Chirp! Project Report

ITU BDSA 2025 Group 6

Andreas Schiøtt `anys@itu.dk`
Maja Schewe-Jensen `mcls@itu.dk`
Sebastian Holt Svendsen `sesv@itu.dk`
Ludvig Kirkeby `luki@itu.dk`
Anna Bohn Hoff `akih@itu.dk`
Nina Hansson `cenh@itu.dk`

Git Repository:
`https://github.com/ITU-BDSA2025-GROUP6/Chirp.git`

1 Design and Architecture of *Chirp!*

1.1 Domain model

Chirp has the following key entities:

```
- Author: Enables user management through extending ASP.NETs IdentityUser
- Cheep: Represents posts by authors using timestamps and text.
- Follows: Tracks following, and being followed by, other authors.
- Recheep: Allows reposting specific cheeps by other authors on our own timeline.
```

Below is a UML class diagram of our domain model:

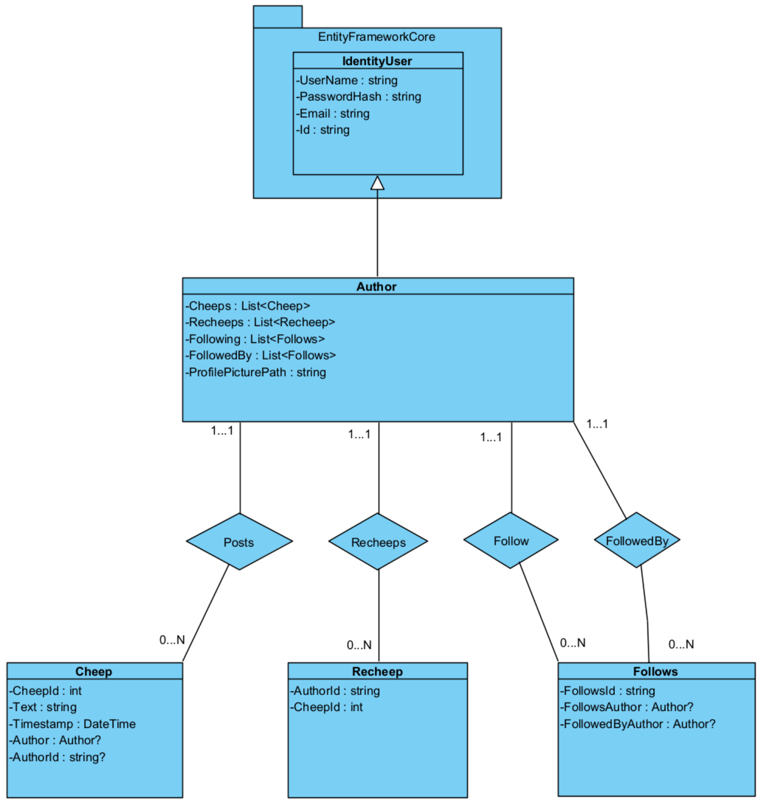


Figure 1: DomainModel

- **NOTE:** `IdentityUser` only shows the first part of the library from which it comes from in the diagram. The full library path is: `Microsoft.AspNetCore.Identity.EntityFrameworkCore.IdentityUser`
- **NOTE:** While not shown in the diagram, all three classes connected to **Author** (`Cheep`, `Recheep`, and `Follows`) use `System.ComponentModel.DataAnnotations` to support `required` parameters, among others.

1.2 Architecture — In the small

The Onion Architecture of Chirp is seen in the below UML.

- **UI Layer:** Outer layer seen by the Client. Includes Pages, Identity Core scaffolded Pages, Startup code through Program.cs.
- **Service Layer:** Data flow between our repository, the UI layer and the Client.
- **Repository Layer:** Functionality, DTOs, Interfaces and database retrieval methods supported by Entity Core.
- **Domain Layer:** Contains our domain entities only.

