

# Chirp! Project Report

Edward Rostomian      Thorbjørn Peter Høgsbro Pedersen  
Daniel Holm Larsen      Halfdan Eg Minegar Brage  
Nicklas Ostenfeldt Gardil

December 20, 2023

# Table of Contents

## Contents

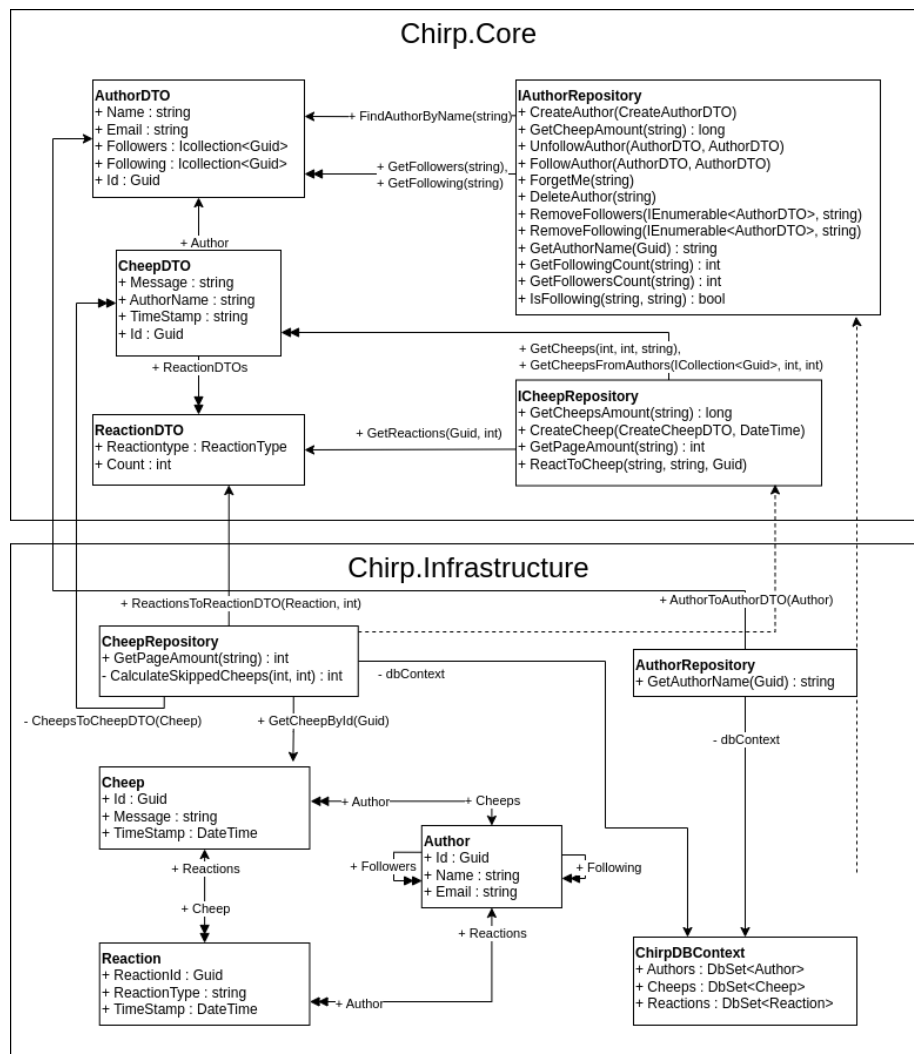
<b>Table of Contents</b>	<b>2</b>
<b>1. Introduction</b>	<b>3</b>
<b>2. Design and Architecture of Chirp!</b>	<b>3</b>
Domain model . . . . .	3
Architecture — In the small . . . . .	4
Architecture of deployed application . . . . .	5
User activities . . . . .	5
Sequence of functionality/calls through Chirp! . . . . .	6
<b>3. Process</b>	<b>9</b>
Build, test, release, and deployment . . . . .	9
Merge to main workflow . . . . .	9
Teamwork . . . . .	10
How to make Chirp! work locally . . . . .	12
How to run the test suite locally . . . . .	12
<b>4. Ethics</b>	<b>13</b>
License . . . . .	13
LLMs, ChatGPT, CoPilot, and others . . . . .	13

# 1. Introduction

In this report we will briefly describe the project work and outcome of our social media application, Chirp. The project is developed for the course “Analysis, Design and Software Architecture (Autumn 2023)”, taught by Helge Pfeiffer and Rasmus Lystrom, at the IT University of Copenhagen.

