IT UNIVERSITY OF COPENHAGEN

Project Report

DevOps, Software Evolution and Software Maintenance

Group O "TBD"

Andreas Kongstad
kong@itu.dk

Charlotte Schack Berg
csbe@itu.dk

Eythor Mikael Eythorsson
eyey@itu.dk

Christian Lyon Lüthcke
clyt@itu.dk

Frederik Baht-Hagen
fbah@itu.dk

Instructors:
Helge Pfeiffer & Mircea Lungu

May 31, 2022

Contents

1. System's Perspective

1.1. Overview

The initial project was a full stack *Flask* application written in *Bash* and *Python2* with an *SQLite* database. This implementation (henceforth referenced as `TBD-MiniTwit`) included refactoring the initial project to the following containerized microservices:

- *C#* backend using the *ASP.NET Core web framework* and *EF Core* object-relational mapping.
- *ReactJS* single-page application frontend.
- *PostgreSQL* database.

Upon which a monitoring stack, and temporarily a logging stack, were added to be served as a containerized application behind a load-balancer on a managed *Kubernetes* cluster (*DOKS*)

1.2. System Design

The system consists of 5 containerized microservices (see figure 1) of which the largest and most specific to our application is the C# backend. It contains all our business logic and provides two open APIs, an Object-Relational Mapping to our database and exposes application-specific PromQL metrics for *Prometheus*.

This design pattern favors load balancing and horizontal scaling as most microservices serve a singular purpose and are non-critical to each-other.

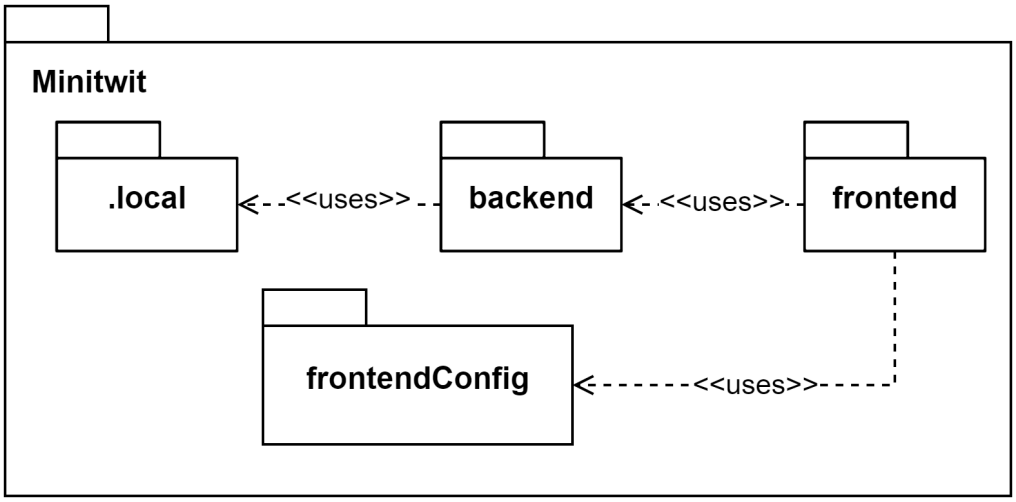

Figure 2: Package Overview Diagram From `TBD-MiniTwit`

The abstract overview of the main packages that makes up the system can be seen in figure 2. Some files which is in the top main package has been excluded to simply the diagram. These excluded files include `Docker Compose` files, setup files for NGINX and Filebeat

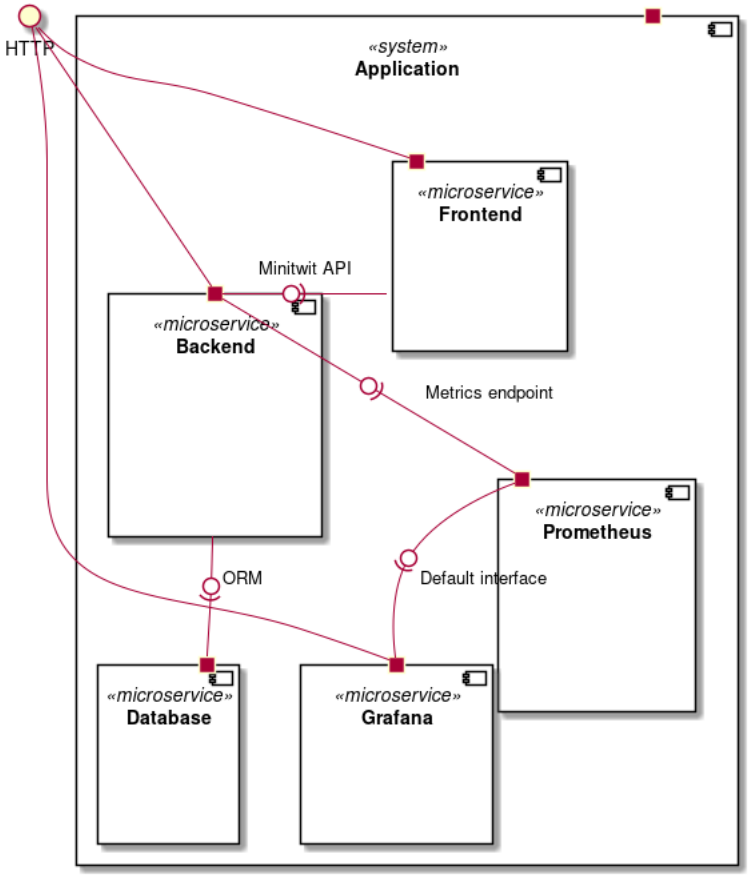

Figure 1: Component diagram of application microservices

etc. `frontendconfig` contains some configurations for the `frontend`, the `backend` contains everything needed for the system to work, and the frontend includes our UI elements.

The most interesting part of the system of `TBD-MiniTwit` is the backend, which decomposition can be seen in figure 3. Some files has been excluded as well in this diagram to simplify it. These files include Dockerfiles, Appsettings files, the program.cs file which is the entry point of the system as well as a legacy controller and its interface for the frontend, which is now only used in tests. The old controller has been updated by separating it into smaller controllers.

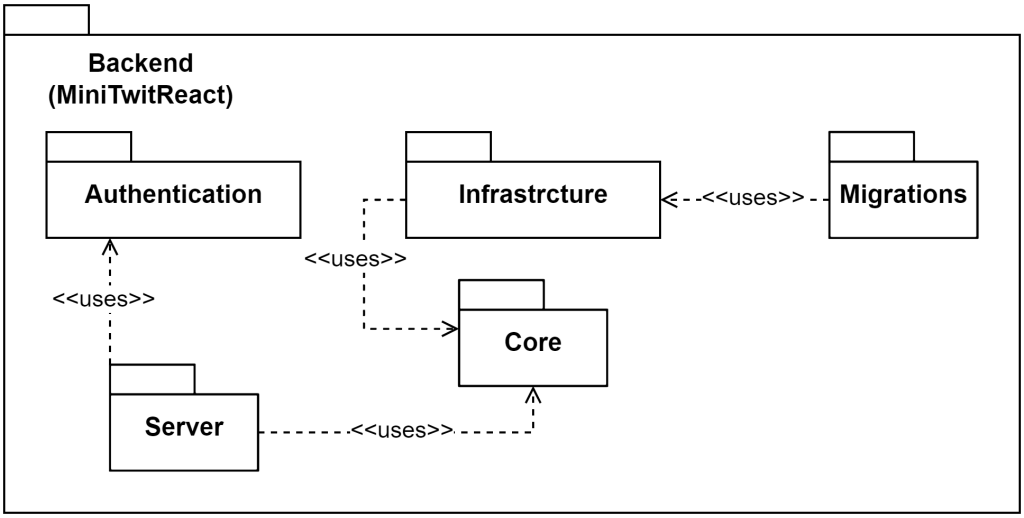

Figure 3: Decomposition of Backend Package of the `TBD-MiniTwit` System

1.3. System Architecture

The implementation of `TBD-MiniTwit` follows the *Microservice Architecture Pattern*. This is achieved by separating the application into smaller, independently deployable parts. In figure 5 it is illustrated how the backend, frontend, database, and monitoring stack are separated and run independently within individual containers. They communicate through the nodes' internal network and receive requests through the ingress controller proxy, which encrypts requests using an auto-renewed SSL certificate. Secrets are retrieved by individual services from key-value storage, while persistent volumes are claimed through a persistent volume claim. See the pod containing the Postgres database in figure 6.