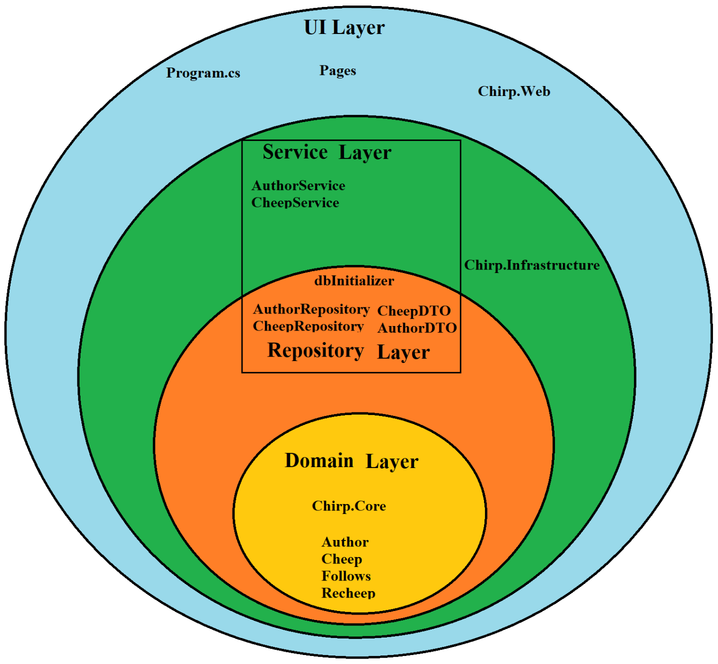


Figure 2: Onion Architecture

1.3 Architecture of deployed application

The Deployed Application Architecture can be seen below.

Remote Architecture Client interacts with an Azure Database through an HTTP (Converted to HTTPS) request. Hosted offshore.

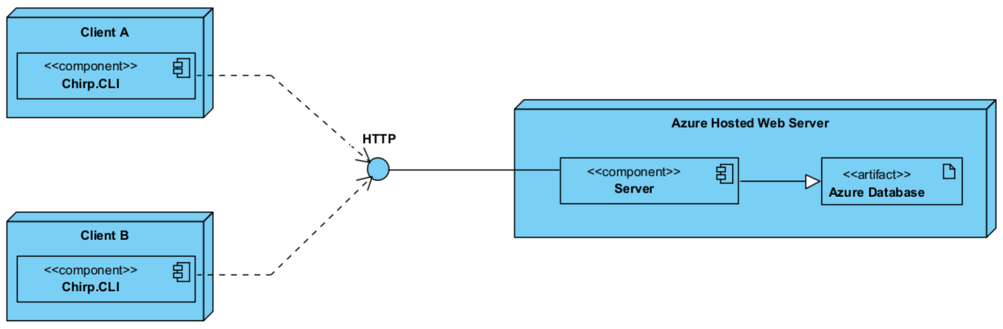


Figure 3: Deployed Application Azure Architecture

Local Architecture Client interacts directly with a local database. Does not require internet.

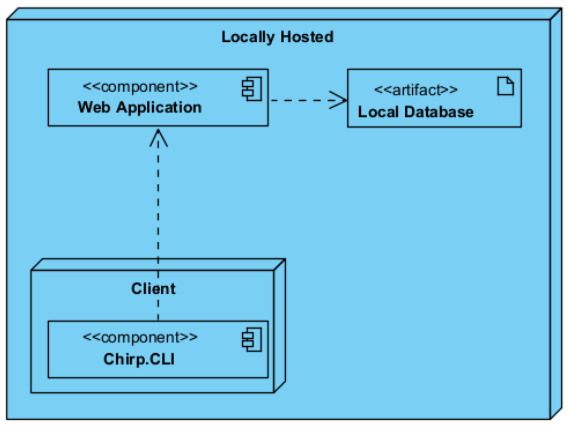


Figure 4: Deployed Application Locally Hosted Architecture

1.4 User Activities

We have two types of users: `Authorized` and `Unauthorized`.