## 2. Design and Architecture of Chirp!

### Domain model



In our program the user post messages in the form of a cheep. the Cheep class is a model, it represents what a cheep is. A cheep consists of an id, Author, message, TimeStamp, and a list of Reactions

The author class represents a user of our application. it contains all the information the program needs to model a user.

The reaction class is used to keep track of the different reactions a user can have given a cheep. It contains the reaction type, the author, and the cheep that has been reacted to.

We have repositories for author and cheep. These repositories contain the methods to manipulate and retrieve data in/from the database. The repositories are used in the outer part of the onion. We use Data transfer objects to send and receive data between the different layers of our program. The DTOs contain the same information as the classes but they are not used as entity classes for the model. So they are safer when dealing with the user, to make sure the user can't change the database in an unwanted way.

## Architecture — In the small

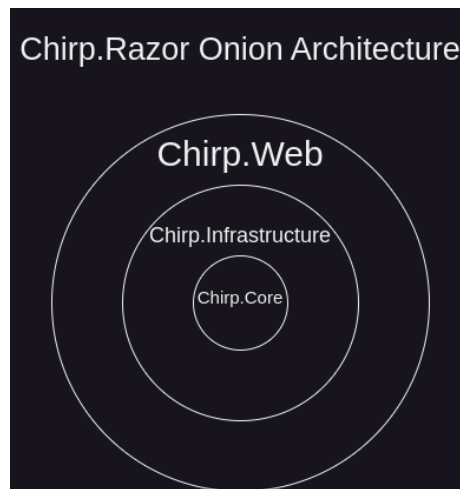


Figure 1: Onion model

Our chirp application is implemented with an “onion skin architecture”. This means that our program is divided into three layers, core, infrastructure and web. The three layers follow a hierarchical structure where core < infrastructure < web. In this comparison, only greater layers may use or know the contents of the lower layers. Following this structure should result in reusable and loosely coupled code.

In a company setting, code from “core” could be reused in many different

applications and contexts around the entire company. In our project, core only contains DTO's and interfaces that are used throughout our entire project. Chirp.Infrastructure contains all our domain implementations. This means that our repositories, domain classes (Cheep, Author, Reaction) are located here. Both our database migrations and our database-context (dbContext) are additionally located in infrastructure. Chirp.Web contains all our frontline code, in the shape of cshtml files, and their corresponding cshtml.cs code. Chirp.Web is the main executable c# project, which means that the Program.cs file is located here. Additionally, a database initialization script is also located here, which can populate and initiate our database with data provided by the course.

## **Architecture of deployed application**

Our application is a web-application, hosted by Azure. Clients use our web application through http calls. Our application sends and receives data from and to our Azure SQL server database. If a user tries to access a page, that requires authentication, they are redirected to authentication, through Business to Consumer (B2C). Authentication is done through their GitHub account. Next, they are redirected back to our page. If already authenticated, a cookie is saved, and they can skip the login process.

## **User activities**

The navigation bar is shown on all pages, and is used to redirect the user to other pages.

### **Not authenticated:**

When accessing our webpage, users are presented with the public timeline, which displays cheeps. On this page, users can navigate between pages to view older or newer cheeps. The navigation bar consists of links to "Public Timeline" and "login". Furthermore, users have the option to click on the author's name within cheeps, redirecting them to the author's private timeline, showing cheeps made by that author. Also, user can click on the login button, which facilitates authentication through B2C, using their GitHub account. If already logged in to Github on their browser, they are directed to the Public Timeline. If not, they must login with a Github account.

### **Authenticated:**

The navigation bar is changed upon user authentication. It has links to pages such as "My Timeline", "Public Timeline", "For You", "About Me" and "Logout." The navigation bar is visible on all pages. On every page where there are cheeps, the user is able to express reactions, and follow/unfollow authors of all cheeps, not made by themselves. On the public timeline, they are also able to submit cheeps and sign out. They can also react to cheeps and follow/unfollow authors on cheeps, if not they are the author of the cheep themselves. On "my timeline",