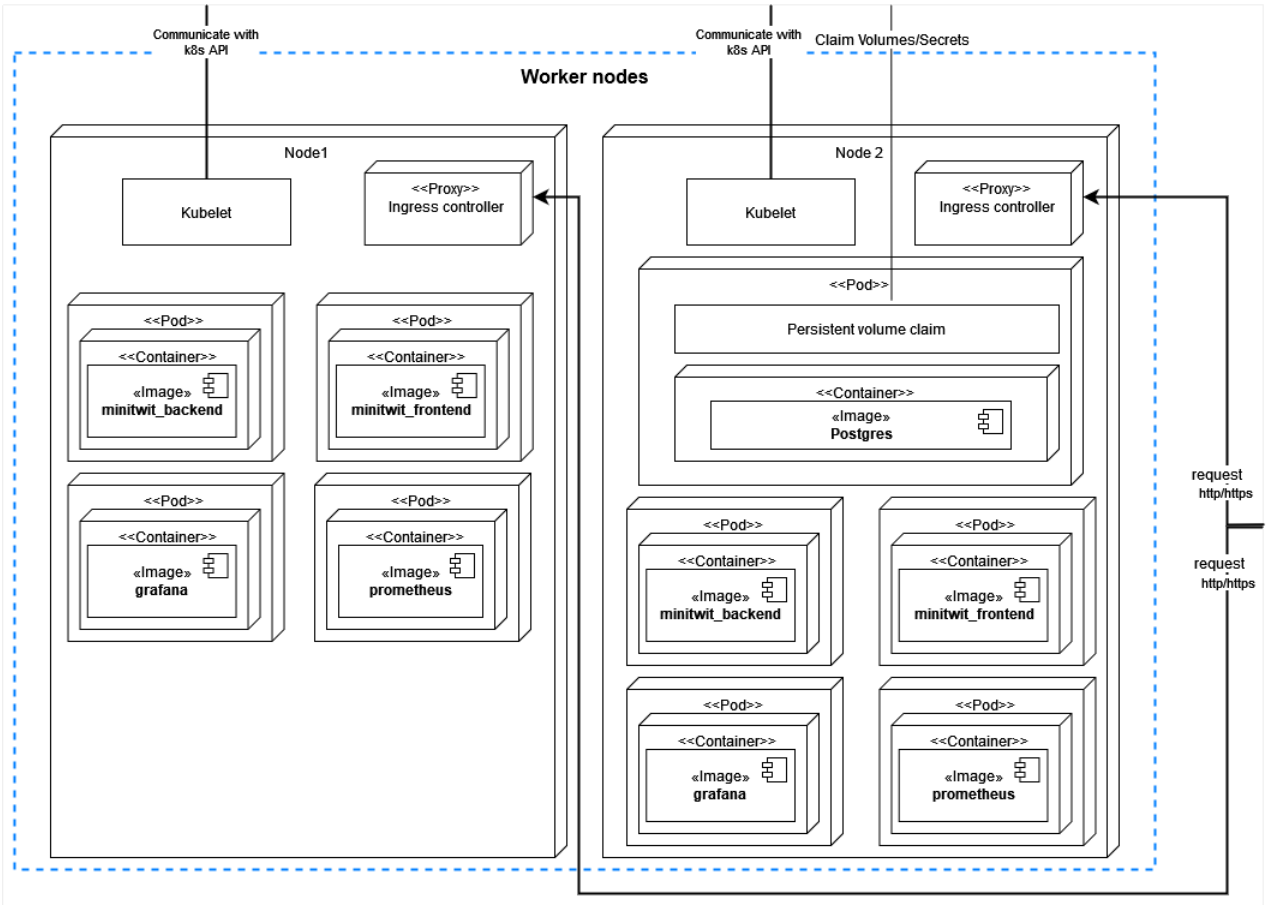

Figure 4: `TBD-MiniTwit` deployment diagram decomposition of worker nodes.

The benefits of using microservices include updating and deploying individual services instead of having to bring the entire system down for an update and the option to integrate different stacks and programming languages without issue, e.g., frontend, backend, and monitoring stack. Services can also be scaled independently to support the increased load.
Microservices do, however, not come without challenges. Management and deployment complexity increases heavily when having to deploy services individually. That is where DevOps enters the picture. "Microservices both enable, and require, DevOps" according to IBM. The reason being that adopting the microservices approach would be unmanageable without implementing DevOps practices like automated ci/cd and monitoring[1].
The system still needs to support scaling and load balancing, which is achieved by deploying services to a Kubernetes cluster running on digitalocean. Further information on the migration to Kubernetes will be provided in 2.7 scaling and load balancing
Kubernetes is used for container orchestration to automate service discovery, networking, load balancing, rolling updates, and service health checking. figure 6 provides a deployment diagram visualizing the deployment of the `TBD-MiniTwit` service on the Kubernetes cluster managed by the digital ocean and shows the worker nodes from figure 5 interact with the rest of the cluster.

[1] ibm[9]

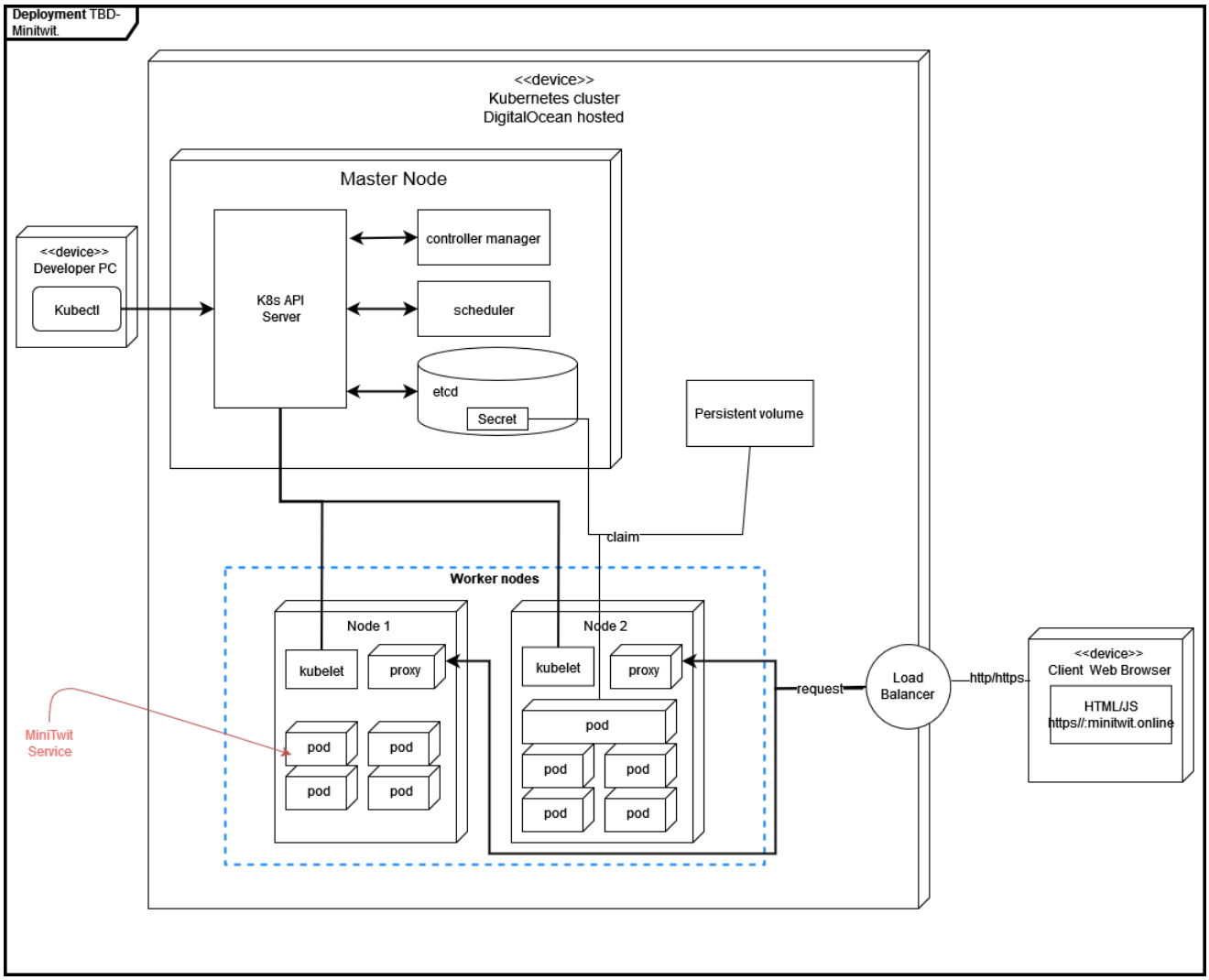

Figure 5: `TBD-MiniTwit` deployment diagram

Through the Kubernetes command-line tool, kubectl, services can be deployed, updated, scaled, and inspected. The load balancer automatically distributes client requests to the proxies of different worker nodes.

1.4. Technologies & Tools

`TBD-MiniTwit` depends on a number of different technologies and tools. This section will briefly describe these dependencies.

1. **Docker**
2. **Docker-Compose**
3. **DockerHub**
4. **Markdown**

5. **Markup languages**
6. **.NET 6** - Used for building the backend. Includes:
    - **C# 10**
    - **ASP.NET Core**
    - **NuGet** - The .NET package manager. The application dependency tree include all packages installed and their dependencies.
    - **Entity Framework Core**
7. **React** - Used for building the frontend. Includes:
    - **JavaScript**
    - **WebPack**
    - **npm** - The node package manager. The application dependency tree include all packages installed and their dependencies.
8. **PostgresQL**
9. **Grafana**
10. **Prometheus**
11. **Nginx**
12. **GitHub**
    - **Git**
    - **GitHub Actions**
    - **dependabot**
13. **Code scanning tools**
    - **sonarcloud**
    - **CodeClimate**
    - **BetterCodeHub**
    - **DeepScan**
    - **snyk**
14. **Terraform**
15. **Bash**
16. **DigitalOcean**
17. **Kubernetes**
18. **LetsEncrypt**

1.5. Subsystem Interactions

A notable downside of the microservice architectural pattern is the unreliability of networking interfaces. The key feature of eliminating single points of failure and providing high availability to services makes tracking ip addresses problematic. Kubernetes solves this with the Service object.

1.5.1. The Service Object

Services are ADTS providing stable IP addresses, DNS names and ports via loose coupling to Deployments. Traffic to microservices is sent to DNS addresses which the internal cluster DNS resolves to IP addresses of the relevant Services. Services then in turn route the traffic to healthy pods by way of an Endpoint object which stores a dynamic list of healthy Pods matching the service object's labels [3].

1.5.2. The Cluster Network

Kubernetes abstracts a cluster of hosts to a single platform which behaves in many ways as a decentralized operating system, e.g. by providing a filesystem, dns, network and a means to share memory and computational resources.

The Cluster Network operates similarly to a local network on any regular host in such a way that any device or process on the network is discoverable by other entities on the same network. Hence, given that a pod knows the IP of another pod on the cluster network, traffic between them is possible, though traffic via Service objects is preferable.

1.5.3. The Ingress Object

Ingress is all about accessing multiple web applications through a single Service object [3]. The Service object has several subtypes, two of which provide a one-to-one mapping between an internal Service and a public port. Ingress is a Kubernetes resource which serves various purposes but most importantly provides a reverse proxy to multiple Service objects.