- `Unauthorized User` can:
  - Register and log in
  - View public and private timelines
  - Navigate multiple pages of content
- `Authorized User` can:
  - Log out
  - View public and private timelines
  - View their own timeline with posted Cheeps
  - Navigate multiple pages of content
  - Access an *About Me* page with account information and account deletion (*Forget Me*)
  - Set a profile picture in the "About me" page
  - Post new Cheeps
  - Recheep other Authors' Cheeps

1.4.1 Unauthorized User Diagram

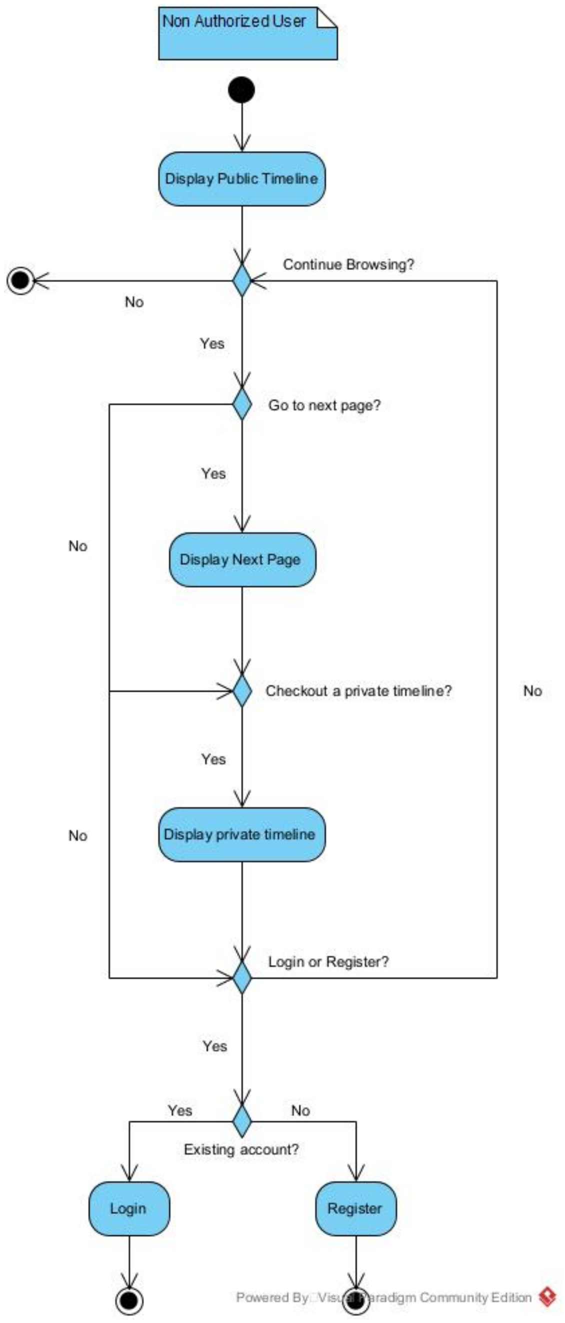


Figure 5: Unauthorized User Activity Diagram

