DevOps 2022 - Group E Final Report

jubr@itu.dk, jsow@itu.dk, npal@itu.dk, oeje@itu.dk, osha@itu.dk

June 2022

Contents

1 Introduction **4**

2 Process **4**
2.1 Team work 4
2.2 Version Control 4
2.3 Work practices 4
2.3.1 Work division 4
2.3.2 Branch strategy 5

3 Technologies **5**
3.1 Application Framework 5
3.2 Object-Relational Mapping Framework 6
3.3 Database Management System 6
3.3.1 Hosting on Digital Ocean 6
3.4 Provisioning Tool 6
3.5 CI/CD Hub 7
3.5.1 Automatic releases 7
3.6 Cloud Provider 7
3.7 Load Balancing and Scaling 8

4 Software Maintenance **8**
4.1 Continuous Integration and Continuous Deployment 8
4.1.1 SSH into the deployer 8
4.1.2 Load environment variables 9
4.1.3 Clone repository onto deployer-machine 9
4.1.4 Create environment for web server 9
4.1.5 Run `vagrant up` on deployer 9
4.1.6 Run scripts to replace the old production server 9
4.1.7 New version is deployed 9
4.2 Monitoring 10
4.2.1 Stack 10
4.2.2 Provisioning 10
4.2.3 Visualization 10
4.3 Logging 11
4.3.1 Setup 11
4.3.2 Idea 11
4.3.3 Result 11
4.4 Risk Analysis 11
4.4.1 Risk Matrix 11
4.4.2 Solutions/mitigations 12

5 Stack Diagram **13**

6 Status on Static Analysis **13**
6.1 Code coverage 14
6.2 License 14

7 Perspective **14**
7.1 Structuring of infrastructure 14
7.2 Concurrent deployment caused outage 15
7.3 Handling secrets 15
7.4 Quality of tools 15

8 Bibliography **16**

9 Appendix **16**
9.1 Appendix A - Choice of the main language (C#) 16
9.1.1 Our measures 16
9.1.2 Python 17
9.1.3 Functional Languages 17
9.1.4 C# 17
9.2 Appendix B - SQL Vs NoSQL 17
9.2.1 SQL vs NoSQL 17
9.3 Appendix C - Scripting languages 18
9.3.1 Shell 18
9.3.2 Python 18
9.4 Appendix D - Security analysis 18
9.5 Risk Identification 18
9.5.1 Assets 18
9.5.2 Threat Sources 19
9.5.3 Risk Scenarios 19
9.6 Risk analysis 20
9.6.1 Determine likelihood 20
9.6.2 Determine impact 21

1 Introduction

This report describes the process of developing and maintaining an application "MiniTwit", which is a simple Twitter clone. The project was made as a part of the Spring 2022 course DevOps, Software Evolution and Software Maintenance. The finished application can be viewed on the website http://shittytwitter.ml, and the code for the application can be viewed in our GitHub repository: [1].

2 Process

2.1 Team work

When the project first started, we quickly decided on a set of rules to help us throughout the project. Such rules were for example: we would always create new branches for features, never push directly to the master branch on GitHub, and use Discord for both communication, as well as a place to share app secrets between group members. To discourage rule breaking, if someone were to break a rule, they would owe beers or something similar for a final celebration once the project was over.

The team would always meet up on campus on Tuesdays, which was the designated work day for the DevOps course. Here we would discuss the previous weeks work, what everyone was working on currently, and what people planned on doing until next week. We often ate lunch together on this day as well, and sometimes played ping pong after class to help us get to know each other, and to help us wind down after a day of work.

2.2 Version Control

Git was picked as the version control system for this project, and was used through GitHub. Everyone in the group was familiar with Git and GitHub already, so we could get quickly started with the project. By using GitHub, we were also able to use GitHub Actions in our workflow, which will be explained in further detail in section 3.5.

2.3 Work practices

The team has mainly been working in smaller groups of 1-2 people on each task throughout the project. As the group consisted of 5 people in total, it did not make sense to involve the entire group in each task, as it would simply not be efficient use of development time.