1.5.4. The Load-Balancer

Of the briefly mentione

Prometheus
MinitwitAPI
LoadBalancer

Table 1: Network interfaces

1.6. System state

To be able to argue about the quality of our code, we have enhanced our *CI* pipeline with static code analysis tools. These analyze and rate `TBD-MiniTwit` per their definition of code quality.

1.6.1. Code Quality, Technical Debt, & Maintainability

According to the *Better Code Hub quality status*[2], our codebase complies with 9 of 10 measures for quality while failing on code duplication. This is because `TBD-MiniTwit` supports both a simulator API and an API for the frontend. When refactoring infrastructure code for the frontend API, we kept the infrastructure for the simulator resulting in code duplication.
Sonarcloud suggests that we have a technical debt of 55min, all of which is from the `Simulation Controller` and its infrastructure class, indicating that the simulation `controller` and `repository` requires refactoring. It also located 8 code smells but still gives the main branch a grade of A[3]. In contrast *Code Climate*[4] scores the technical debt of the system as 2 weeks and finds 17 code smells. This partly shows the difference in the definition of code quality between different tools. However, it is also caused by Code Climate analyzing the entire repository. At the same time, Sonarcloud analyzes only the backend as seen in the Sonarcloud language distribution stats[5] compared to the Code Climate language distribution graph[6]. The frontend contributing heavily to technical debt was expected. The team did not have much experience with React going into the project, so we prioritized the frontend's required functionality and spent time implementing weekly assignments and improving the backend. The difference in the definition of code quality of these tools is fascinating. *DeepScan*[7] suggests no code quality issues.
In conclusion, the technical debt of the frontend hurts the maintainability of the system. The backend has little to no technical debt, and code quality is good except for the simulation controller and simulation repository needing refactoring.

1.6.2. Vulnerabilities

2 dedicated security vulnerability scanning tools are used to increase chance of discovering and removing vulnerabilities as fast as possible. Both *snyk*[8] and *dependabot*[9] finds a vulnerability of high severity stemming from a *npm* dependency, enforcing the fact that it is a known vulnerability. In conclusion, the `TBD-MiniTwit` code base contains two vulnerabilities in the frontend npm modules.

1.7. License Compatibility

The initial license chosen was *MIT*, which means everyone can use and modify the code freely. However, when we used a tool called *ScanCode*[10] to determine if our chosen license would clash with a license in the imports we use by scanning the files in the project. Through the result we discovered that some of our imports used *Apache License 2.0*[11] and *BSD-3-Clause*[12], meaning we

[2] Better Code Hub quality status
[3] Sonarcloud Maintainability Scores
[4] Code Climate
[5] Sonarcloud language distribution stats
[6] Code Climate language distribution graph
[7] DeepScan
[8] Snyk
[9] Dependabot
[10] ScanCode toolkit documentation[11]
[11] Apache License 2.0 document[4]
[12] BSD-3-Clause[6]

had to change license. Some imports used all three licenses mentioned above. The tool has briefly been mentioned during the course. The new license is Apache 2.0 as a result of running ScanCode to have a license that is compatible with our imports. The scan seemed to fail on some licenses as it stated them as `Unknown license`, however, since the other licenses encountered are the three mentioned previously, then it is likely not to be a problem. Additionally, when manually searching for the licenses on the imports we use in the frontend we discovered they that the majority used MIT license, and one used Apache license 2.0. Therefore the BSD-3-Clause might be some *nodejs* dependency as the license is located in a few development packages alongside Apache license 2.0 and MIT license, but in none of used imports in the code.

2. Process' Perspective

The main focus of the Process' Perspective section will be how code and artifacts go from idea to production and which tools and processes are involved?
2.2 Team Interaction and Organization will discuss communication within the team. 2.3 CI/CD chains will reason for the choice of CI/CD platform and take a deeper look and both CI and CD chains. 2.4 Version Control discusses strategies related to the way version control is used and managed. These include repository setup and branching strategy. In 2.5 Development process and tools we discuss how issues and projects were used to track, label, and assign tasks within the team. 2.6 Monitoring And Logs will introduce the Prometheus Grafana stack and the elk stack and what value these have brought to service. 2.7 Security will be a brief rundown of results from the security analysis and pentest, and an introduction to the techniques used for secret handling. The final section 2.8 Scaling And Load balancing will focus on the migration from single server architecture to a Kubernetes cluster and provide a discussion on some of the pros and cons of using Kubernetes for scaling and load balancing.

2.1. Team Interaction and Organization

Most internal planning and communication have gone through Discord. The team has a server with text channels for planning, resources etc. Work has been split between remote using discord voice channels and physical at and in continuation of the allotted exercise time on Tuesdays.

2.2. CI/CD chains

Utilizing a CI/CD pipeline allows us to test, build, and deploy `TBD-MiniTwit` incrementally with minimal manual interference. Thus, saving us a significant amount of time setting up environments, allowing us to deploy fast and often while still being able to revert to an earlier version quickly. It provides us with ways of measuring and improving the quality of all code coming from local development to version control and eventually reaching production, and by that reduces human error.[13] In essence, an automated CI/CD pipeline puts multiple DevOps ideas into practice:

- Flow (Keeping batch sizes small)[14]
- Feedback (Instant, rapid and continuous feedback on code entering pipeline)[15]

2.2.1. CI/CD Platform

An abundance of CI/CD platforms are available to developers. Our CI/CD chains are set up using GitHub Actions. GitHub Actions integrates seamlessly with our GitHub repositories and allows us to trigger workflows every time an event occurs in a repository[16]. Many other providers such as Travis CItravisCI or TeamCityTeamCity offer these same features and some even more, but not for free and not with little to no initial configuration. However, we are not blind to

[13] Continuous delivery: Huge benefits, but challenges too[1]
[14] The DevOps handbook[2]
[15] The DevOps handbook[2]
[16] Understanding GitHub Actions[13]

concerns with using the same provider for most tools. The most notable of these would be more considerable consequences when GitHub inevitably goes offline or experiences problems. We accept these concerns as we prefer the ease of use and price tag over distributing our tools and having a smaller chance of being impaired due to provider outages.

2.2.2. CI - Continuous Integration

As illustrated in Figure 1, the entry point for the CI pipeline is creating a pull request to the main branch, which will trigger several GitHub Actions workflows.

1. **.Net Build and test** - The backbone of the CI pipeline, which compiles, builds, and tests the backend. It provides us immediate feedback on whether the changes run and passes the test suite. When implementing the backend, we spent a significant amount of time and effort setting up a test suite comprised of unit tests and integration tests, which allows us to trust that this workflow will catch breaking change coming into the codebase.

2. **Quality analysis tools** - These tools all provide feedback on the quality of the pushed code per their definition of code quality. These include .Net analyzers, Sonarcloud, Better Code Hub, and DeepScan.

3. **Dependency scanning tools** - Scans dependencies and codebase for security hazards and fails if the security gate specified to "Critical" severity is met. These include snyk and CodeQL.

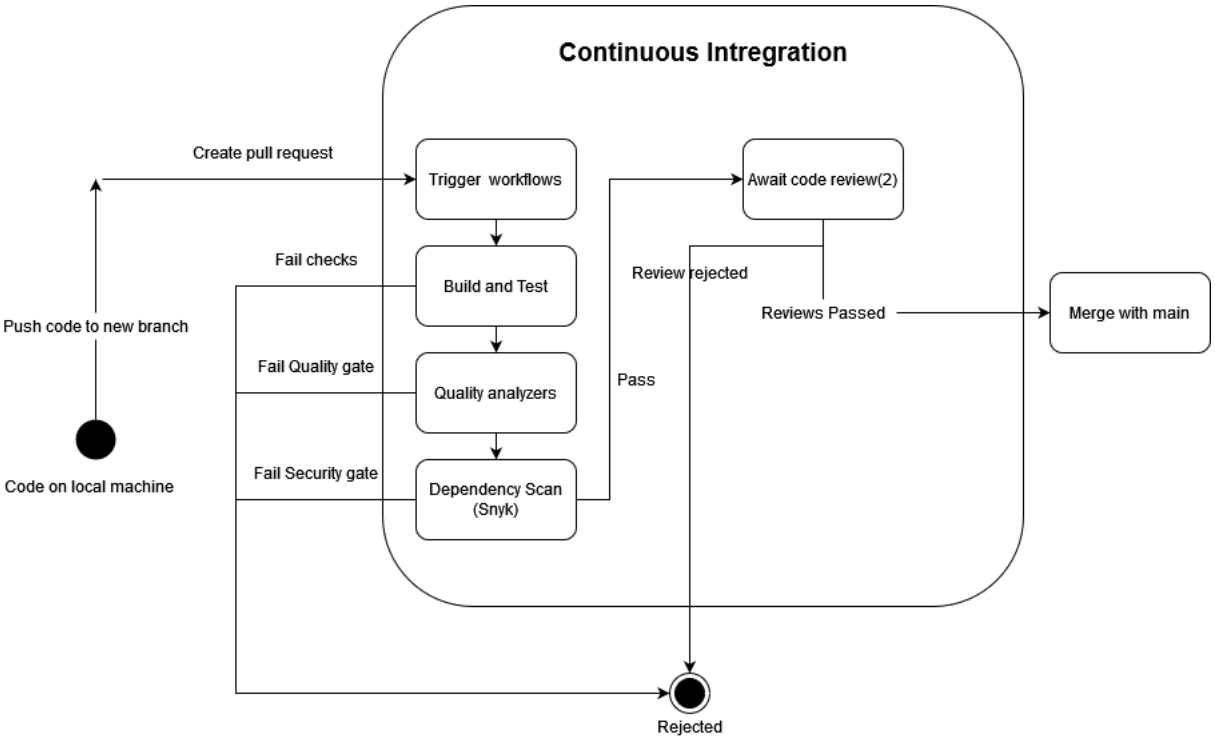

Figure 6: CI Pipeline State Machine Diagram

As suggested by figure 1, if any of these fail, the CI pipeline will direct to a rejected state, from where a developer can fix problems, push to the rejected branch, and the workflows will rerun. Once all workflows are passed, the pull request awaits review until at least two team members have approved it. From here, changes can be merged into the main branch.