1.4.2 Authorized User Diagram

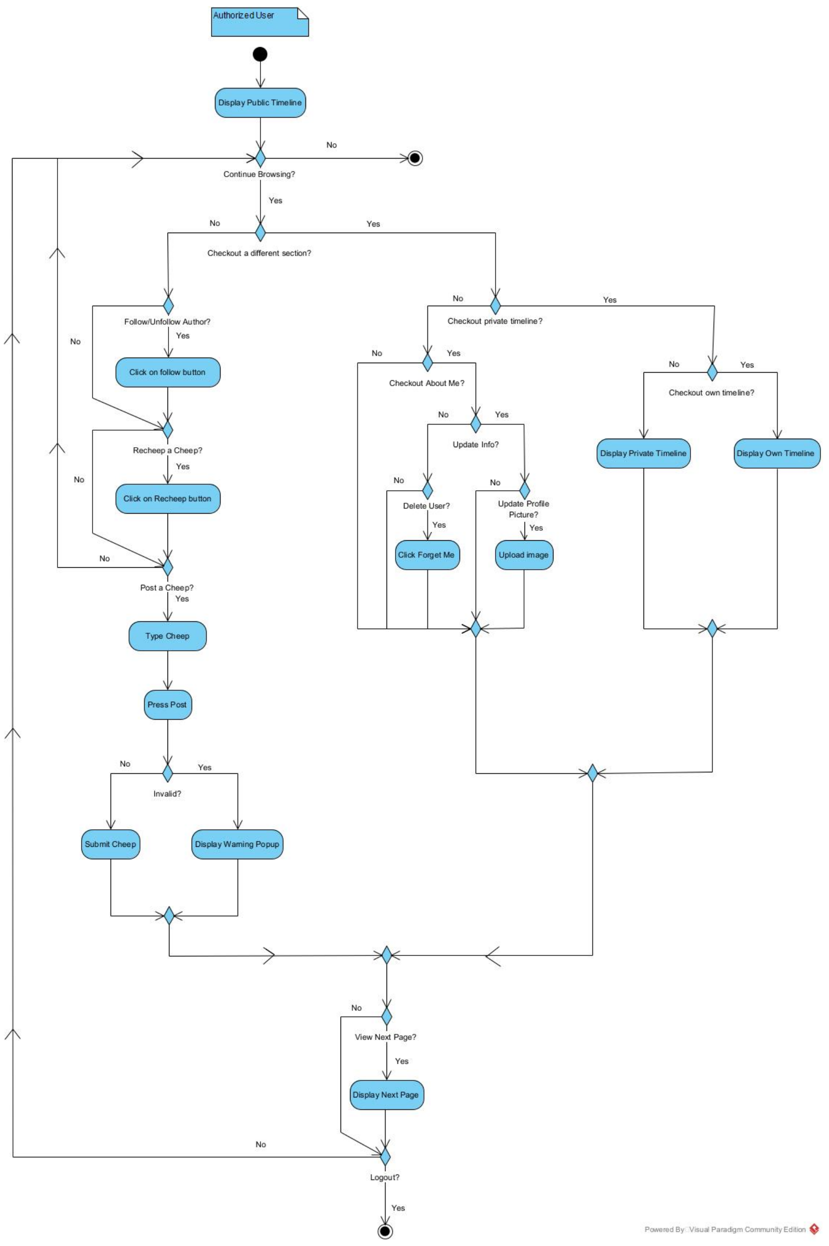


Figure 6: Authorized User Activity Diagram

1.5 User Activities - Expanded Login Diagrams

1.5.1 Standard Login (Typing in details on the Registration or Login page)

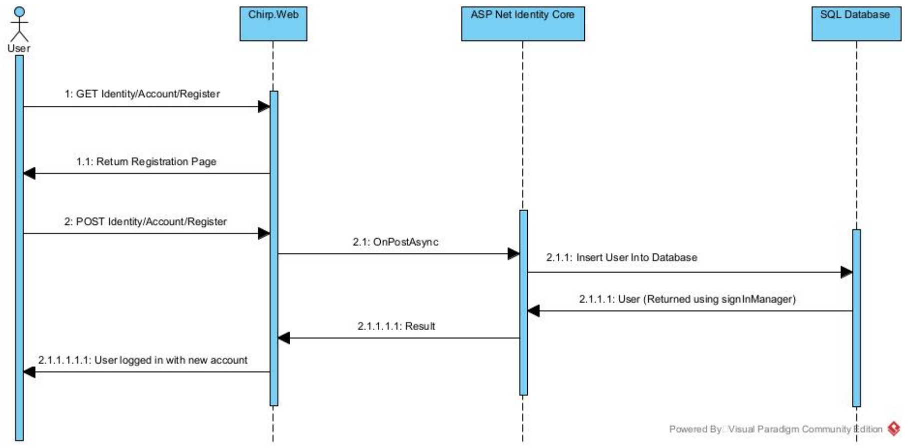

User
Chirp.Web
ASP Net Identity Core
SQL Database
1: GET Identity/Account/Register
1.1: Return Registration Page
2: POST Identity/Account/Register
2.1: OnPostAsync
2.1.1: Insert User Into Database
2.1.1.1: User (Returned using signInManager)
2.1.1.1.1: Result
2.1.1.1.1.1: User logged in with new account
Powered By: Visual Paradigm Community Edition

