Chirp! Project Report

ITU BDSA 2023 Group 18

Benjamin von Barner Altenburg beal@itu.dk
David Alexander Feldner dafe@itu.dk
Jack Kryger Sørensen jkrs@itu.dk
Mads Valentin Jensen mvje@itu.dk
Severin Ernst Bøgelund Madsen sevm@itu.dk

1 Design and Architecture of *Chirp!*

1.1 Domain model

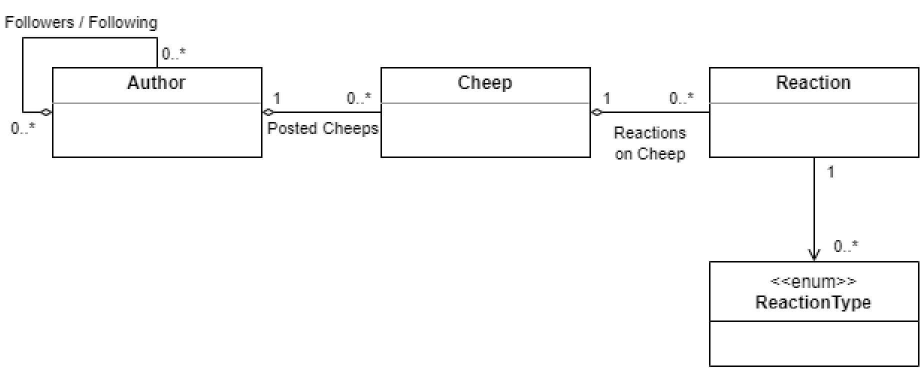

Figure 1: Class diagram showing the types used in the application and stored in the database.

1.2 Architecture — In the small

1.3 Architecture of deployed application

Our application is hosted on Azures App Service where the component Chirp.Web handles the GUI and Chirp.Infrastructure handles both the repositories and database model. A user connects their machine to Chirp.Web through the Azure App Service. Reading or writing data will make Chirp.Web read from the repositories in Chirp.Infrastructure, that will then call the Azure SQL Server database. To log in, Chirp.Web calls GitHub Authentication.