2.2.3. CD - Continuous Delivery/Deployment

Our pipeline introduces a mix of Continuous Delivery and Continuous Deployment(Illustrated in Figure 2). Deployment is done entirely by the deployment workflow(cluster-deploy.yml). The workflow is triggered every time a release is created. It also supports manual dispatch for hot fixing errors, and the weekly release workflow runs every Sunday evening, triggering the deployment pipeline. The deployment workflow is comprised of 4 jobs.

1. **.Net Build and Test** - This job is described in CI - Continuous Integration.
2. **Build and Push containers** - Builds docker containers for the frontend and backend, tags them with proper versions, then pushes them to docker hub. This allows us to keep our Operations repository lean as we can then pull all necessary containers from docker hub.
3. **Snyk Dependency scan** - Our security gate. If a risk exceeding the gate is found, the deployment will stop immediately and move to the canceled state. See Figure 2
4. **Dispatch** - Dispatches the apply workflow in the Operations repository.

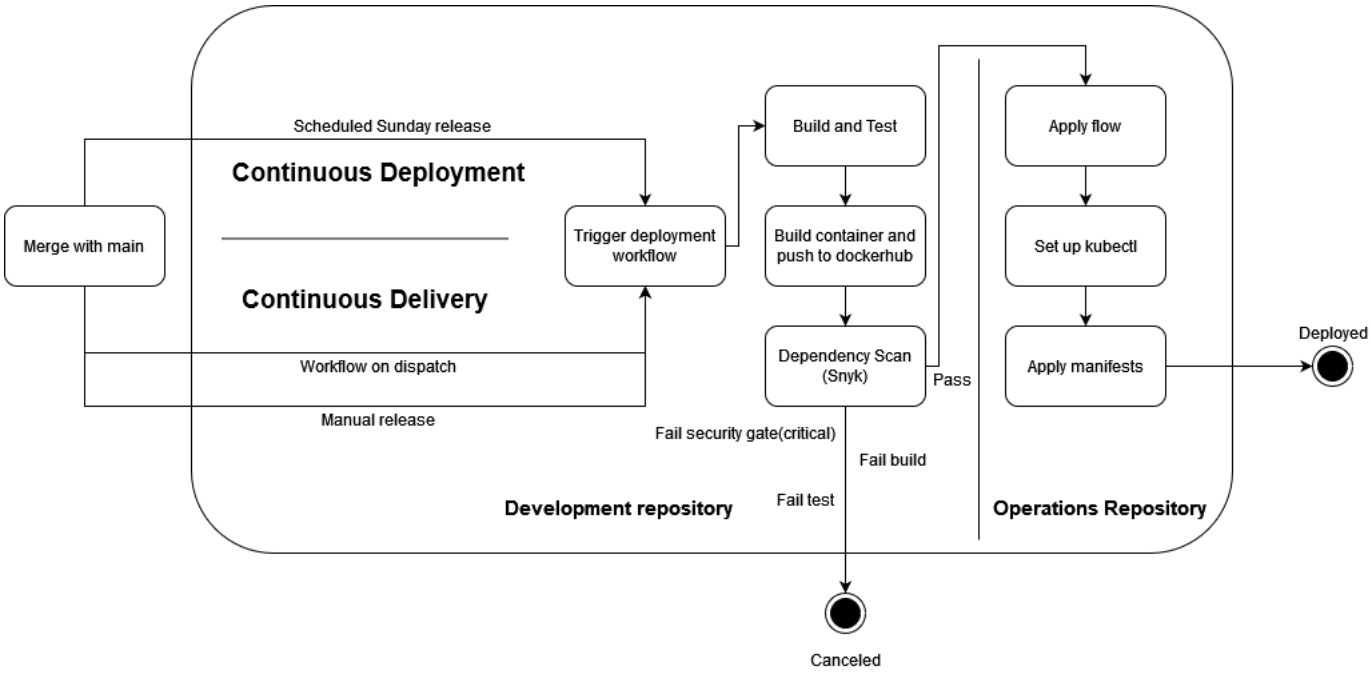

Figure 7: CD Pipeline State Machine Diagram

When checks have passed and the apply workflow has been dispatched, the runner will set up kubectl using a Digital ocean token, with the cluster name saved as secrets in the repository.

With access to kubectl, the runner now has access to the Kubernetes cluster. The apply shell script is executed, which "applies" all the configuration manifests, thus deploying the changes.

2.3. Version Control

2.3.1. Organization of Repositories

We have submodules for extra repositories, and use automatic steps to update them. We have two submodules, one for the report, and one for the deployment.

2.3.2. Branching Strategy

Our organization uses a Trunk Based Development branching model. We have two centralized branches, which were continuously debugged, tested, and in a working state. Features, patches, and bug fixes are always developed on dedicated temporary branches, which are deleted after being merged into their relative centralized branch.

Dev Repository @ main

The **main** branch lives on GitHub and is the alpha of our centralized workflow. While we develop features and patches on temporary branches, everything worthwhile is eventually merged into main.

Primary applications of the main branch are:

1. Store our working source code
2. Make releases
3. Run workflows
4. Build and push docker images

Ops Repository @ master

The master branch of the Ops repository contains infrastructure as code for creating and configuring a Kubernetes cluster with DigitalOcean as provider, manifest files used to deploy our service to the cluster, and shell scripts for automating these steps. It is developed in a manner such that our multi-container application can be rebuilt and updated to include updated Docker images and changes in configuration files requiring minimal effort.

2.4. Development process and tools

GitHub issues track what needs to be done and how far a task is from completion. Upon creation, issues are tagged and organized into GitHub Projects. By using a combination of issues, projects, and enforcement of code reviews, we promote transparency between developers, thus making sure to spread progress and knowledge of the codebase between multiple developers.

2.5. Monitoring And Logs

2.6. Security

To discover, define, and asses vulnerabilities of our system, we have conducted a security assessment and a pen-test. This section will include only select results. The rest is available in Secuirity Assesment section of the Appendix.

Using the OWASP Top 10 we identified possible insecurities in our system. We constructed risk scenarios and analyzed their likelihood and impact. The analysis yielded "Outdated Components" as a top concern. As security breaches on are discovered half a year later on average, the way to combat security threats is proactivity[17]. To decrease chance of having outdated components in production, we added dependabot to our GitHub repository and snyk to our CI chain. Dependabot creates pull requests automatically suggesting updates to outdated components. Snyk scans the repository for dependencies with vulnerabilities. It also scans for sensitive data leaks. Handling of secrets to prevent such leaks will be described in the Secret Handling section. We also conducted an automated penetration test to:

1. Detect vulnerabilities
2. Test the system under stress.

Using the logging system, we noticed that the server received requests from all around the world, e.g. Nevada. In conclusion, except for acting as a DOS attack on our own system, eventually crashing the ELK stack. The pen test did not yield any system vulnerabilities.

2.6.1. Secret Handling

Securing sensitive data while allowing access to multiple parties is a challenging ordeal. We use GitHub secrets for securing tokens and keys that are then accessed and injected into our CI/CD pipeline. We have a local folder containing the database connection string, database-password, and jwt token key. The folder is shared through discord. Security could be improved by splitting the key into multiple parts and sharing them separately. However, we deemed this excessive.

[17] Security lecture. [12]

2.7. Scaling And Load balancing

2.7.1. Single server setup

The original single server setup that was deployed on a digital ocean droplet using docker-compose is located on **Dev Repository @ production** Although now deprecated, the production branch had our production server and was our main branch's lean and automated counterpart. It was developed in a manner such that our multi-container application could be rebuilt and updated on the production server to include updated Docker images and changes in configuration files with a single command, minimizing downtime without using load-balancers.

Although cheap and easy to set up. The original single-server architecture had several shortcomings if we were to handle additional traffic while minimizing downtime. Only vertical scaling using the digital ocean API was possible. This approach has multiple downsides. It will become exceedingly more expensive as more virtual CPU cores, RAM, and disk space is added, eventually reaching an upper limit. The single monolithic VM will forever be a single point of failure, and upgrading the VM will require the server to shut down.