2.3.1 Work division

To keep track of tasks to be done, we used GitHub Projects, which is a Kanban board directly attached to GitHub. It allowed us to quickly create tasks from GitHub Issues, as well as assign developers and branches to said tasks. This was

under-utilized throughout the project, but was picked up again near the hand-in date, where it proved to be very helpful to create a final to-do list. Outside of the Kanban board, we also reminded each other of tasks to be done through other communication channels, such as Discord or in person.

2.3.2 Branch strategy

When a new task was being worked on, people involved were to create a new branch on the GitHub repository, with a fitting name of the feature, which was to be merged into a central development branch at a later point. Once the task was done, at least one person outside of the task had to review the pull request. Afterwards, the new feature was discussed in the group, either online or in person, ensuring everyone was caught up to speed. When the central development branch had had a reasonable amount of changes, it would be merged into the master branch, and then deployed automatically.

3 Technologies

This section will briefly describe the entire technology stack, and our reasoning for each technology, sometimes comparing them to competitors.

3.1 Application Framework

We chose ASP.Net as our framework for the web application for multiple reasons: Support, maturity and familiarity. ASP.Net also provides built-in support for many things that we will need, including but not limited to: session management, object-relational mapping, model-view-controller design pattern, security and server-side rendering.

Appendix A (9.1) is a more in-depth discussion of different languages and frameworks, and Appendix C (9.3) is a description of the scripting languages we used in CI/CD.

Support In 2021, the most popular non-JavaScript web framework was ASP.NET Core [6]). As ASP.NET Core is currently one of the largest web frameworks, there is a large amount of support available, as the community is very active and it is usually easier to troubleshoot problems, as someone likely has had the same issues before.

Maturity ASP.Net has been in development for 20 years, and has become solid software that will reliably fit into most tech stacks.

Familiarity A less technical reason for choosing ASP.Net is that the group was already familiar with it, which would likely reduce development time

3.2 Object-Relational Mapping Framework

To manage data between application and database, we use the Object-relational mapping library Entity Framework. This particular framework was chosen partly because we were familiar with it, but also because of the following points:

Entity Framework provides a lot of useful tools and features for web applications, such as:

lazy loading which improves performance, postponing the execution of a query until it is actually needed

query translation/inspection which allows for checking how the actual queries performed on the database look like, and enables us to tweak them to improve the performance

changes tracking which makes deploying the database schema changes easier, since it takes a single command to generate a migration based on the current state and a new database schema definition, and seldom have to make decisions on how something should be migrated.

3.3 Database Management System

Since we are only storing data about users and their tweets, a standard DBMS like PostgreSQL, MySQL or MS SQL Server is sufficient, with none having a major benefit over another. We ended up going with Postgres, because of our cloud provider and our familiarity with PSQL. During our consideration we compared the scalability of various solutions - even NoSQL options with their horizontal scaling - but we did not foresee that Postgres would cause us any issues.

3.3.1 Hosting on Digital Ocean

Our choice of DBMS has also been heavily affected by the fact that Digital Ocean (our chosen cloud provider) offers hosting a managed database, which automatically logs all database transactions and backs up our data independently of our application. While hosting the database and application separately increases response time per query, it simplifies difficult problems with data-migration during redeployment.

3.4 Provisioning Tool

For this project we have opted for using Vagrant as a virtual machine provisioning tool. It is *lightweight* in the sense that you only need a "vagrantfile" to provision a new cloud hosted virtual machine, and run an initialization script on it - essentially deploying the service from nothing by running a single file. This was initially a very attractive choice, as this file can simply be shared over GitHub, and any team member or deployment service would be able to provision

and start the application. While a Docker image of the web server might have taken more initial set-up of containerizing the application, had we been able, we would probably have switched to Docker, because of maturity issues with Vagrant. Vagrant is not as common as Docker, and thus not available out-of-the-box for services like GitHub Actions, and with limited support/issues with Digital Ocean.

3.5 CI/CD Hub

The need for a good CI/CD pipeline required us to find a tool, that could easily integrate into our workflow. The simplest solution, was to use the built-in solution provided by GitHub: GitHub Actions, which allows one to create pipelines by using *.yaml* files, and then to inspect their results on GitHub. One can argue that GitHub Actions is a poor choice, since we would rely even more on GitHub being online, and if GitHub goes down, we would lose access to all the pipelines. For that reason, we were considering using Travis CI, however we did not want to pay out of pocket for a tool for the project, so we settled on GitHub Actions, and accepted the risk of the service being offline. Even when using other CI/CD solutions, they would have to be integrated into our repository, in order to allow for functionalities like testing on pushes to a branch. GitHub would have to be online for that to work anyway.