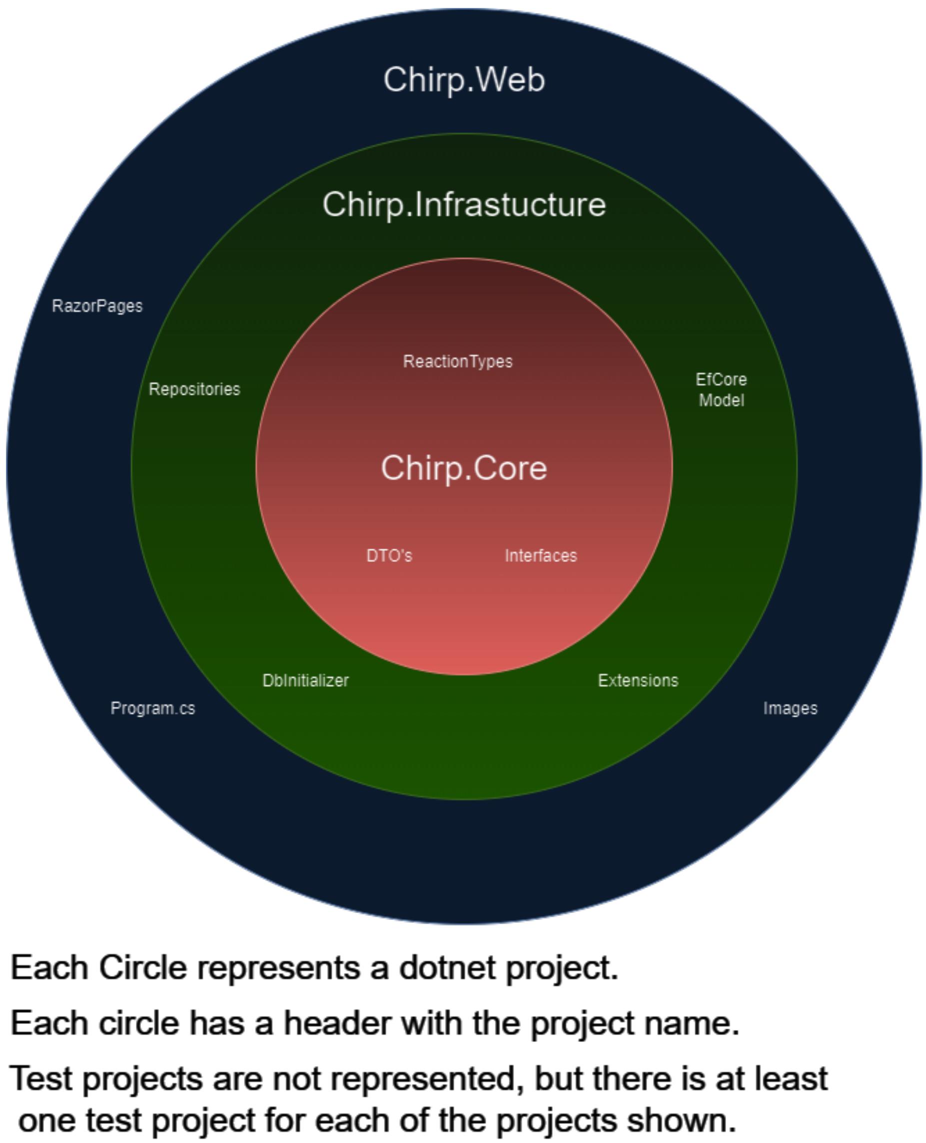

Figure 2: Diagram showing our onion architecture.

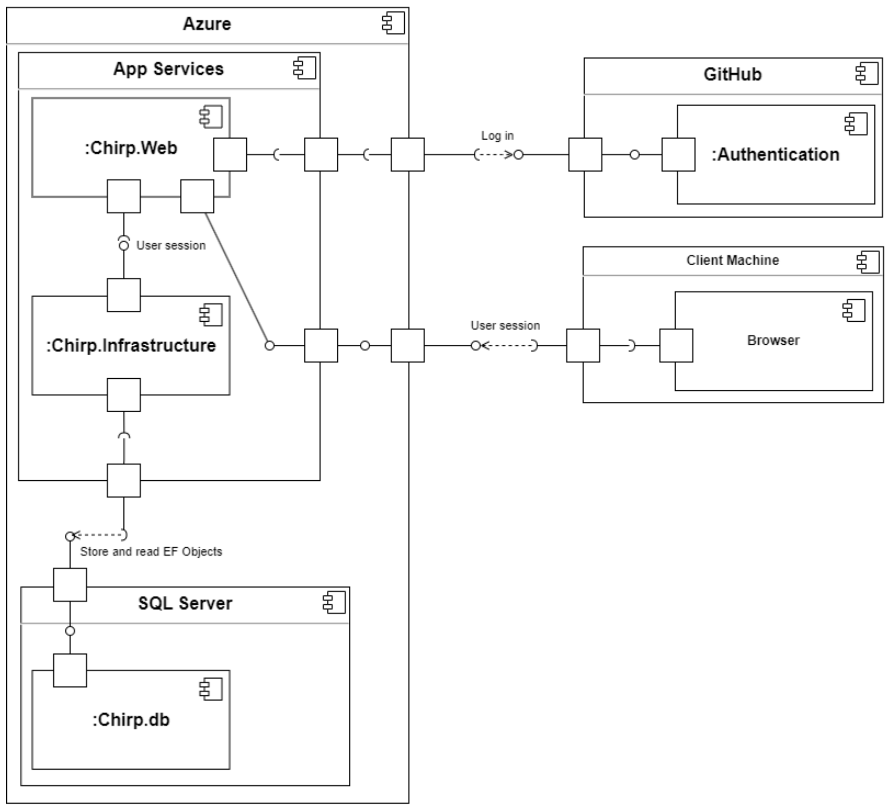

Figure 3: Component diagram showing the architecture of the deployed Chirp application.