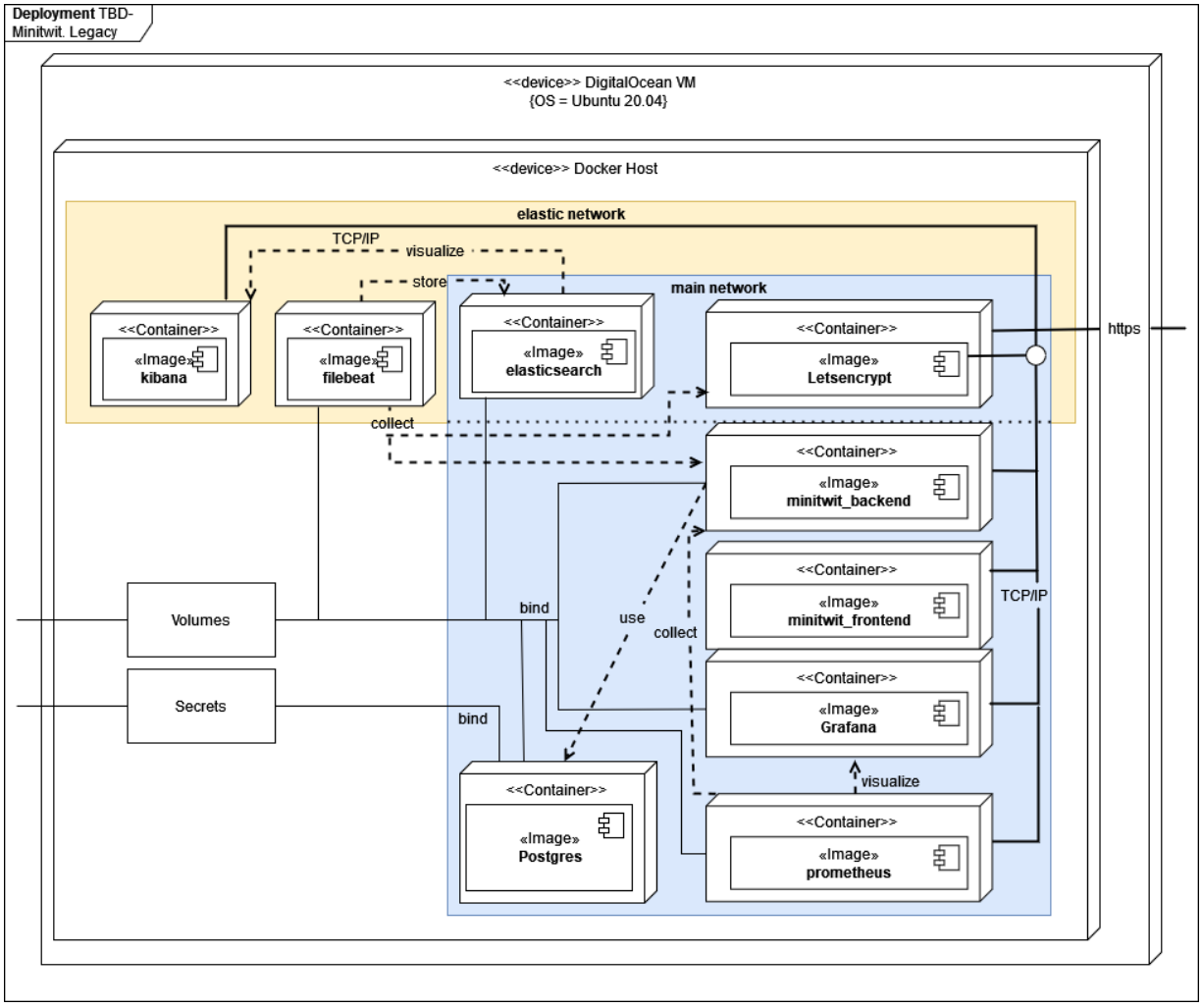

Figure 8: `TBD-MiniTwit` single server deployment diagram

2.7.2. Scaling the application

In order to scale for handling additional traffic while minimizing downtime, the team had a couple of options. Eliminating the server as a single point of failure while allowing for rolling updates out without shutting down the application could be accomplished by introducing a setup with a primary server and backup server while swapping around IPs, but that would not allow for horizontal scaling.
Options that, out of the box, come with horizontal scaling, load balancing, and rolling updates while eliminating the single point of failure could be container orchestration tools like Docker Swarm and Kubernetes.
We have chosen to migrate our application to a Kubernetes cluster. There is an argument that the additional complexity and setup required for deploying a Kubernetes cluster is unnecessary since Docker Swarm fulfills our requirements, thus conflicting with the simplicity principle. After all according to the agile manifesto, "Simplicity–the art of maximizing the amount of work not done–is essential"[18]. The first reason for the choice is that documentation on the setup and management of a Kubernetes cluster was a lot more extensive, although the increased complexity might cause that. Secondly, the team wished to gain experience using Kubernetes for container orchestration. That said, automated scaling[19] does save time for developers, even if monitoring service load and manually scaling a swarm when required is a possible solution. In conclusion, by migrating to a Kubernetes cluster, we support horizontal scaling, load balancing, and rolling updates. However, we must admit that even though the database was flushed and deployed to the cluster, we do not have a replicated database because of consistency concerns and can therefore not claim to have eliminated all single points of failure.

[18](Beck, et al., 2001)[5]
[19]https://kubernetes.io/

3. Lessons Learned

The following section contains the lessons learned, bigger issues encountered, how they were solved, and reflections on the project.

3.1. Refactoring

An issue we had when refactoring from the original mini-twit to C# and ReactJS was *transcribing* the flask frontend to ReactJS. Most of the team had little to no experience with the framework before working on the project, though the biggest issue were understanding what the flask module were doing. The team solved the issue by stop looking at the flask frontend and make our own with inspiration from the original look in the UI. The team learned to understand python when refactoring to C# as well as when we needed to make the API for the simulation. It was easier determining what the language in the backend should be as most preferred C# , another suggested language would have been python without the flask module.

3.2. Maintenance

As the simulation were running, we made maintenance on the program, fixing smaller issues such as the login and register for the simulator did not seem to be working on the start day of the simulation, which we caught onto and fixed in the same evening[20]. We figured out the issue might was something with how the checked as well as not awaiting async calls as our methods in our controllers are async, meaning we have to await them when calling the methods. While fixing this issue we did some clean up on the affected parts of the code to have cleaner code and prevent that some odd code would case unexpected bugs. However, this issue caused us to lose data as we could not register users, but after the refactoring the code, and when the next batch of users from the simulation were sent to our API the problem no longer occurred. Another issue we had were that our follow and unfollow endpoints suddenly stopped working[21]. The problem were indirectly solved when we did more refactoring on the system, however we would likely not have caught the issue if it were not for the monitoring we had implemented at that time.

3.3. DevOps Adaptation

During the project the team has implemented automatic releases, meaning we release every week at a specific time. This is done through a GitHub Action called `release_schedule`, which runs every Sunday at 22:18. This action can also be triggered manually in case something goes wrong with a release, so we can quickly make another release[22]. Additionally, as mentioned earlier in 1.6, we have multiple Actions running on every push and pull request to the main branch, that aims to ensure code quality, etc. checking if there is any code duplication, unused variables and imports etc..

Using automatic releases and static code analysis tools have helped improving the overall quality of the code as well as ensure that we are releasing once every week with automatic release

[20] pull request fixing login & register[7]
[21] issue illustrating problem[8]
[22] release_schedule workflow[10]

notes of new accepted changes to the main branch. This also means we do not have to think too much about releasing as it is done automatically for us, meaning we have more time to focus on other aspects of the project, such as maintenance.

References

[1] L. Chen, "Continuous delivery: Huge benefits, but challenges too," *IEEE Software*, vol. 32, 2 2015, ISSN: 07407459. DOI: 10.1109/MS.2015.27.

[2] G. Kim, J. Humble, P. Debois, and J. Willis, *The DevOps Handbook : How to Create World-Class Agility, Reliability, and Security in Technology Organizations*. 2016.

[3] N. Poulton, *The Kubernetes Book: 2022 Edition*. 2022.

[4] *Apache license 2.0*, https://www.apache.org/licenses/LICENSE-2.0, (Accessed: 31/05/2022).

[5] K. Beck, M. Beedle, A. van Bennekum, *et al.*, https://agilemanifesto.org/.

[6] *Bsd-3-clause*, https://opensource.org/licenses/BSD-3-Clause, (Accessed: 31/05/2022).

[7] *Frontend login/register. #102*, https://github.com/Akongstad/DevOps-group-p/pull/102, (Accessed: 31/05/2022).

[8] *Investigate and solve follow issues #172*, https://github.com/Akongstad/DevOps-group-p/issues/172, (Accessed: 31/05/2022).

[9] *Microservices*, https://www.ibm.com/cloud/learn/microservices, (Accessed: 31/05/2022).

[10] *Release_schedule*, https://github.com/Akongstad/DevOps-group-p/blob/410e163f267f20ccdee18545.github/workflows/release_schedule.yml, (Accessed: 31/05/2022).

[11] *Scancode toolkit*, https://scancode-toolkit.readthedocs.io/en/latest/index.html, (Accessed: 26/05/2022).

[12] *Security lecture slides*, https://github.com/itu-devops/lecture_notes/blob/5df8b93e7af22f57a0ce6a556202510c35a92b29/sessions/session_09/Security.ipynb, (Accessed: 31/05/2022).

[13] *Understanding github actions*, https://docs.github.com/en/actions/learn-github-actions/understanding-github-actions, (Accessed: 18/05/2022).

A. Appendix

A.1. Constitutional Artifacts

A.1.1. Development Repository

https://github.com/Akongstad/DevOps-group-p

A.1.2. Operations Repository

https://github.com/mikaeleythor/itu-minitwit-ops/tree/master

A.2. Security Assessment

https://github.com/Akongstad/DevOps-group-p/blob/main/SECURITY.md

A.2.1. Tests

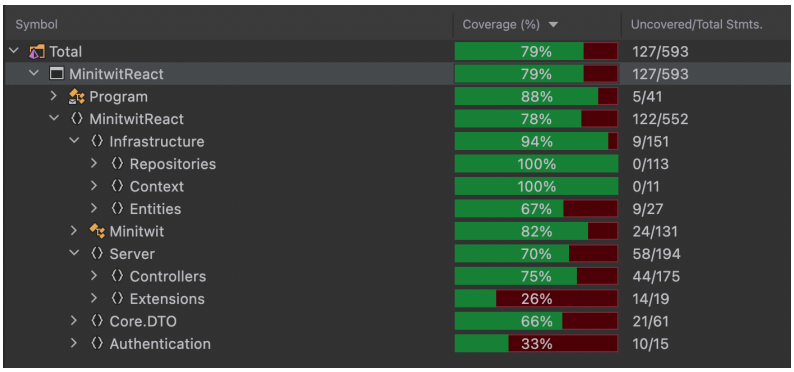

Figure 9: Test coverage 31/05/22. (Generated code and main excluded)