Figure 7: Authentication Diagram

1.5.2 Open Authentication Login

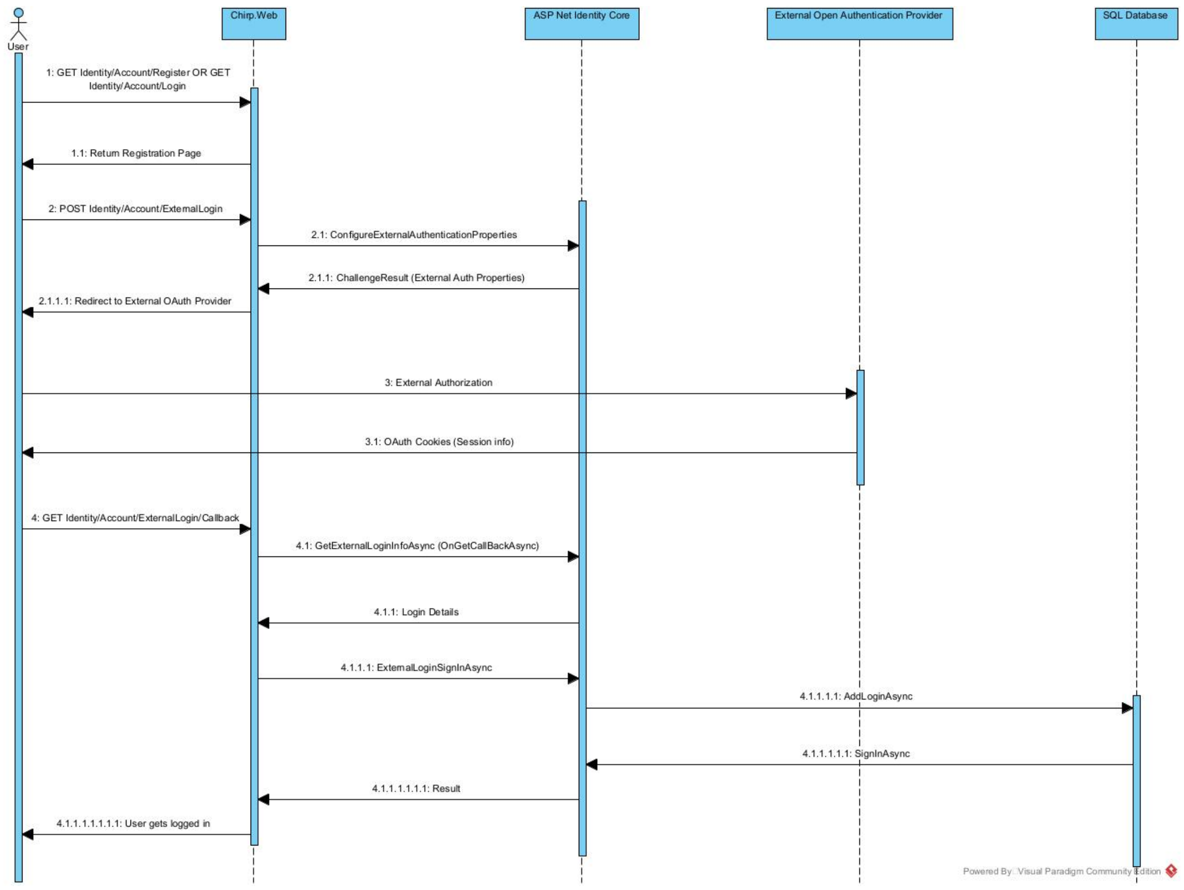

User
Chirp.Web
ASP Net Identity Core
External Open Authentication Provider
SQL Database
1: GET Identity/Account/Register OR GET Identity/Account/Login
1.1: Return Registration Page
2: POST Identity/Account/ExternalLogin
2.1: ConfigureExternalAuthenticationProperties
2.1.1: ChallengeResult (External Auth Properties)
2.1.1.1: Redirect to External OAuth Provider
3: External Authorization
3.1: OAuth Cookies (Session info)
4: GET Identity/Account/ExternalLogin/Callback
4.1: GetExternalLoginInfoAsync (OnGetCallBackAsync)
4.1.1: Login Details
4.1.1.1: ExternalLoginSignInAsync
4.1.1.1.1: AddLoginAsync
4.1.1.1.1.1: SignInAsync
4.1.1.1.1.1.1: Result
4.1.1.1.1.1.1.1: User gets logged in
Powered By: Visual Paradigm Community Edition

Figure 8: Open Authentication Diagram

1.6 Sequence of functionality/calls through *Chirp!*

Below is a Diagram of an example functionality call of an Unauthorized user accessing the site, causing us to display all cheeps, which are grabbed from the database.