1.4 User activities

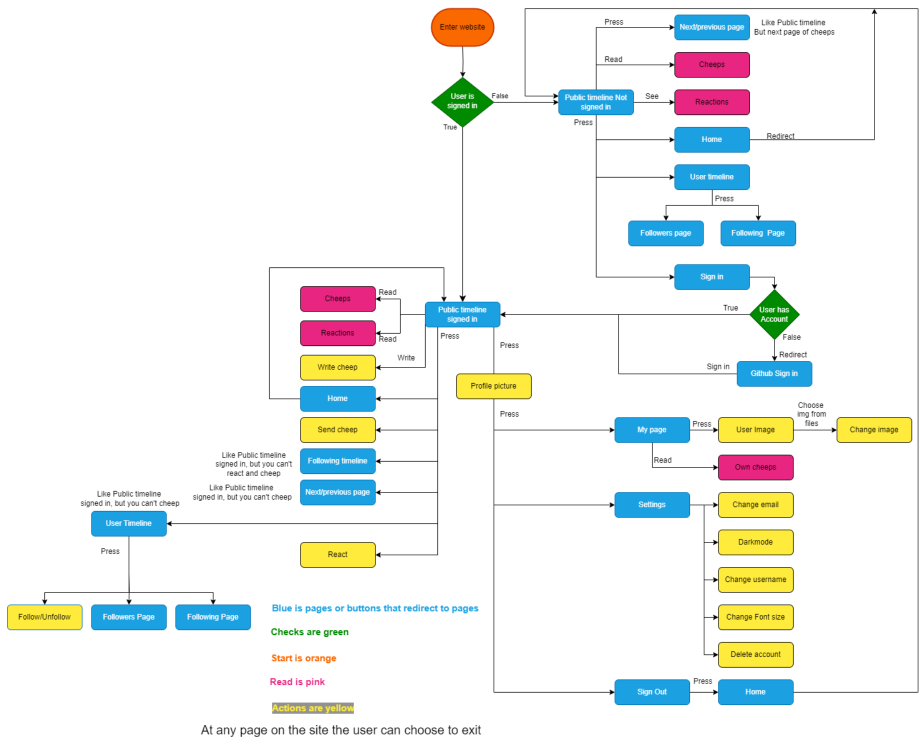

Figure 4: Diagram showing all possible user activities from any given action.

1.5 Sequence of functionality/calls through *Chirp!*

2 Process

2.1 Build, test, release, and deployment

We use 3 GitHub workflows to manage automatic building, testing, releasing, and deployment of our application.

2.1.0.1 Test workflow The first workflow is our test workflow, which runs on every push to any branch, and on changes to a pull-request. The workflow runs through all of our tests and ensures that they always pass.

2.1.0.2 Release workflow Our second workflow is the release workflow which runs on every closed pull request to the main branch and firstly checks that the pull request was merged and not just closed. The workflow then runs our tests in the same way as our test workflow. It then tries to increment the

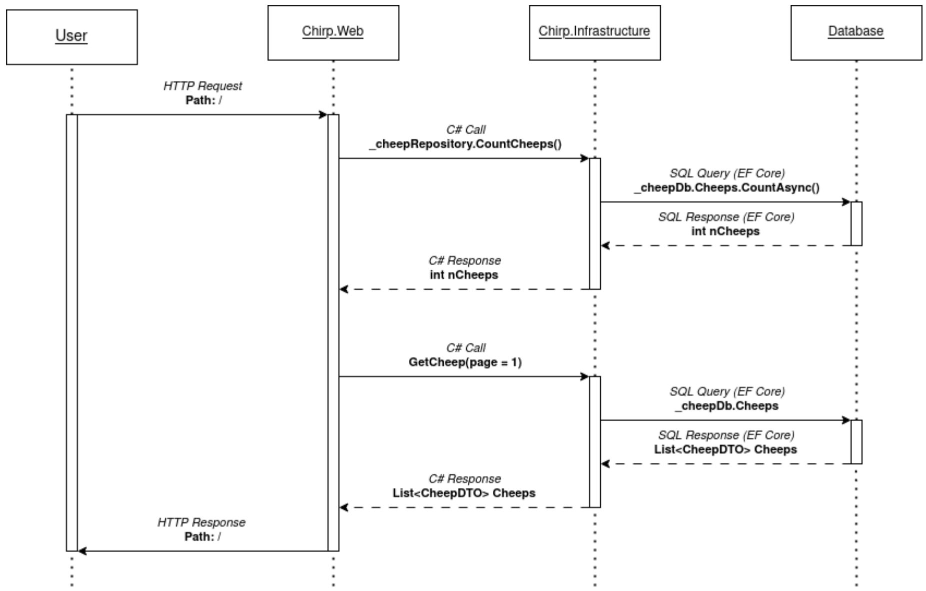

Figure 5: UML Sequence Diagram of calls through Chirp

version tag for the release by getting the latest tag from GitHub and using labels on the pull request to determine the importance of the pull request.