3.5.1 Automatic releases

As part of the project we had to do regular releases of our code. GitHub Actions can automatically release when new code is pushed to master, which we have set up, such that the latest release always has the newest version without us having to write anything.

3.6 Cloud Provider

For this project, we picked Digital Ocean as the cloud provider to host both the application and other necessary infrastructure, as well as the PostgreSQL database.

In the group, we have members who have either used Amazon Web Services, Azure or Digital Ocean before. These three were therefore the main competitors in our optics.

Amazon Web Services and Azure provides a lot of different services that Digital Ocean does not. In return, Digital Ocean is (in our eyes) much more user friendly for non-experts, where the AWS dashboard and Azure portal can be a mouthful. Digital Ocean has an easy to understand interface with one-click deployments, as well as an intuitive API which can be invoked with a token.

In the end it came down to the fact that we each had $100 of free credit for Digital Ocean's services.

3.7 Load Balancing and Scaling

Towards the latter half of the project we began moving towards using Docker Swarm, rather than the singular Digital Ocean droplet.

There were a couple of reasons that made Docker Swarm an attractive approach for orchestrating the services in our system. The tool allowed us to change our deployment strategy from a blue-green strategy to a rolling update strategy instead, allowing the singular instances within the swarm to be updated one at a time, which should allow for faster updating once scaled horizontally. It also has a built in load balancer, allowing for much easier distribution between worker nodes. Because of these functionalities, Docker Swarm provides an approach that is more favorable if the application should be scaled to match a growing number of user requests.

One of the main reasons for why we choose Docker Swarm over other orchestration tools, is the relative ease of use and implementation into our existing setup.

4 Software Maintenance

4.1 Continuous Integration and Continuous Deployment

During the project, we set up a pipeline for continuous integration and deployment. The pipeline begins with someone creating a pull-request to the dev branch from a feature branch. The code from the feature branch is automatically built and tested with `dotnet test` when pushed, so we know it works and is ready to be merged. Once the pull request is reviewed and merged, the dev branch can be merged into master, which starts our deployment pipeline. The pipeline works in the following steps:

1. SSH into the deployer-machine.
2. Load environment variables into the deployer (tokens, ssh keys, connection strings, etc.).
3. Clone repository onto deployer.
4. Create environment for the web server.
5. Run `vagrant up` on deployer to provision the web server VM.
6. Run scripts to replace old production server with the new one, if the new deployment was successful.

4.1.1 SSH into the deployer

The deployer is a machine hosted separately from GitHub for two reasons. It decreases our dependence on GitHub, letting us deploy even if GitHub is down. It also gives more freedom to configure our deployer-machine, since GitHub

Actions does not have an out-of-the-box image with Vagrant on Linux, which we would then have to configure ourselves anyways.

4.1.2 Load environment variables

Since we do not want to store secrets in our public repository, we use GitHub Actions to store our secrets, and load them into environment variables during deployment. This also lets us switch the target machine-name that Vagrant uses by exporting a new environment variable, which simplifies swapping between the old production server and the new temp server.

4.1.3 Clone repository onto deployer-machine

The deployer clones the repo such that it can transfer the application to the web server.

4.1.4 Create environment for web server

The database and logging connection strings are loaded as environment variables. The deployer creates a script `env.sh`, which loads the connection string into the environment of the web server. The script is then used as part of the Vagrant provisioning.

4.1.5 Run `vagrant up` on deployer

The web server is provisioned and the program is started on port 80.

4.1.6 Run scripts to replace the old production server

Now that the server should be running, we need to make sure that it is actually alive, after which we can replace the old production server through a blue-green strategy. We have three scripts that run after each other:

1. `MinitwitAlive.py` repeatedly sends a request for the front page to the newly opened web server. If it responds with status code `200`, we assume that the server works.
2. `reassign_floating_ip.py` moves a static IP from the old production server to the newly provisioned server. If this script succeeds, the old production server is shut down, having seamlessly replaced the production server without downtime.
3. `rename_droplet.py` renames the new server from *temp* to *prod*.