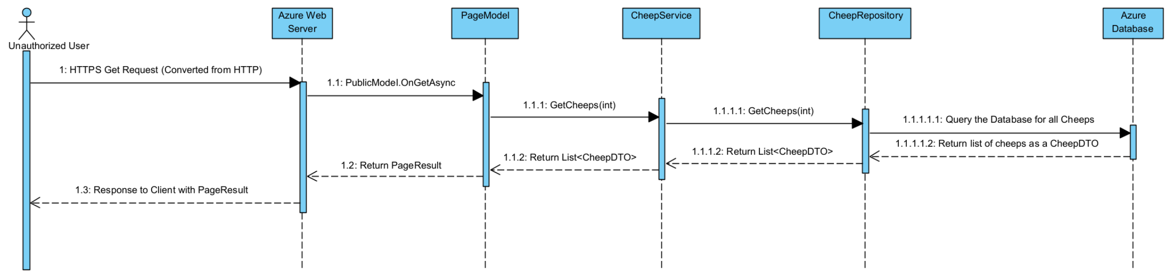

Unauthorized User
Azure Web Server
PageModel
CheepService
CheepRepository
Azure Database
1: HTTPS Get Request (Converted from HTTP)
1.1: PublicModel.OnGetAsync
1.1.1: GetCheeps(int)
1.1.1.1: GetCheeps(int)
1.1.1.1.1: Query the Database for all Cheeps
1.1.1.1.2: Return list of cheeps as a CheepDTO
1.1.1.2: Return List<CheepDTO>
1.1.2: Return List<CheepDTO>
1.2: Return PageResult
1.3: Response to Client with PageResult

Figure 9: Example functionality call of an Unauthorized user

2 Process

2.1 Build, test, release, and deployment

We have created four workflows for the different tasks:

- **CI:** Automatically builds and tests the application on every change, and publishes a versioned release artifact when a release tag is pushed.
- **CD:** Build and deploy to Azure Web App.
- **Executables:** Builds, tests, and publishes single-file executables for multiple operating systems when a GitHub release is created.
- **Auto Label:** Automatically labels issues based on keywords.

The first diagram is over our Continuous Integration workflow, which ensures build- and test correctness. In addition it publishes release web app artifact on tags, with both title and release notes.

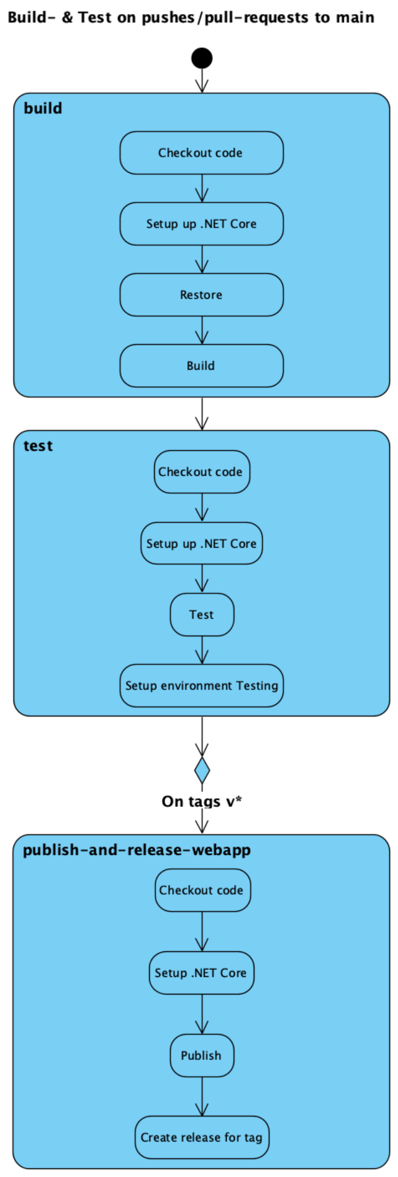


Figure 10: CI Diagram

The second diagram illustrates the Continuous Deployment workflow, which consists of a build- and deploy job. This workflow is triggered on ‘push’ to the main branch. The workflow ensures that the ASP.NET application is built, published, and deployed on Azure App Service. The workflow does not include a dedicated test step, as this is done in CI.