A.2.2. Monitoring

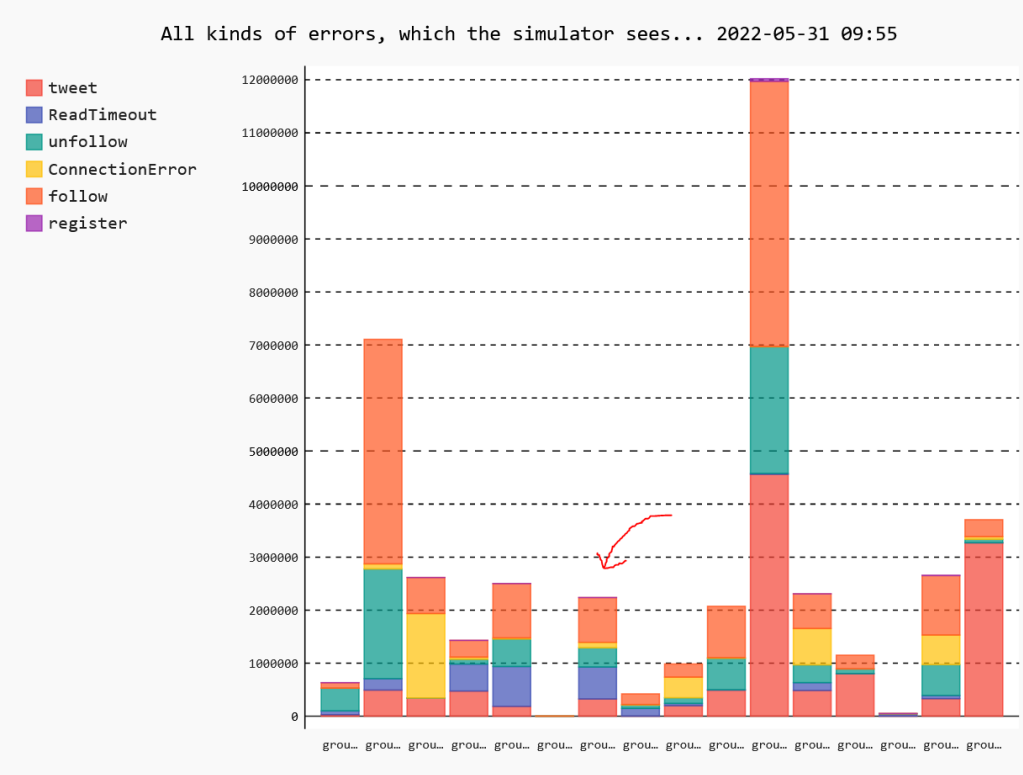

Figure 10: Monitoring Dashboard from the simulator

A.3. Code analysis dashboards

A.3.1. sonarcloud

https://sonarcloud.io/project/overview?id=Akongstad_DevOps-group-p

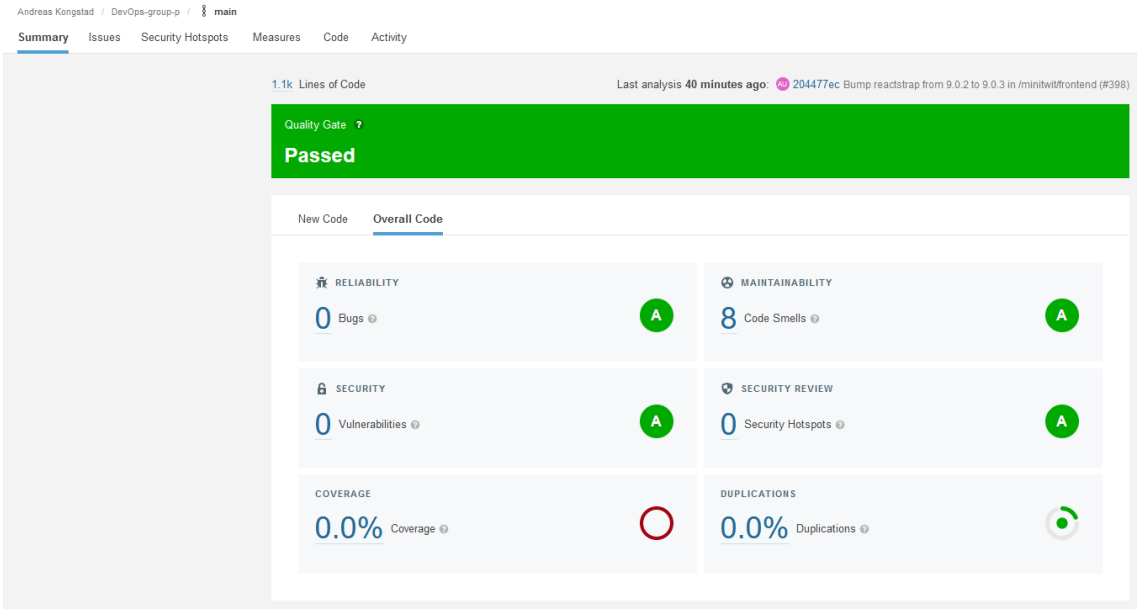

Figure 11: Sonarcloud maintainability scores 31/05/22

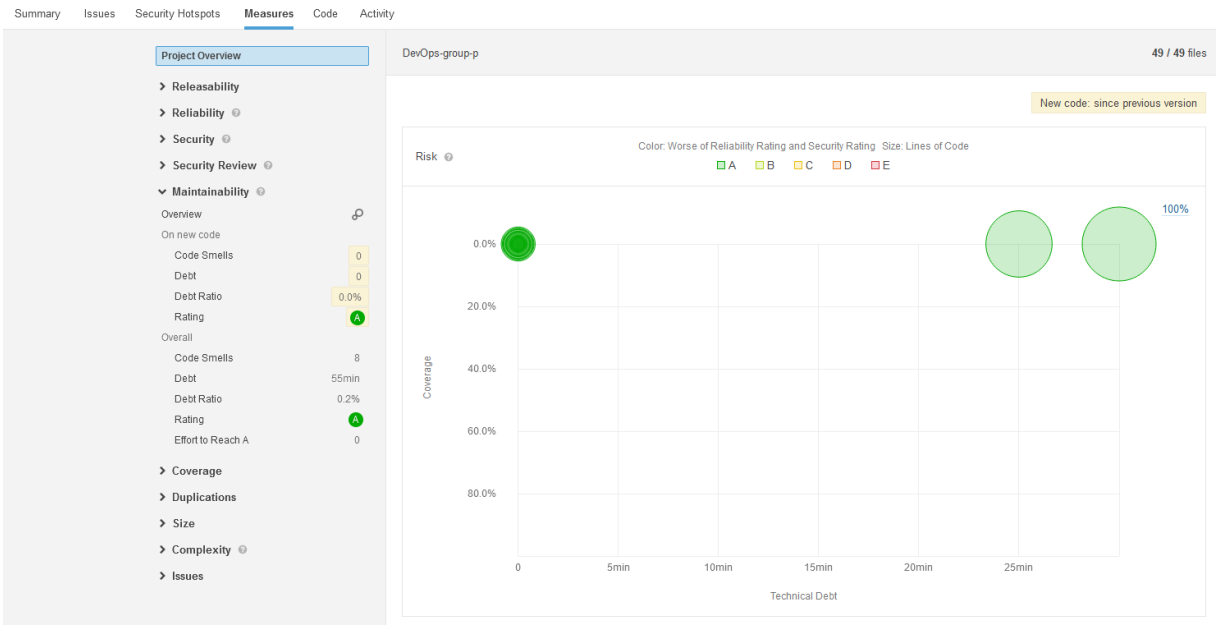

Figure 12: Sonarcloud maintainability scores 30/05/22

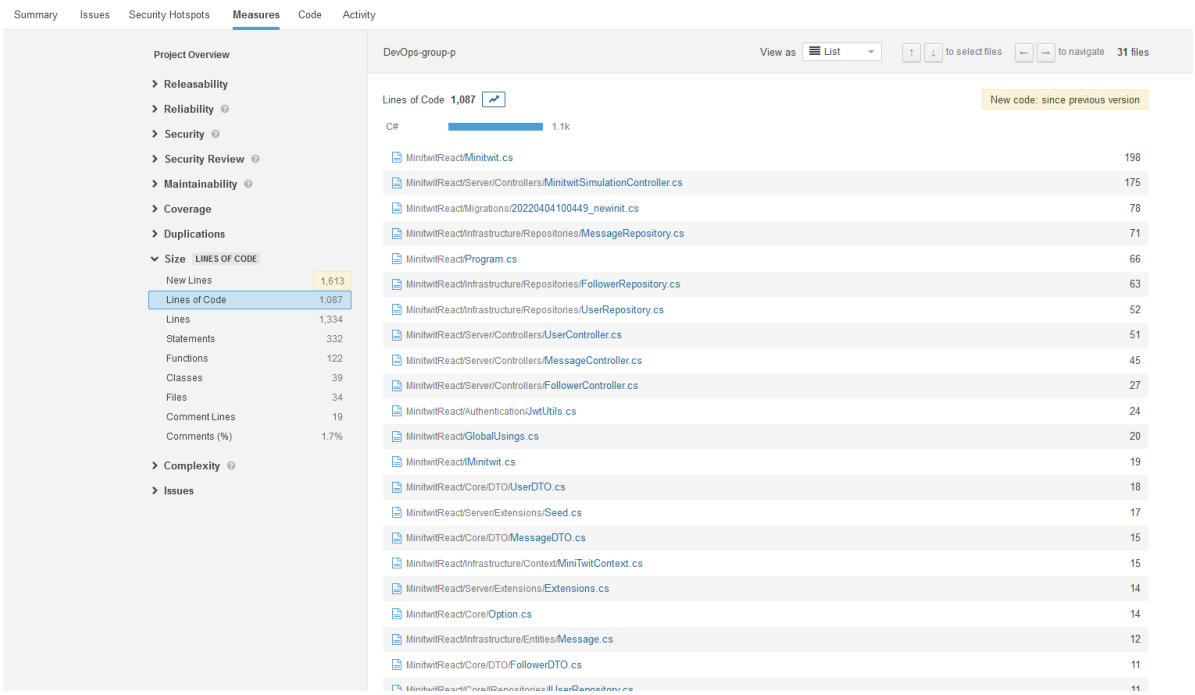

Figure 13: Sonarcloud language distribution 31/05/22

A.3.2. Code Climate

https://codeclimate.com/github/Akongstad/DevOps-group-p

