would be able to see it immediately and fix it as fast as possible.

Whenever we had downtime, new users could not be registered. However, the logs helped us with this since we had a script that could create the users from the logs directly, such that when the system was up and running again, the users who tried to register during downtown would now be registered in the database.

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 means 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 current system is overloaded, it would be required to 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

We decided to do a security assessment to discover potential security flaws in our system implementation. 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 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 5: 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 matrix can be seen in figure 5. The table shows us that the most critical scenario is (B) attacker performing a DDOS attack to make our server unresponsive. This would make our service useless and could be very bad for business. However, there is no imminent way to mitigate this other than scaling the application, having a load balancer, or paying for DDOS protection. Our system has a load balancer, but the other options were deemed to be outside this project's scope.

While the security assessment has helped us realize which parts of the system are most vulnerable and how they might be attacked, it does not guarantee that our system is not exploitable or that every possible scenario is considered. The most typical way to discover security flaws is when an attacker takes advantage of them. The security assessment did, however, give us confidence that the most ordinary tools for discovering weaknesses didn't find anything on our system. Based on our findings, changing any parts of the system was unnecessary.

2.8 AI assistants

In this project, we used ChatGPT. It did not write any of our source code, but it was helpful for error handling. When googling an error, finding the correct stack overflow response can sometimes be difficult, whereas, with ChatGPT, we could paste the error message, and it could often point us in the right direction. It has also been useful for setting up Nginx and docker swarm. For instance, ChatGPT provided help on how to use IP hash.

We tried not to rely on it too much to keep the work as authentic as possible. While getting help from it can be helpful, we never copy/pasted anything it gave us. Instead, we used ChatGPT as an extra TA.

3 Lessons Learned Perspective

3.1 Evolution and refactoring

As mentioned, we initially wanted to use Blazor to develop 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 code base would be much more similar to the project we had to refactor. Blazor is notoriously known for splitting up the model, view, and controller, which is not the case with Razor.

3.2 Operations

We discovered 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 took time for all members to have an overview of the entire system. To solve this issue, we started using retrospectives as a way of sharing knowledge every Tuesday before the next week's task.

3.3 Limited work time

One of the most significant issues the group faced was our limited work time. Since the project was rather extensive, and the amount of new tools that should be incorporated was large, it required much time from the group members. This was an issue since we primarily only had Tuesday as the day when we met physically. We all agreed that we work better when in the room together, and this was not easy to do every time. The result of this issue was that we sometimes had to work until the late evening hours on Tuesdays and work only on essential parts of the application. Still, it was a price that we were willing to pay not to have to work more online than we already did.

3.4 Major lessons learned

1. We learned a lot of new tools such as Grafana, Prometheus, Vagrant, and Digital Ocean, among others. We also reinforced our learning and learned more about tools like Bash, Docker, and Github.
2. We learned how to use the DevOps theory in practice on a project, which helps a lot with understanding the concepts more thoroughly.
3. We learned what it felt like to work with a DevOps approach, which many companies use in real life.
4. We learned 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. We learned about the value of automating work: E.g., turning the entire application on/off with a single script.
6. We learned about the practical sides of writing code meant to be published.

3.5 The differences with DevOps

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

Usually, 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 our code with what the others have written. This is in sharp contrast to the DevOps approach we 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 than in our previous projects, resulting in our code being live faster. If one has code locally on their device without it being released, it is considered dead code since it is not used.

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"):