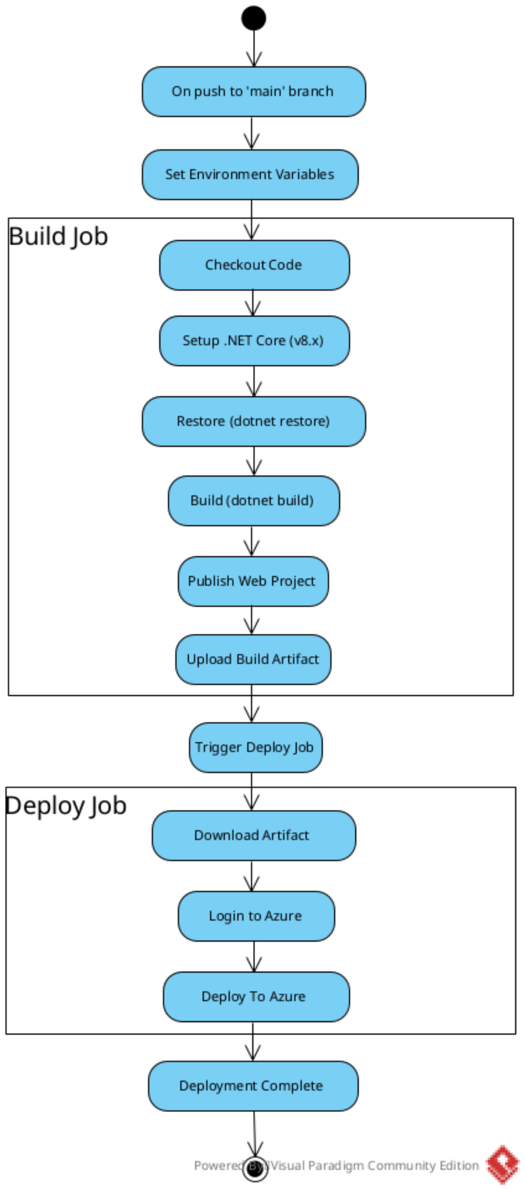


Figure 11: CD Diagram

The third diagram is for releasing executables. When a version tag and release is created, the workflow builds, tests, and publishes four platform-specific executables (Linux, Windows, and macOS for both x64 and arm64), which are packaged and uploaded to the GitHub release.