After setting the correct tag it goes through each platform we want to release to, where it publishes and creates a zip file with all the correct files. After creating zip files for all platforms, it releases them all to GitHub in a new release with the previously set tag.

2.1.0.3 Deployment workflow Lastly our deployment workflow is very similar to the auto-generated deployment workflow from Azure. It runs when there is a push to the main branch, and similarly to the other workflows, it sets up and tests our code. It then publishes and uploads the program artifacts such that the next job can download them and push them to our Azure web app.

2.2 Teamwork

2.2.1 Project board

All issues in the project backlog are solved/closed, which includes all the required features and additional features.

2.2.1.1 Future Issue: optimize profile-picture handling In this version of Chirp, Author profile pictures are stored as a reference string to an image stored locally in the `wwwroot/images` folder. This means that you have to push

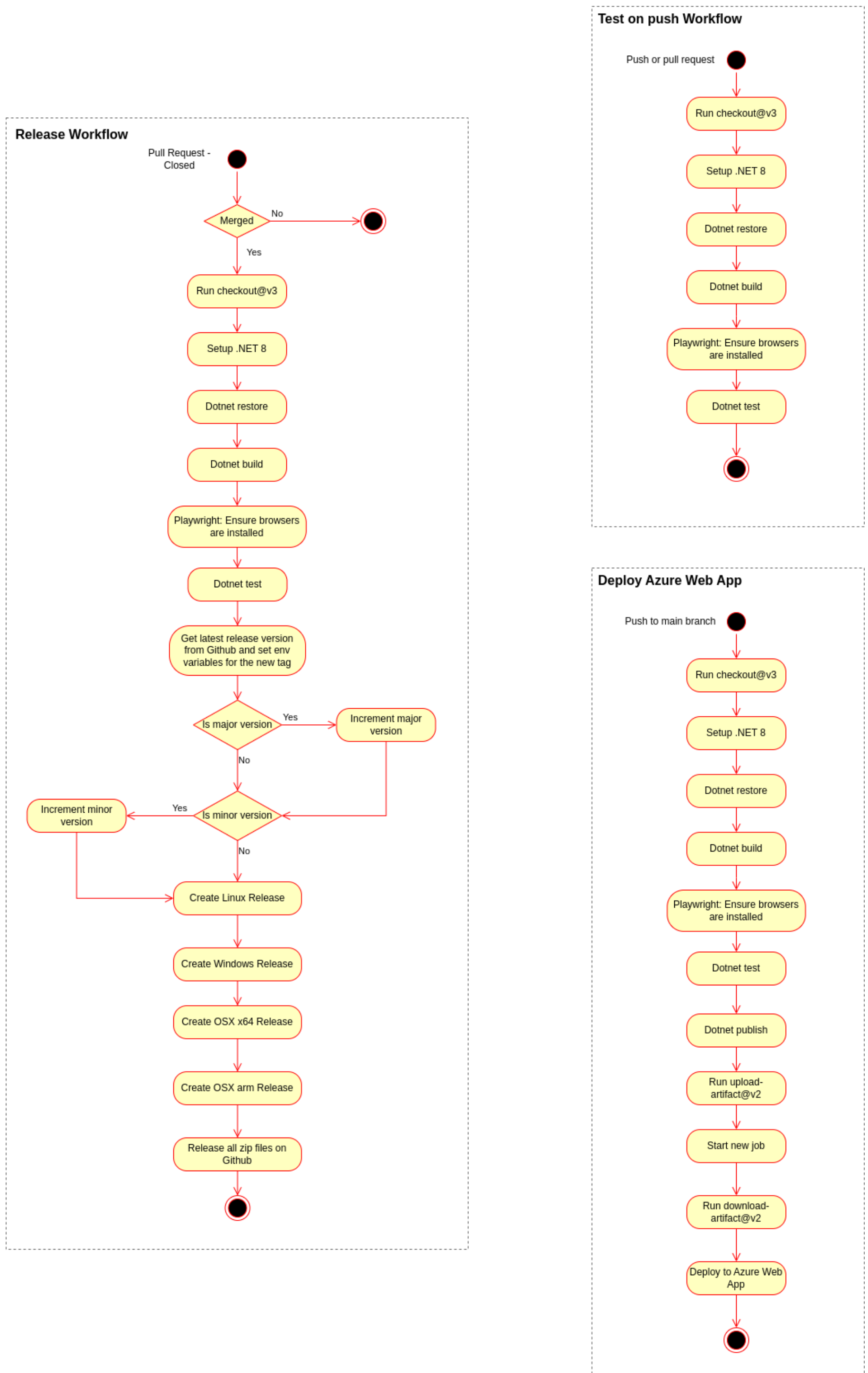

Figure 6: UML Activity diagram of GitHub Action Workflows.

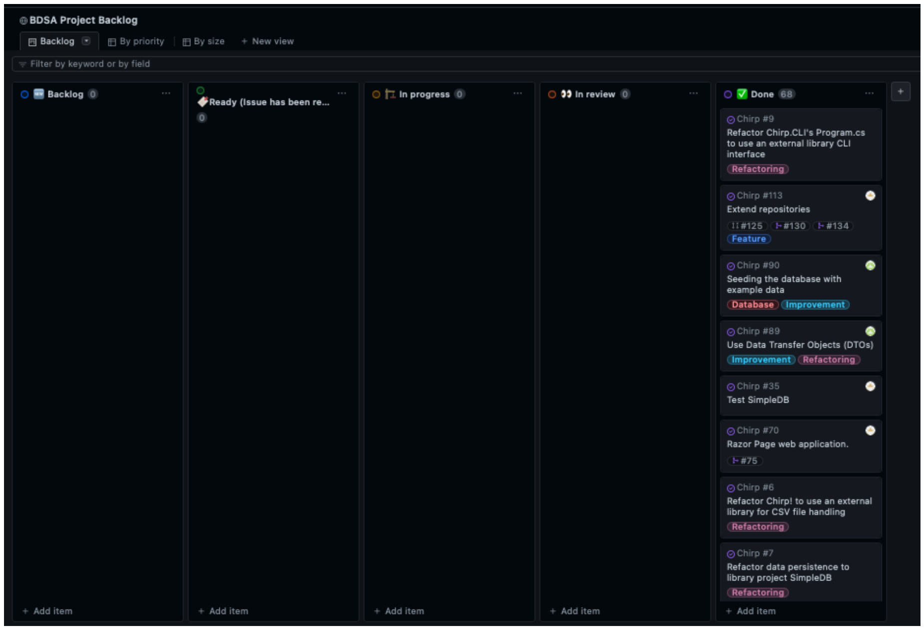

Figure 7: Project oard of our issues.

your image to GitHub each time you change profile picture. We have therefore discussed that if we had more unpaid space in the database we would store the whole image under each Author, hence have a more scalable and secure solution.

2.2.2 From issue to main

The following flowchart illustrates how group 18 creates issues (green box), develops solutions (red box), and merges the solutions into the `main` branch (purple box). The flowchart also shows how different decisions/statements are made through the flow, such as "Is the task understood".

2.3 How to make *Chirp!* work locally

OBS! .Net 8 is required for running Chirp locally. Run the following commands:

```
Git clone https://github.com/ITU-BDSA23-GROUP18/Chirp.git
```

Go to the directory of the cloned repository (via `cd Chirp`) and run:

```
dotnet watch --project src/Chirp.Web
```