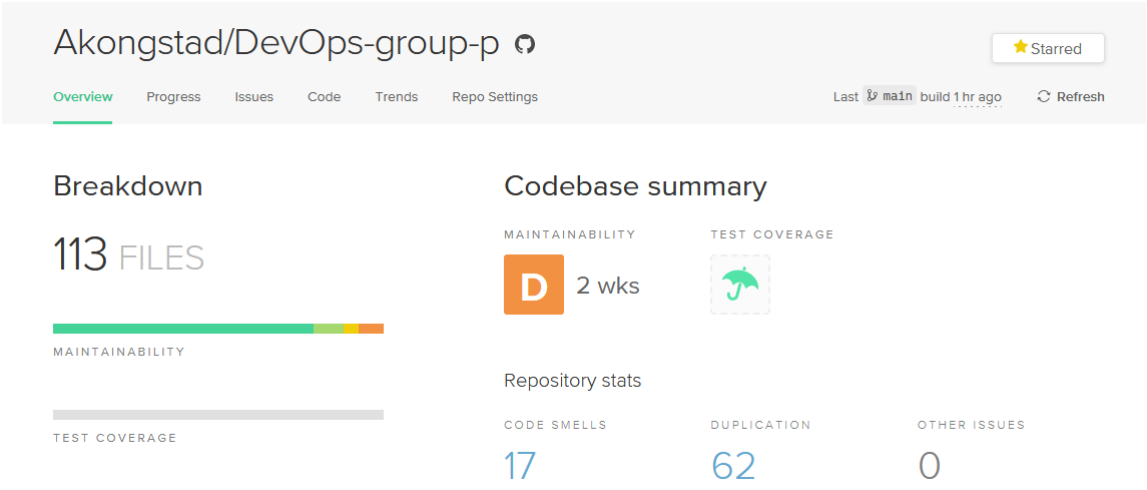

Figure 14: Code Climate repository status 31/05/22

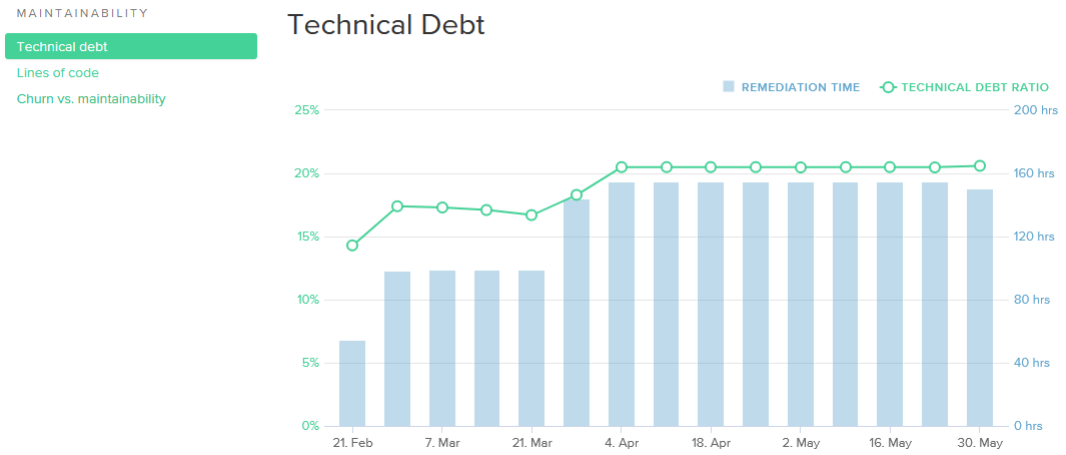

Figure 15: Code Climate technical depth progression graph 31/05/22

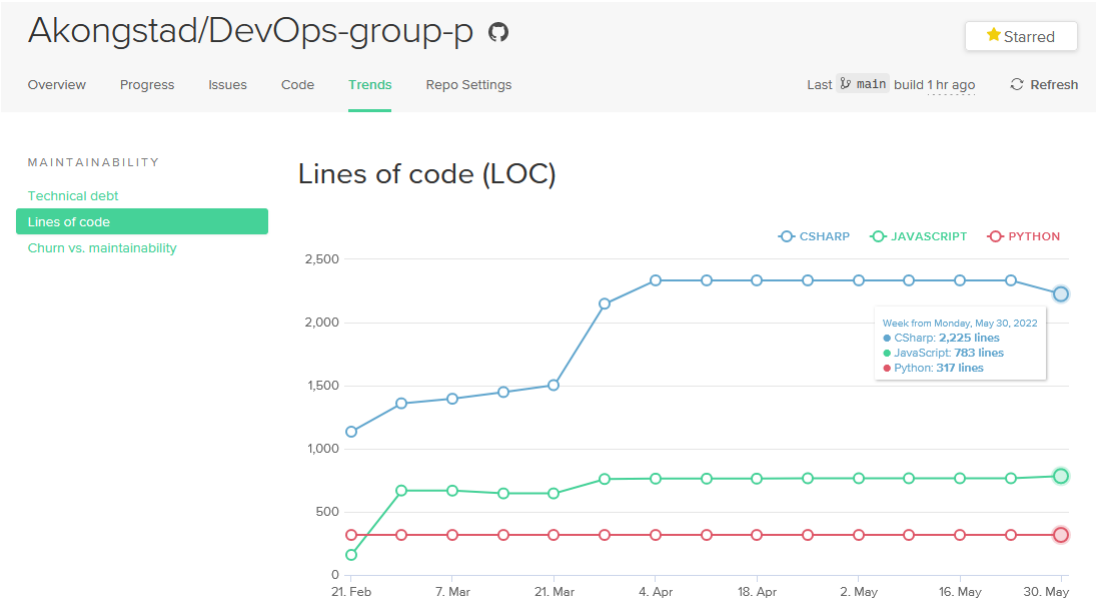

Figure 16: Code Climate language distribution graph 31/05/22

A.3.3. Better Code Hub

https://bettercodehub.com/results/Akongstad/DevOps-group-p

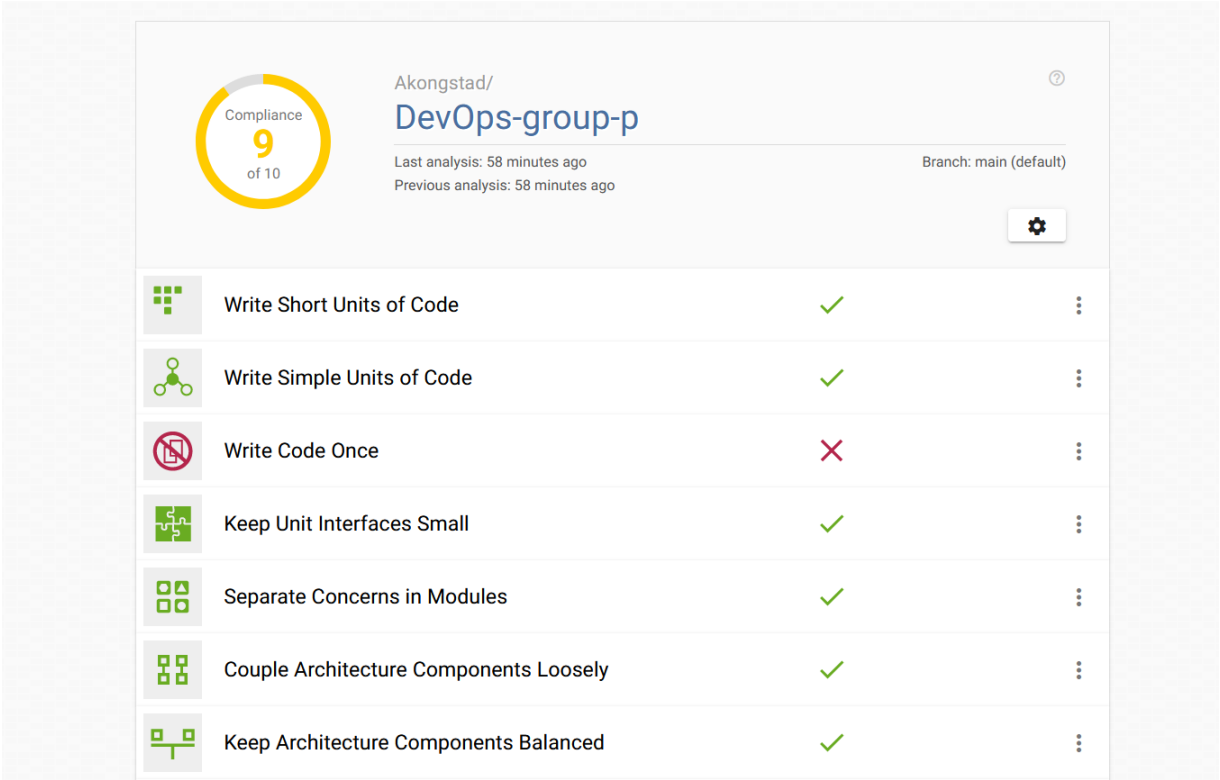

Figure 17: Better Code Hub repository status 30/05/22

A.3.4. DeepScan

https://deepscan.io/dashboard/#view=project&tid=17220&pid=20572&bid=562975

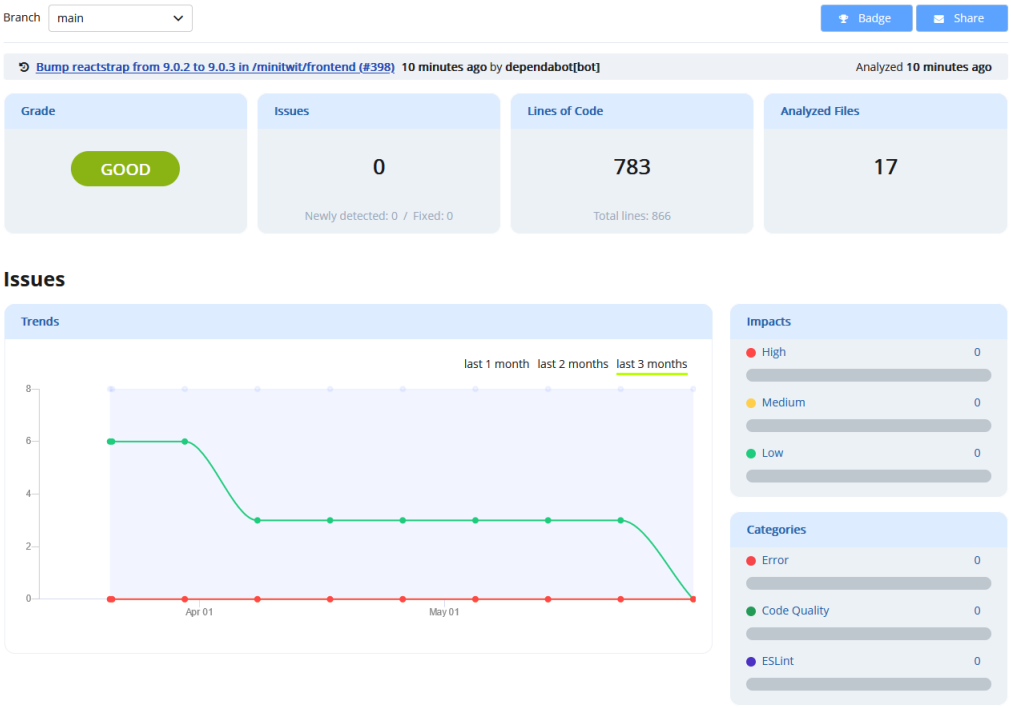