4.1.7 New version is deployed

When all these steps have completed without errors, we have successfully deployed the newest version of the application. If any of these steps fail, the new deployment is rolled back with a failure on the GitHub Actions page.

4.2 Monitoring

4.2.1 Stack

To monitor the application, we used a basic Prometheus-Grafana setup consisting of:

- Calls to Prometheus metrics library within source code
- Prometheus instance that aggregates the metrics
- Grafana instance for displaying the data

4.2.2 Provisioning

Both Grafana and Prometheus are provisioned using configurations stored inside the "monitoring" folder in our repository.

Furthermore, we have dockerized both setups and created a Vagrant file, allowing us to start the monitoring service with a single command.

4.2.2.1 Problems

While provisioning of the Prometheus instance worked flawlessly, we encountered some issues with Grafana. Namely, the provisioning files for database as a data source were not working correctly, so we had to insert some details manually each time.

It was also very difficult to provision an automatic alert/notification system, so we decided to do that manually, since provisioning of this server happens infrequently.

4.2.3 Visualization

4.2.3.1 Idea

Grafana was used to help visualize metrics, to gain insight into the usage of the application.

Since everything in our app heavily relies on database queries, we decided to time each of them to see how fast they are.

To understand the average user better and to see how much data we actually have, we added database queries as another data source.

Finally, in order to see how well the system performs, and how many resources it uses, we added some metrics that display those parameters. In case the system went down, Grafana would alert us through a specified channel on Discord.

4.2.3.2 Results

Thanks to the visualization, we managed to fix some issues regarding database queries. We could also see usage patterns and act accordingly in case we would need to. At one point in the project, we also got a notification from the outage-alerter during development, which we quickly fixed.

4.3 Logging

4.3.1 Setup

Our logging setup consists of 4 components:

- Serilog - utility for Asp.NET for logging the events, with a target set as an Elasticsearch instance
- Elasticsearch - for searching and aggregating the logs
- Kibana - for visualizing the logs
- Nginx - reverse proxy for restricting access to other components

Much like in the monitoring case, the whole stack was dockerized, and a Vagrant file was created for deployment purposes. The machine used for serving the logs had to be a bit bigger in terms of RAM (4gb), otherwise Elasticsearch and Kibana would not work. It occurred to us later in the project, that the server's 80GB disk filled up rather quickly, as we had not set a "time to live" on the logs. Ideally, this would be set to two weeks or so. In the end, this meant that we were not able to index the logs, due to lack of space.

4.3.2 Idea

We wanted to have the ability to inspect queries that were being made to the database. This, with the addition of out-of-the-box logging allowed us to have a full overview of what the application was doing.

4.3.3 Result

As part of project we introduced an intentional bug somewhere in the application, and had another part of the group attempt to identify it with only the logs we collect. The results of this bug-hunt can be seen in [2].

4.4 Risk Analysis

4.4.1 Risk Matrix

In this section we present the results of our internal security audit, for a more in-depth description of the items in the matrix please see appendix 9.4.
Below is a table showcasing the likelihood of identified risks and their impact if they were to be exploited.

Likelihood \ Impact	very low	low	medium	high	critical
very low			DDOS	Open ports	SQL injection
low				XSS	
medium					Unauthorized access
high	Knowledge of stack				
very high			Big messages		

4.4.2 Solutions/mitigations

Based on the risk analysis and the problems we found, we came up with solutions to mitigate some of the issues.

Big messages. Sending long messages to saturate the network connection can be mitigated heavily by imposing character limits on the length of messages. **This was implemented by the team.**

Open ports. There is a multitude of port scanning tools available that would allow the team to ensure no port is left open. **This was implemented by the team.**

Unauthorized access. The likelihood of leaking secrets is reduced by us enforcing required reviews on pull requests to the repository, making uploading keys less likely. Likewise the developers should ensure that no authentication information is shared through insecure methods. **This was exercised by the team.**

DDOS. There are a few defense measures one can employ to reduce the impact of DDOS attacks, such as rate limiting, detecting bot patterns and blacklisting bad actors. Some of these measures are hard to build, and is therefore usually outsourced to companies like CloudFlare.