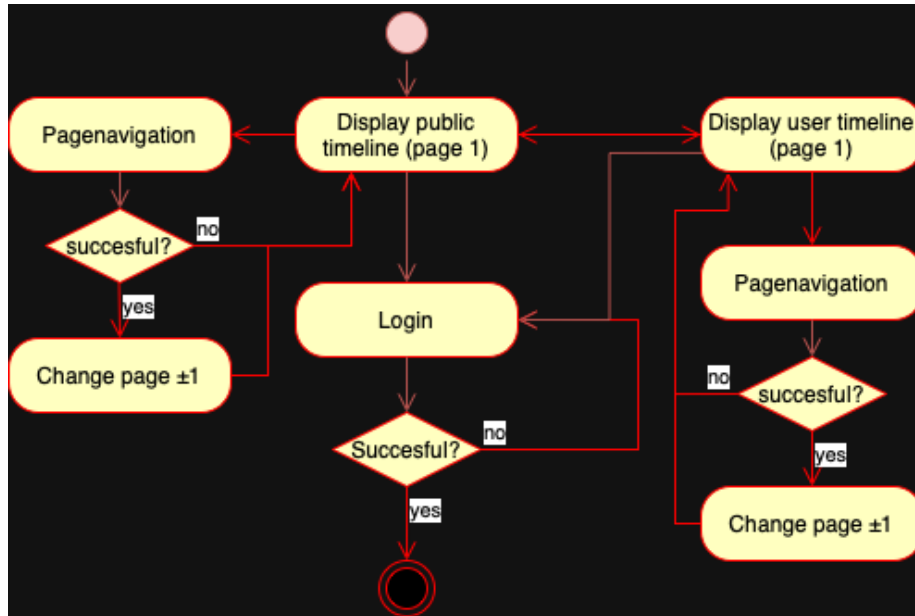


Figure 2: User Activity diagram: not authenticated

the user can submit cheeps, and see their own cheeps. On “For you”, they can see the cheeps of the people that they follow. On the “About me” page, they can see the users they follow, the people who follow them, the number of each, and their own most recent cheeps. They can press the “Forget me” button, which deletes everything about them, from the database. They can also go to the timeline of other users, by pressing their name, found on one of the lists.

### Sequence of functionality/calls through Chirp!

When a user access the website they make a http get requested. If they do it to a page which they are not authorized to then the program makes a authgorize code request + code challenge to Azure AD B2C to try and Authenticate the user. Azure B2C then sends a Authorization code request to Github Where the user can authorize with github to login. If the user is successful at github, then it returns a authorization code to B2C and B2C get a token from github with the code. B2C then return a authorization code to the Client. The client can get authorazation id and token from B2C. When the user then has login and are granted authorozation to the page then the server returns the web-page and the client can render it.