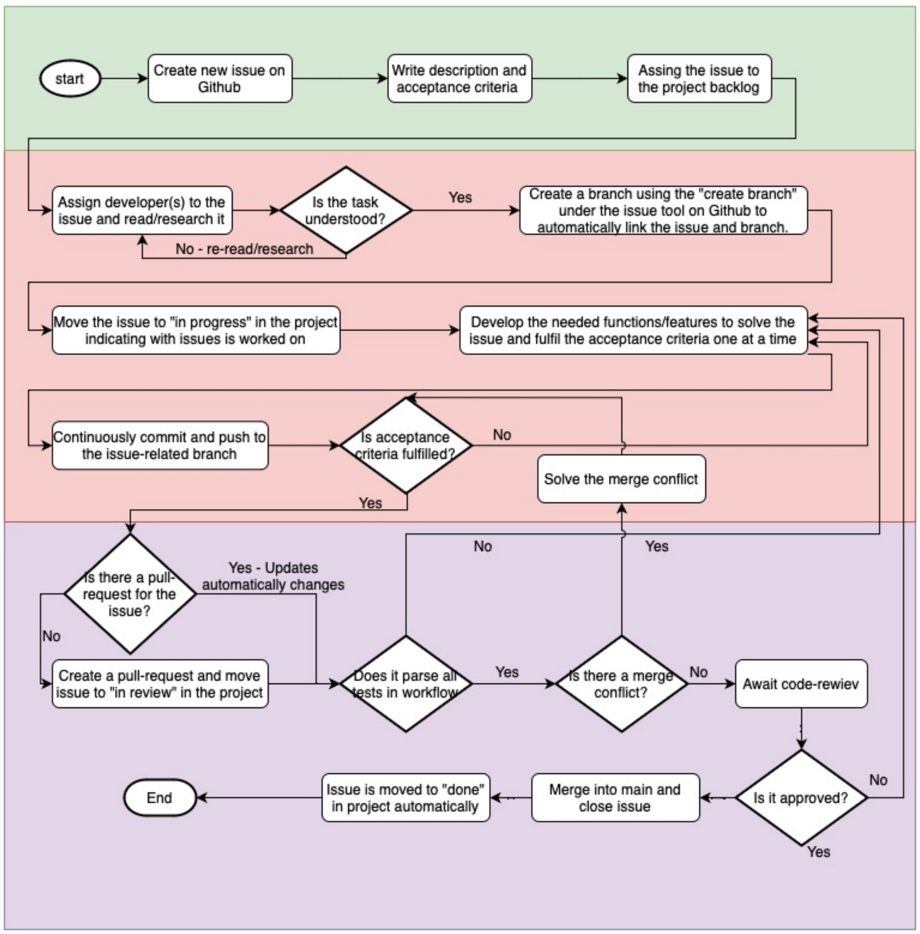

Figure 8: FlowChart From Issue To main.

2.4 How to run the test suite locally

2.4.1 Our tests

We are testing our systems functionalities using; **End-to-end tests, Integration tests, and Unit tests** for the different projects in our onion architecture.

2.4.1.1 Chirp.Infrastructure.Tests The tests for Chirp.Infrastructure, test all the different repositories. This is done using a combination of unit and integration tests.

2.4.1.2 Chirp.Web.Tests/Chirp.Web.Ui.Tests We use playwright for our UI tests, which functions as our end-to-end tests. The tests go through our different features acting as a user would, assessing if the features work as intended. We also have some unit tests, testing the most basic things.

2.4.1.3 Chirp.Core.Tests For the Chirp.Core we have some simple tests for the DTOs ensuring that their parameters can't be null.

2.4.2 How to run the test

In order for the ui test to run, you have to install Powershell if you are on Linux.

2.4.2.1 Windows and Linux

```
cd test/Chirp.Web.Ui.Tests
pwsh bin/Debug/netX/playwright.ps1 install
```

2.4.2.2 macOS (Apple) *We haven't found a way to successfully run the playwright test on macOS (Apple). If you want to look at the result of the tests, look on GitHub.*

2.4.3 Running the tests

Go to the root of the project and run.

```
dotnet test
```

3 Ethics

3.1 License

- The MIT license

3.2 LLMs, ChatGPT, CoPilot, and others

3.2.1 CoPilot and ChatGPT

CoPilot and ChatGPT were both used for:

- Code generation and auto-completion.
- Debugging and understanding errors.

For writing code, both LLMs were only helpful to a minor degree, for increasing the development speed and code readability when knowing what to prompt or via CoPilot auto-completion. However, in many cases it was faster to read the documentation and manually implement the code, especially when not knowing exactly what to prompt. Mainly, it was ChatGPT that had such cases as it only relies on the prompt and has no code base knowledge. CoPilot's auto-completion also suggested old and incorrect code a few times, once again making it faster to manually write the code.

Both LLMs were a bit more helpful when debugging, as they in many cases were able to quickly give suggestions to fix the issue and an explanation of the error without having to read through many long error stacks containing confusing commands and methods.

3.2.2 CodeFactor

CodeFactor was used on each pull request, automatically checking the cleanliness and readability of the pushed code. If CodeFactor found any unclean or irregular code in the pull request, it would either issue or apply fixes to the code.