5 Stack Diagram

We have created a diagram to give an overview of our infrastructure and how every service interacts. It shows how the system uses four virtual machines and one managed database to run everything.

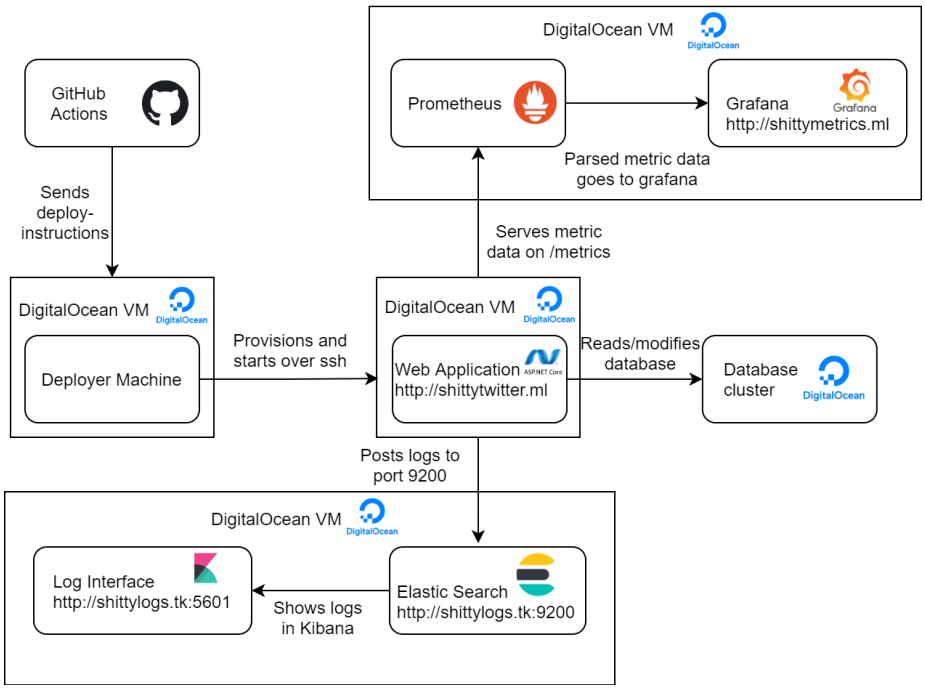

Figure 1: Complete diagram of all services

6 Status on Static Analysis

The static analysis tools we have used are SonarQube (through SonarCloud), Code Climate, Better Code Hub, Snyk, CodeQL and ActionLint. Badges showing the status of all the run tools can be viewed at the top of the `readme.md` file in the GitHub repository [1].

Sonar Cloud: The output of our latest run on Sonar Cloud can be seen here in [4], and shows that we get a score of 'A' on every mark. We have some code smells which relate to null references in the code when we fetch from the database.

Code Climate: Code Climate gives us a good maintainability score, with an 'A' in technical debt. It can be found in [3]

Better Code Hub: We could not get better code hub to exclude parts of our repository (like the old python version of MiniTwit), which results in it giving us a lower score, with many errors. Because of this, we have put less emphasis on the results from Better Code Hub, in comparison to the other tools.

Snyk: Snyk was used to analyse packages and their vulnerabilities. As of writing this report, there are no found vulnerabilities in the MiniTwit project, but throughout the project we used snyk to fix vulnerabilities in a dependency called "Gravatar Helper" and outdated system packages.

CodeQL: CodeQL notified us of hard coded user credentials which were for testing, and some false positive security flaws, due to how the system is built.

ActionLint: We used ActionLint to verify that our scripts on GitHub Actions were not malformed, and thus would not run. As of writing this report they are all correctly formed.

6.1 Code coverage

Using `dotnet test` to capture code coverage in our application. At first glance we get a *fairly bad* score of 13%. This is because it includes a bunch of automatically generated files, such as database migrations, as well as our frontend cshtml, which we agreed not to test. When we narrow our coverage to just the controllers, as well as what we call "Repository files" (files that access the entities, such that the controllers do not access the database directly), we improve to 47.6%. While it is still a relatively low score, we do not test our controllers since they only call the "Repository" files, which are fully tested. Because of this, we believe we have acceptable code coverage, while we could still add tests in controllers.