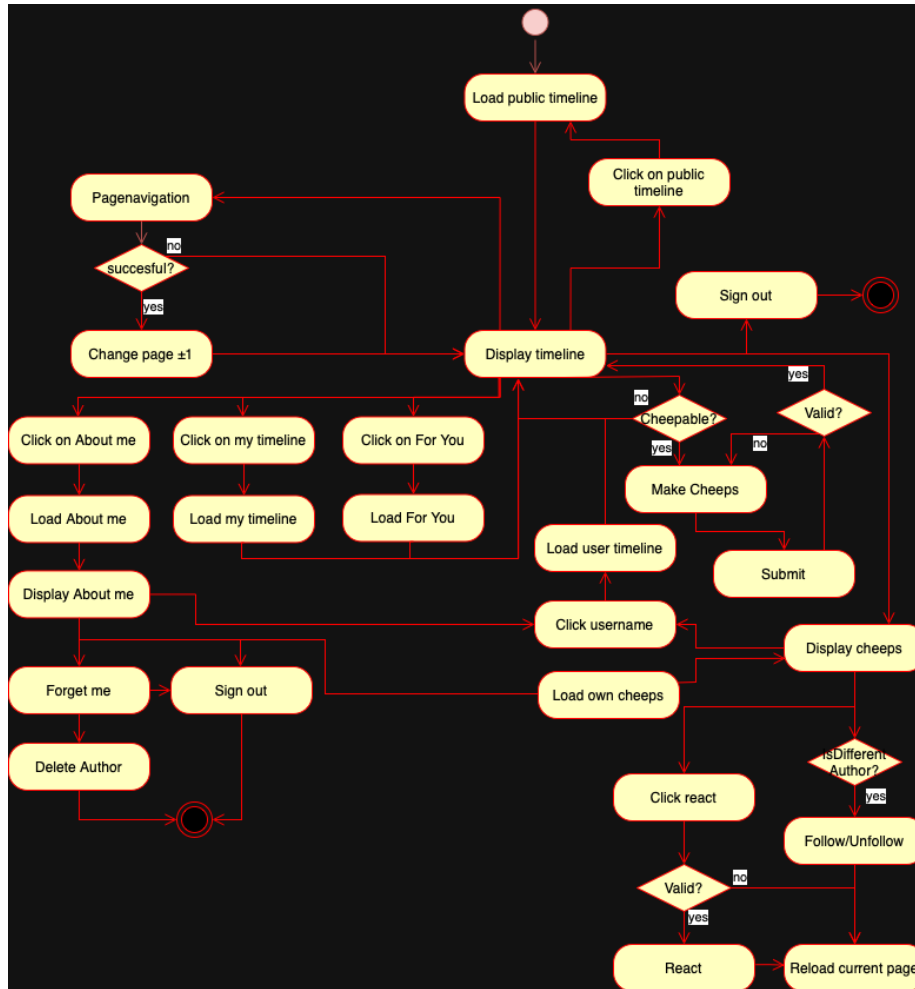


Figure 3: User Activity diagram: Authenticated

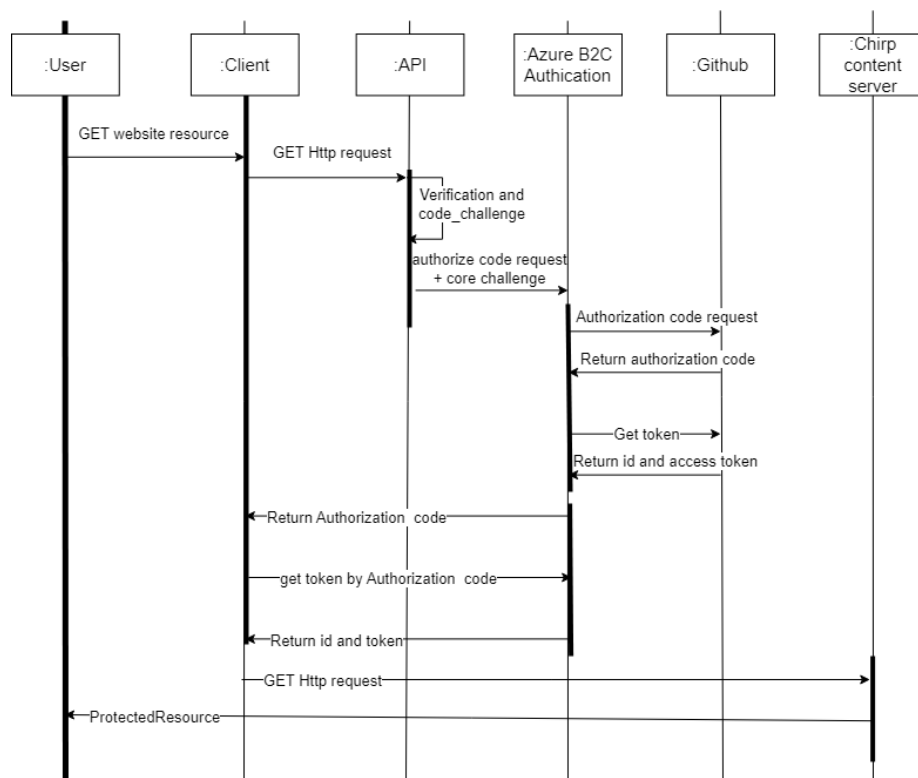


Figure 4: Sequence Diagram



### 3. Process

#### Build, test, release, and deployment

##### Merge to main workflow

During our project development process, our main method of building, testing and deploying was with two automated workflows. The structure of which is described by the diagram below:

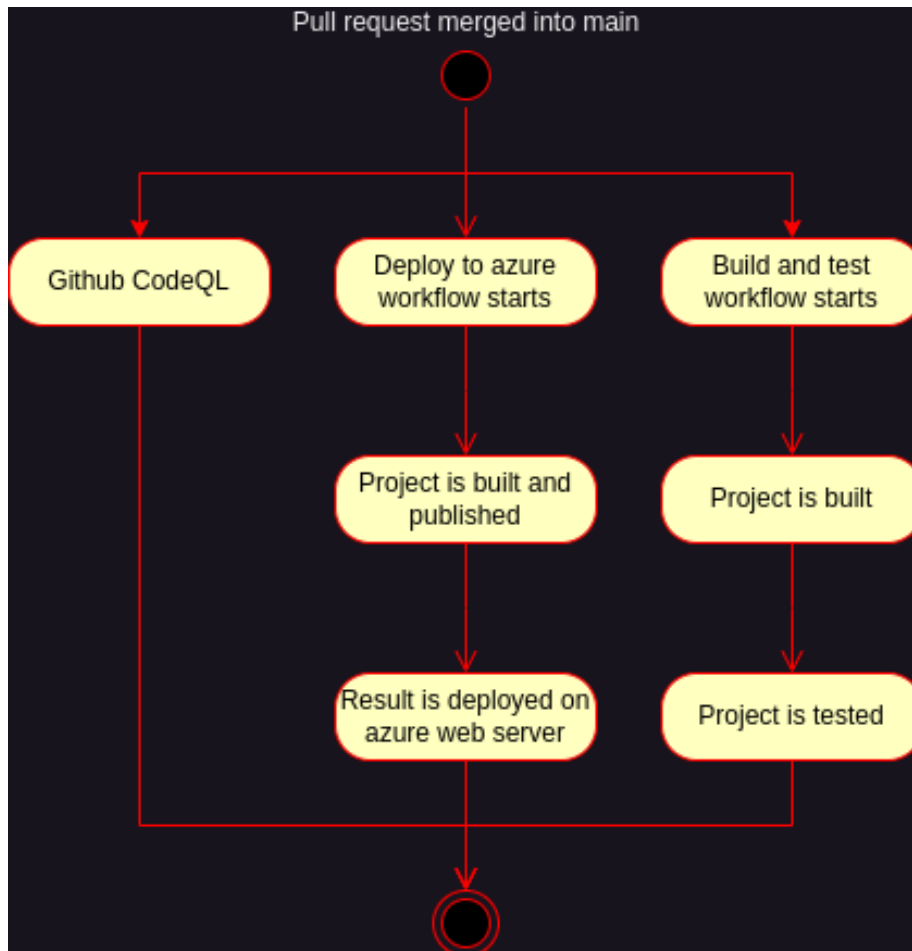


Figure 5: Merge to main workflows

In the diagram you can see our two main workflows, the “deploy to azure” workflow (filename `main_bdsagroup23chirprazor.yml`) and the build and test workflow (filename `dotnet.yml`). Github CodeQL is also included in the diagram as it is an automated process, which runs whenever we merge into main. The

“deploy to azure” workflow is auto-generated by azure and slightly modified. It publishes our application and uploads it to our azure web application. This was our preferred method of automated deployment. Our build and test workflow builds our project, and runs our tests (not including ui tests). The two workflows and CodeQL run in parallel. This means that our web app would be deployed even if our test suite failed. This was practical in our case, for rapid development, since our tests in many cases were not updated to work with our newer code. This meant that we could test our code on the live server without updating all our tests first. Not testing before deploying also meant that the deployment process was quicker. Going around our tests suite would in a real-world scenario, with an active application, result in huge stability issues.

For “automatic releases” we used a separate workflow (filename publish.yml). The process of which can be seen in the diagram below:

The functionality of this workflow is to build, test and package our application automatically, whenever a release is created.

We aimed to introduce singlefile releases, but prioritized new features and other requirements, delaying its implementation. The infrequent releases resulted from both postponing until singlefile capability, and a lack of defined milestones for stable functionality. Insufficient release planning, and constant development on important features contributed to this pattern. Since different features were almost always under development, we rarely felt our program was in a stable, shippable state.

Back when we were developing Chirp.CLI, we had a more solid release schedule. This is because it was the primary distribution of the software. When the project transitioned into a Razor application, the primary distribution became our Azure Web App, and our releases became way less frequent. Releases of our Razor application would also be quite difficult to use (since it requires docker), and would lack all online functionality. So for an ordinary user, there would be absolutely no reason to run our code from releases.

## Teamwork

For this project, we made most of our work, while sitting together in a meeting, either physically, or on a discord server. Then we would split off into smaller groups, but still be available for other team members.

On Github, we have used a centralised workflow, with protection on the main branch, and a requirement for pull requests to make changes.

When creating a new issue, we focus on the functional requirements, and make sure to make an issue, that covers these. Sometimes, we already do the design process here, and describe in the issue, more precisely, how to reach the functional requirements. Other times, we let the one(s) who work on the issue, make all design decisions. For large issues, or very important design decisions, we often



Figure 6: Release workflow  
11

discuss it in the group, even with team members who aren't assigned to the issue.