Figure 18: DeepScan dashboard 31/05/22

A.3.5. snyk

https://app.snyk.io/org/akongstad

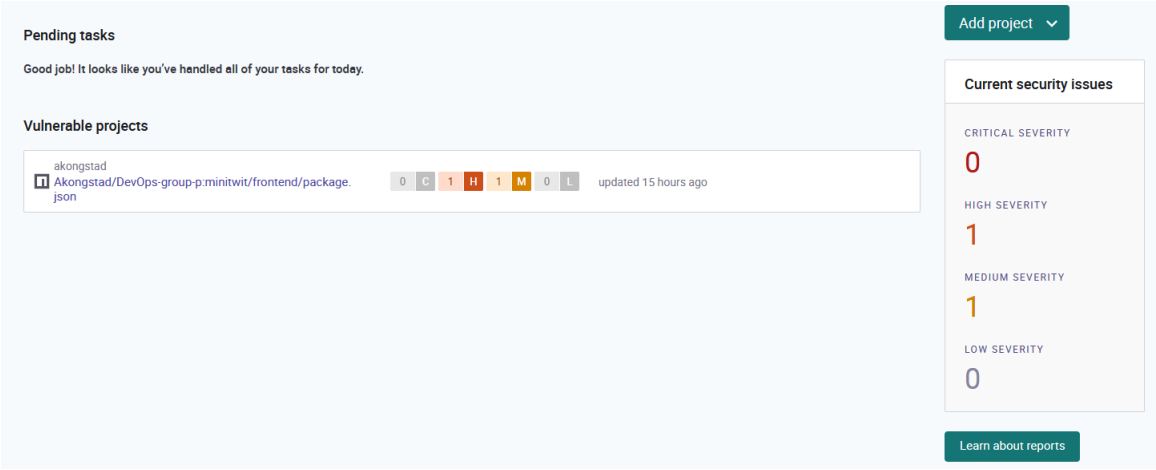

Figure 19: Snyk Dashboard 31/05/2022

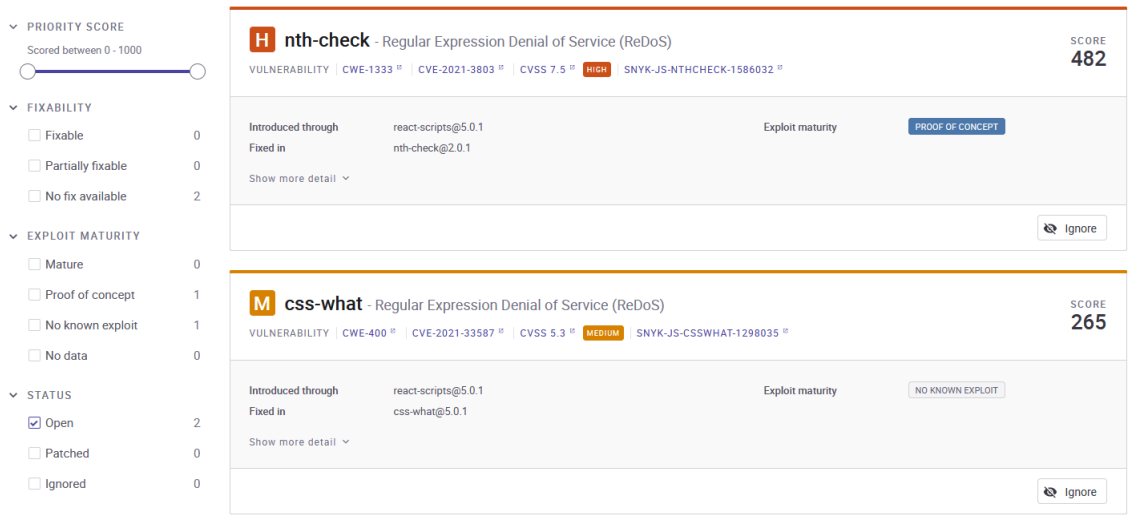

Figure 20: Snyk scan alert 31/05/2022

A.3.6. Github - Dependabot

https://github.com/Akongstad/DevOps-group-p/security/dependabot

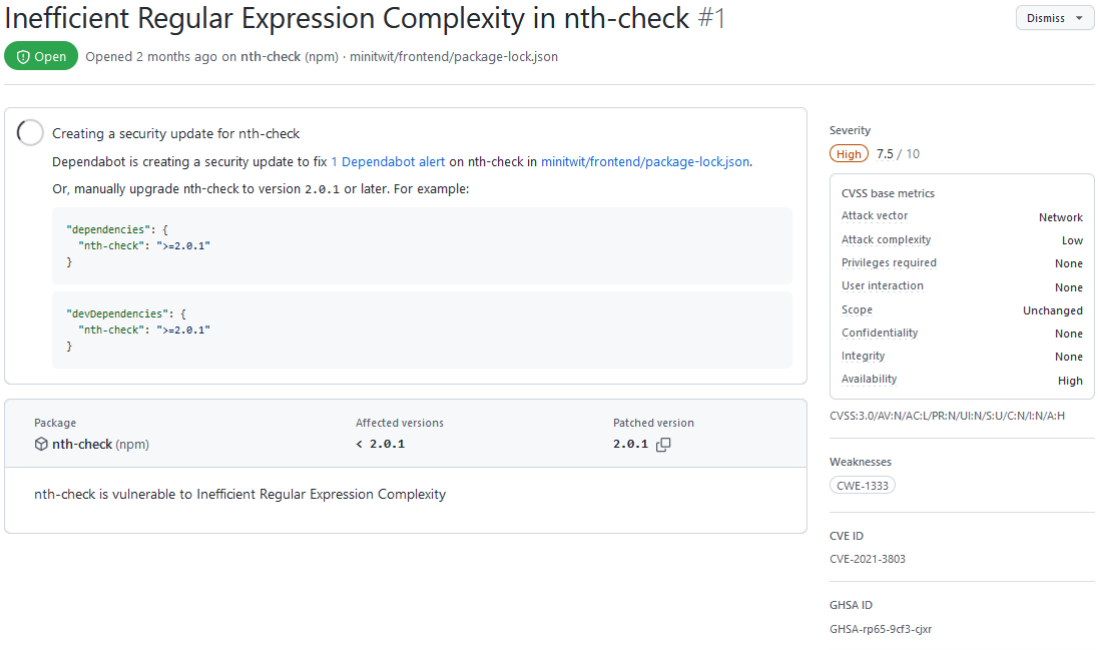

Figure 21: Dependabot scan results 31/05/2022

A.3.7. Github - Code scanning tools

https://github.com/Akongstad/DevOps-group-p/security/code-scanning

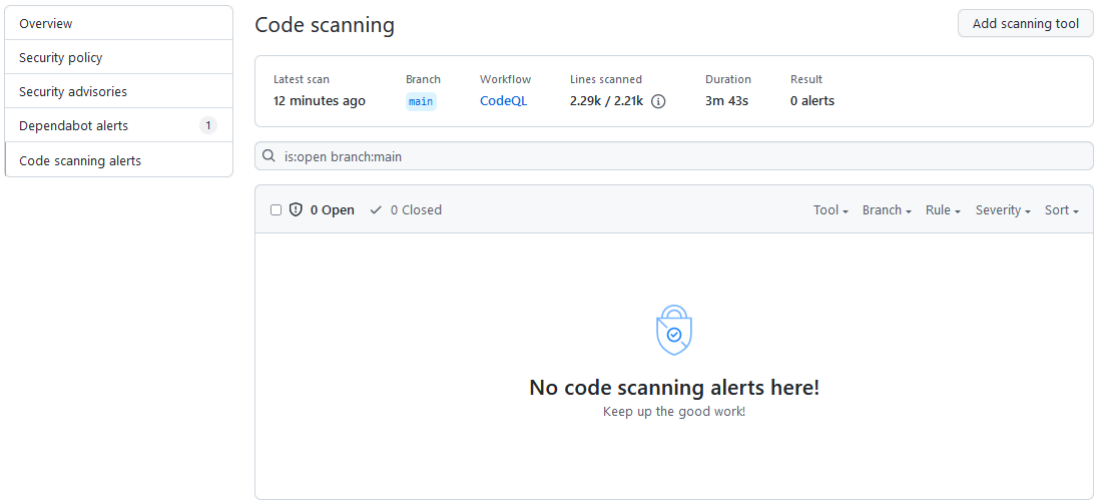

Figure 22: CodeQl scan results 31/05/2022