6.2 License

We have run the ScanCode toolkit to check the licenses of the application, and everything uses the MIT license, apart from the Open Iconic font, which has its own license added in the repository. By the results of this [5], we have decided to use the MIT license for the application.

7 Perspective

7.1 Structuring of infrastructure

We decided early that separating major parts of our infrastructure into separate machines/images/etc. would be a good way to structure the project. As the project progressed, we came to appreciate this separation, as it made each piece easier to manage and easy to swap out.

A concrete example would be our separation of the main application and the database. This separation made deployment easier than trying to move the data from one VM to another without losing it.

7.2 Concurrent deployment caused outage

We used GitHub Actions to deploy our application, unaware of the fact that GitHub Actions defaults to running actions concurrently, instead of queuing them if multiple are triggered. This caused one of our deployment-scripts to fail in such a way that both the new and old production VMs were shut down simultaneously, leaving us without a production server. This was caused by us merging multiple feature branches into the main branch at the same time, and was before we had set up alerts, so the outage lasted for about 30 minutes before we realized the site had gone offline.

The learning experience we acquired from this issue, is that we have to think of our infrastructure just like our code, with critical sections and all the other considerations that comes with concurrent access to shared resources.

We fixed the issue by introducing a flag to make that specific action queue its runs, instead of doing them concurrently:
Fix on github.com

7.3 Handling secrets

During the project work, we had one case of secrets being leaked: Fixed leak

Secrets were kept on the deployer VM, which used them to deploy the application, and the rest were kept on separate platforms such as Discord and GitHub Actions Secrets. This kept it out of anyone else's hands, but ideally we would have set up some machine/key-vault that kept all secrets stored in one place and when needed we fetch them programmatically with individual secrets for each developer. There are two reasons for this. The separate platforms required developers to manually fetch zip files with keys, or copy paste keys individually, so the process was not very streamlined. Security was also at risk, since the secrets were shared over Discord, which meant that Discord kept logs of our messages.

7.4 Quality of tools

We learned that investing time in finding the best tool or workflow is just as important as working with said tool. Our experience with GitHub Actions and Vagrant showcases this well. When we started using GitHub Actions, we found some working examples and went from there. At some point, we encountered an error, and were then left to keep pushing changes to the action to see if the problem was fixed.

We later discovered that there are tools for running actions locally, which allow for faster iteration time, and much better debugging.

Our experience with Vagrant as a provisioning tool was also not the best. It is quite slow - even when we did not use it with GitHub Actions.

We found it very difficult to use SSH keys with Vagrant. Vagrant kept trying to use local keychains or would load the SSH keys in with wrong formatting, and it kept breaking when running the vagrantfile on different members' computers. When we later used Digital Ocean's API and Docker for creating dockerized hosts, we discovered that it was much faster and gave us more freedom to choose how we wanted to interact with the VMs, after they were provisioned. The reduced iteration time would have made configuring our supporting infrastructure more productive, as each attempt with Docker would be several minutes quicker than Vagrant.

8 Bibliography

Bibliography

[1] Aspirin. "Aspirin github repository." (2022), [Online]. Available: `https://github.com/ChadIImus/Devoops` (visited on 05/27/2022).

[2] Aspirin. "Bug hunt postmortem." (2022), [Online]. Available: `https://github.com/ChadIImus/Devoops/blob/master/docs/postmortem.md` (visited on 05/13/2022).

[3] Aspirin. "Latest code climate analysis." (2022), [Online]. Available: `https://codeclimate.com/github/ChadIImus/Devoops` (visited on 05/27/2022).

[4] Aspirin. "Latest sonarqube analysis." (2022), [Online]. Available: `https://sonarcloud.io/summary/overall?id=devops2022` (visited on 05/27/2022).

[5] Aspirin. "Scancode license analysis." (2022), [Online]. Available: `https://github.com/ChadIImus/Devoops/blob/dev/scancode_license.html` (visited on 05/27/2022).