When we work on a feature, we are usually one or two people. Sometimes we use pair programming. Other times one will work on the frontend, while the other works on the backend. Once we believe a feature is ready for main, we make a pull request, and ask a group member who hasn't been a part of this issue, to review it. Depending on the complexity of the code, we ask one or more people to review it. Sometimes we explain the code to the reviewer(s). Sometimes we find that some of the code could be better, or maybe that some of the changes were unnecessary or too intrusive, and should be reverted. Depending on how big of an issue it is, and how much time we have, we either write a comment, and possibly an issue about fixing it, and then approve the pull request, or we write a comment, and request changes, before allowing for a push to main.

## How to make Chirp! work locally

Prerequisites: Microsoft .Net 7.0 and Docker

To make Chirp! work locally, first you must clone the repository:

```
git clone https://github.com/ITU-BDSA23-GROUP23/Chirp.git
```

On windows or osx, make sure that the docker desktop application is running first. On linux systems, ensure the Docker daemon is running. It can be started with:

```
sudo dockerd
```

From here, you must first start a MSSQL docker container using the following command:

```
sudo docker run -e "ACCEPT_EULA=Y" -e "MSSQL_SA_PASSWORD=DhE883cb" \
  -p 1433:1433 --name sql1 --hostname sql1 \
  -d \
  mcr.microsoft.com/mssql/server:2022-latest
```

Next, from the root directory in /Chirp, run the following command:

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

Alternatively, from the /Chirp.Web folder:

```
dotnet run
```

Finally, open your browser of choice and connect to <https://localhost:7040>

## How to run the test suite locally

To run the test suites locally, first you will have to start your docker container.

MAC:

```
docker run -e "ACCEPT_EULA=Y" -e "MSSQL_SA_PASSWORD=DhE883cb" -p 1433:1433 --name sql1 --hos
```

Windows: Open Docker desktop and run the `mcr.microsoft.com/mssql/server:2022-latest` image

Linux/wsl:

```
sudo docker run -e "ACCEPT_EULA=Y" -e "MSSQL_SA_PASSWORD=DhE883cb" \  
  -p 1433:1433 --name sql1 --hostname sql1 \  
  -d \  
  mcr.microsoft.com/mssql/server:2022-latest
```

Next, open up a terminal in the project. Assuming you are in the root of the repository Chirp, direct to either:

```
cd Test/Chirp.Razor.Tests
```

or

```
cd Test/UITest/PlaywrightTests
```

In both the `Chirp.Razor.Tests` and `PlaywrightTests` folder, to run the tests:

```
dotnet test
```

The project contains two test suites, `Chirp.Razor.Tests` and `UITest`. The first test suite contains unit tests, integration tests and end-to-end tests. **\*\*Har vi det?** The unit tests are testing the functionality of the isolated components in our application, that is testing methods within our application of core, infrastructure and web components. The integration tests are testing the interactions of different components in our application, that is testing when using logic from e.g. the infrastructure layer in our web components. The end-to-end tests...?

The second test suite contains our UI tests. These are UI automation tests, using Playwright to simulate a users interactions with the user interface. These are implemented such that we can ensure that the UI behaves as expected, performing actions and receiving expected output, when doing all types of interactions with our application from the UI. Before be able to run the test the program has to be running on the same local machine.

## 4. Ethics

### License

License: WTFPL

### LLMs, ChatGPT, CoPilot, and others

The LLMs used for this project during developments are ChatGPT and GitHub CoPilot. ChatGPT has been used carefully, mainly for asking questions about the code or errors in the code. It has also been used for generating small pieces of

code, mainly in the cshtml files. Likewise, CoPilot has been used for generating some of the code in cshtml, but has also been used for helping with code, partly making some of the methods in the repositories and creating outlines for tests. Generally, the responses of the LLMs has been helpful for better understanding of the code and speeding up the development. It has not really created code that we would not have done ourselves, but it has provided some logic in the methods, which has been helpful in terms of taking inspiration for further method extensions. The application of LLMs has sped up the development process. Especially, CoPilot has made coding much faster, as it for most parts provides the code needed, e.g., if we already made a test for a method FollowAuthor, in no time CoPilot can make the same one for UnfollowAuthor. However, there has indeed been a few times, when ChatGPT or CoPilot does not understand the requests as intended, and therefore not providing useful outputs. But, for most of the time, they have been helpful tools for development.