[6] Stack Overflow. "2021 developer survey." (2022), [Online]. Available: `https://insights.stackoverflow.com/survey/2021#section-most-popular-technologies-web-frameworks` (visited on 03/03/2022).

9 Appendix

9.1 Appendix A - Choice of the main language (C#)

9.1.1 Our measures

One of the hardest decisions to begin this project, was to pick the programming language that would work the best. A couple of things we had to take into account were performance, implementation time, difficulty of usage, support, stability, maturity, feature mapping and many more.

9.1.2 Python

Initially, we considered working with Python and Django, since that would make refactoring the initial application much easier while also making the development time extremely fast due to the structure of the language, and the group's experience with Python. After careful consideration, we decided to give up on this idea, since Python does not support explicit typing, which might result in some unnoticed runtime errors. The other factor that influenced our decision was the "running" speed. Python is slower than other considered languages, and it does not support multithreading natively, so we would need to use special libraries, which would result in a higher memory overhead and would detract from the ease of use which python in known for.

9.1.3 Functional Languages

We considered functional languages as well, however we gave up on the idea rather fast, because most of our team is not as experienced in functional paradigm, meaning the development of the application would be more difficult. A major downside was also our lack of knowledge or experience of the frameworks present in these ecosystems.

9.1.4 C#

After some consultation, we decided to go for a language that is object-oriented, since we all know this paradigm quite well. We also wanted to pick a language that has a fast execution time. Based on these criteria, we decided to settle for C#, since it is relatively fast compared to Python for instance. It also has a lot of support in terms of online documentation and libraries, and is really mature in terms of feature it supports. C# is also the choice for many in the corporate world, which also was a considerable benefit. This choice is not necessarily the definitive best choice, as every programming language has their respective pros and cons, but we decided that C# was most fitting choice for our group, and allowed us to expand our knowledge in other areas, such as Razor Pages as well as the new technologies.

9.2 Appendix B - SQL Vs NoSQL

9.2.1 SQL vs NoSQL

In recent times, there is a huge debate over whether to use SQL or NoSQL databases with web applications. There are a lot of advantages to using either structure. NoSQL databases are often used when working with big datasets that need to scale well. NoSQL offers great support for unstructured data that can be accessed quickly. On the other hand, the standard SQL keeps the database in a consistent state, and provides a high amount of integrity, which we find more important than scalability in this project. Another important factor is that the

previous database from the MiniTwit Python was also a SQL one, which would makes the transition to another SQL-based solution much easier.

9.3 Appendix C - Scripting languages

To allow for easier development, we have decided to automate some tasks with popular scripting languages, so that we would not have to click so much, in turn reducing our workload.

9.3.1 Shell

We have been using shell, as it is the standard on all ubuntu systems, which make most of the currently running servers. Furthermore, all of the most important CLI commands are also ported into Powershell, so it is quite easy to to run those scripts on windows. The shell scripts are extremely useful, when we have to do system wide operations such as installing a program or setting up a development environment. They also come in handy when we need to use some CLI tools such as the ones for building our web application or running the tests.

9.3.2 Python

Even though we did not choose Python as our main development language, it is still very valuable. We used it for scripting as these relatively lightweight scripts are well suited to pythons strengths, such as dynamic typing and speed of development.

9.4 Appendix D - Security analysis

9.5 Risk Identification

9.5.1 Assets

The application consists of multiple services and tools that could all potentially be vulnerable to an attack. These are:

- Web application and stack
- Digital Ocean droplets
- ELK logging chain
- Metrics service (Grafana)
- GitHub source control (public repository)

9.5.2 Threat Sources

1. Web application and stack
    - SQL injection
    - cross site scripting
    - Open ports
    - Distributed Denial of Service
    - Big messages (DOS)
2. Digital Ocean droplets
    - Unauthorized access
3. ELK logging chain
    - Unauthorized access
4. Metrics service (Grafana)
    - Public metrics gives information about the application status
    - Unauthorized access
5. GitHub source control (public repository)
    - Unauthorized access

9.5.3 Risk Scenarios

We will now describe the threats and assets in terms of the threats impact on the assets and what security principles they break. To quickly summarize the three CIA security principles:

1. Confidentiality is the principle of keeping sensitive user data out of the hands of anyone but the user and the application.
2. Integrity is the principle of keeping data complete or whole. The data should not be cut, deleted or altered and can therefore be trusted.
3. Availability is the principle of keeping the service or data available without interruption or error.

9.5.3.1 Scenarios - Marked with corresponding security principle violation

C Attacker gains information about users due to incorrect request handling.

C/I Attacker performs cross site scripting to mine user data or fool users to submit their data

A Attacker sends very large message to bottleneck message-handling or fill database.

A Attacker performs cross site scripting to deny users access to the site

C/I/A Attacker performs SQL injection on web application to download/alter/delete possibly sensitive user data.

C/I/A Attacker finds vulnerability in a dependency listed on the public GitHub repository, potentially allowing full access to an arbitrary part of the system.

C/I/A Attacker gains access to any service through leaked secrets.

9.6 Risk analysis

9.6.1 Determine likelihood

1. SQL injection, likelihood: very low
   The likelihood of this occurring is fairly small, as we sanitize the data received from users, this however should always be a strong consideration if new features were to be implemented.

2. cross site scripting, likelihood: low
   Same as the likelihood for SQL injections, the most important factor for this is to ensure proper sanitation of user input.

3. Open ports, Likelihood: very low
   This problem can be verified fairly easily by performing a scan on the machine to verify whether or not this threat exists. One thing that may be worth consideration is how to approach the issue when a project scales though services exist, such as Shodan, Metasploit, etc., to verify if the issue exists on larger networks.

4. Unauthorized access, likelihood: medium
   From examples early in the course we have seen that it is very important to stay vigilant on access to our services, as these can very easily be a either accessed by malicious actors, or completely shutdown.

5. Knowledge of entire stack (Dependencies and versions), likelihood: high
   This one most likely has a high chance of occurring as many web scrapers exist, that try to gain information of services through their repositories. Whether or not this is a problem is up for debate though, as obfuscation alone, is not a strong enough method for ensuring security. The likelihood of bad actors successfully finding useful vulnerabilities is a completely different case which is much less likely.

6. Distributed Denial of Service, likelihood: very low
   This could have a pretty big effect on our service, but in terms of likelihood,

it isn't very high on the list. A distributed denial of service attack would require a level of resources from the attacker, that automatically excludes many adversaries. Because of this we rank it as very low.

7. Big messages (DOS), likelihood: very high
   The likelihood of a bad actor trying to upload big messages is very high, as they are prompted for input, making them curious how big that input can be.

9.6.2 Determine impact

1. SQL injection, impact: critical
   If an SQL injection exploit is found, that gives access to query the database, it could likely give an attacker access to all data, and modifying all data, because of the querying tool having full privileges to the database.

2. cross site scripting, impact: high
   Depending on the type of XSS attack, the impact may vary. If scripts are somehow deposited to the database and rendered by users, it has much higher impact than if an attacker gains access to a session cookie or can log user input.

3. Open ports, impact: high
   When a systems ports are open an attacker can access any vulnerable services running on those ports.

4. Unauthorized access, impact: critical
   An attacker gaining unauthorized access to any of our services will either grant large amounts of information about our data and application, or it will give unlimited access, in form of an API-key, that would let them close the service or take it over.

5. Knowledge of entire stack (Dependencies and versions), impact:very low
   While it is likely that someone will inspect the technology stack, it should not be a cause for concern as we use tools, such as Snyk, to check for vulnerabilities in our dependencies, and tools like SonarQube and CodeQL to check for errors and vulnerabilities in our application, either catching errors before they become problems or mitigating their impact by updating to more secure.

6. Distributed Denial of Service, impact: medium
   The impact of a DDOS attack is loss of availability, but usually not for extended periods of time. It is rare for these issues to persist longer than weeks or days.

7. Big messages (DOS), impact: medium
   Allowing users to send as large messages as they want will likely lead to bottle-necking some part of the request handling. In the worst case, an

attacker can send many messages so large that our database is filled and will require more space which will, in turn, cost the team more money. If we in the future added censorship rules to messages, the application would also have to parse the content of these huge requests, taking many CPU-resources of our backend.