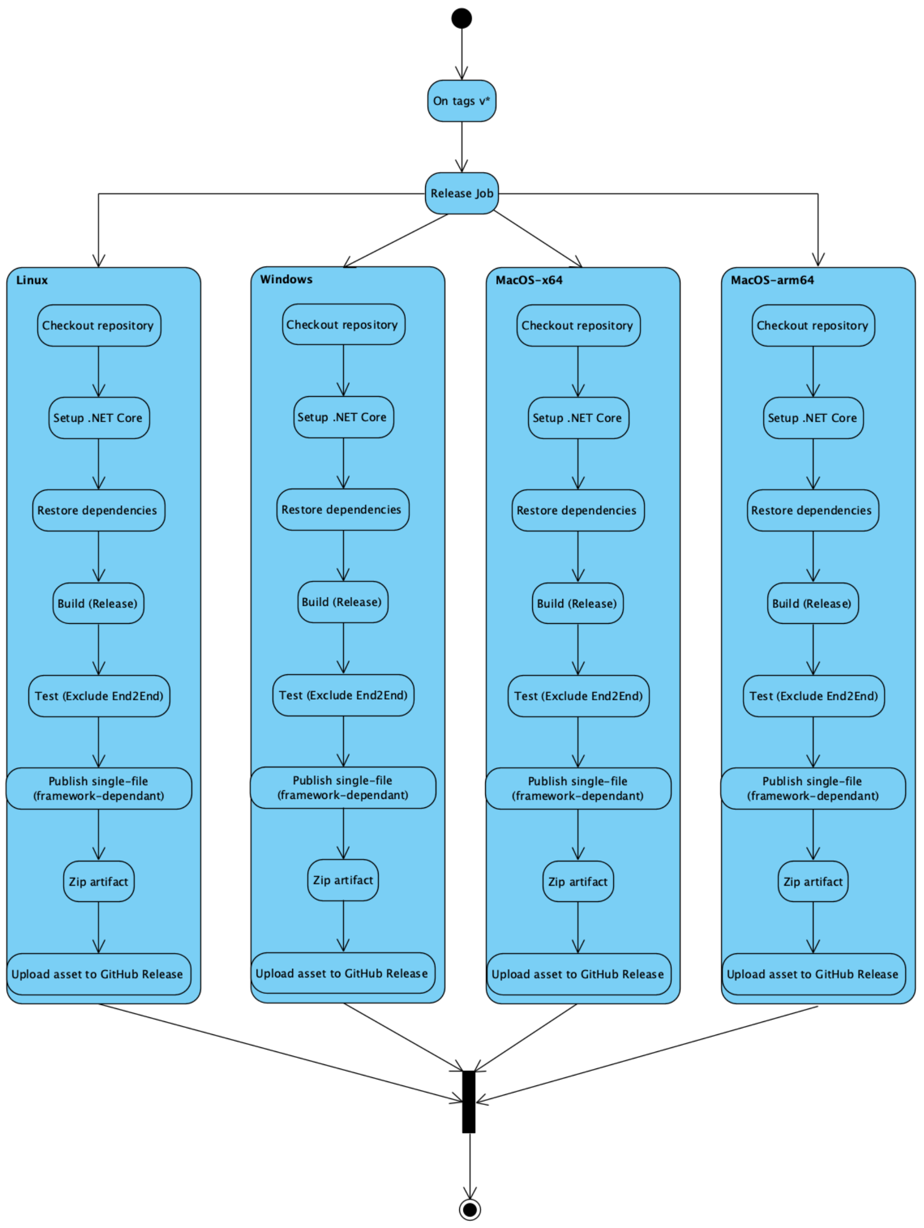


Figure 12: Executables Diagram

The fourth diagram is created over the Auto Label workflow. It shows how labels are automatically assigned to newly opened or edited issues based on predefined keywords, such as “bug”, “layout”, etc. This automation improves issue organization and supports more effective use of the project board.

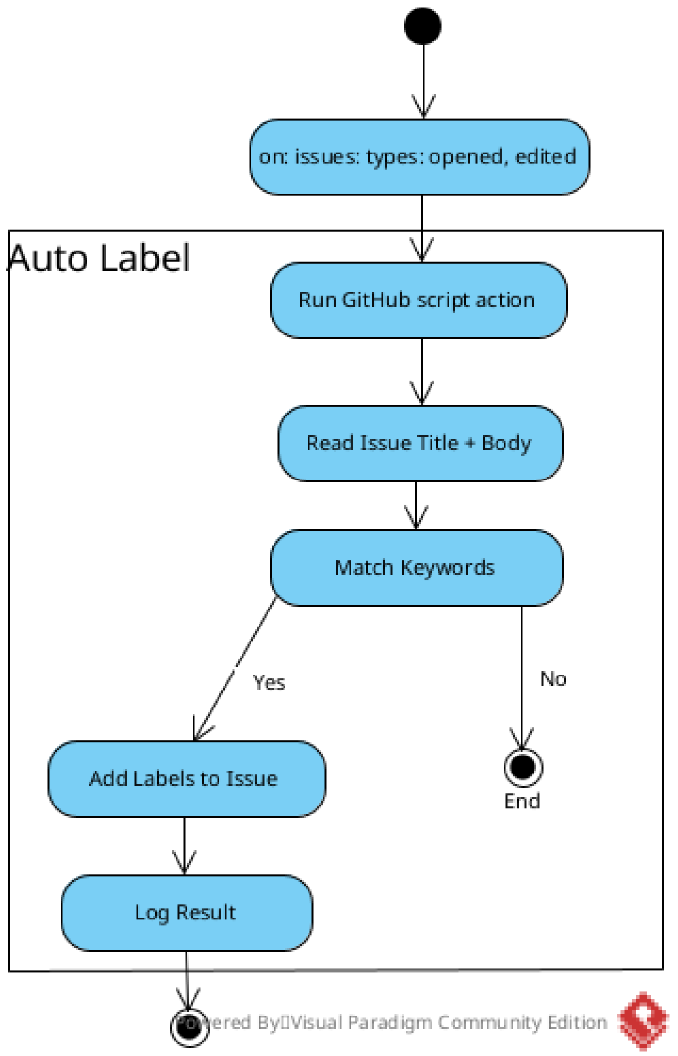


Figure 13: Auto Label

2.2 Team work

The structure around which we have organized our work can be described with the following sequence diagram.

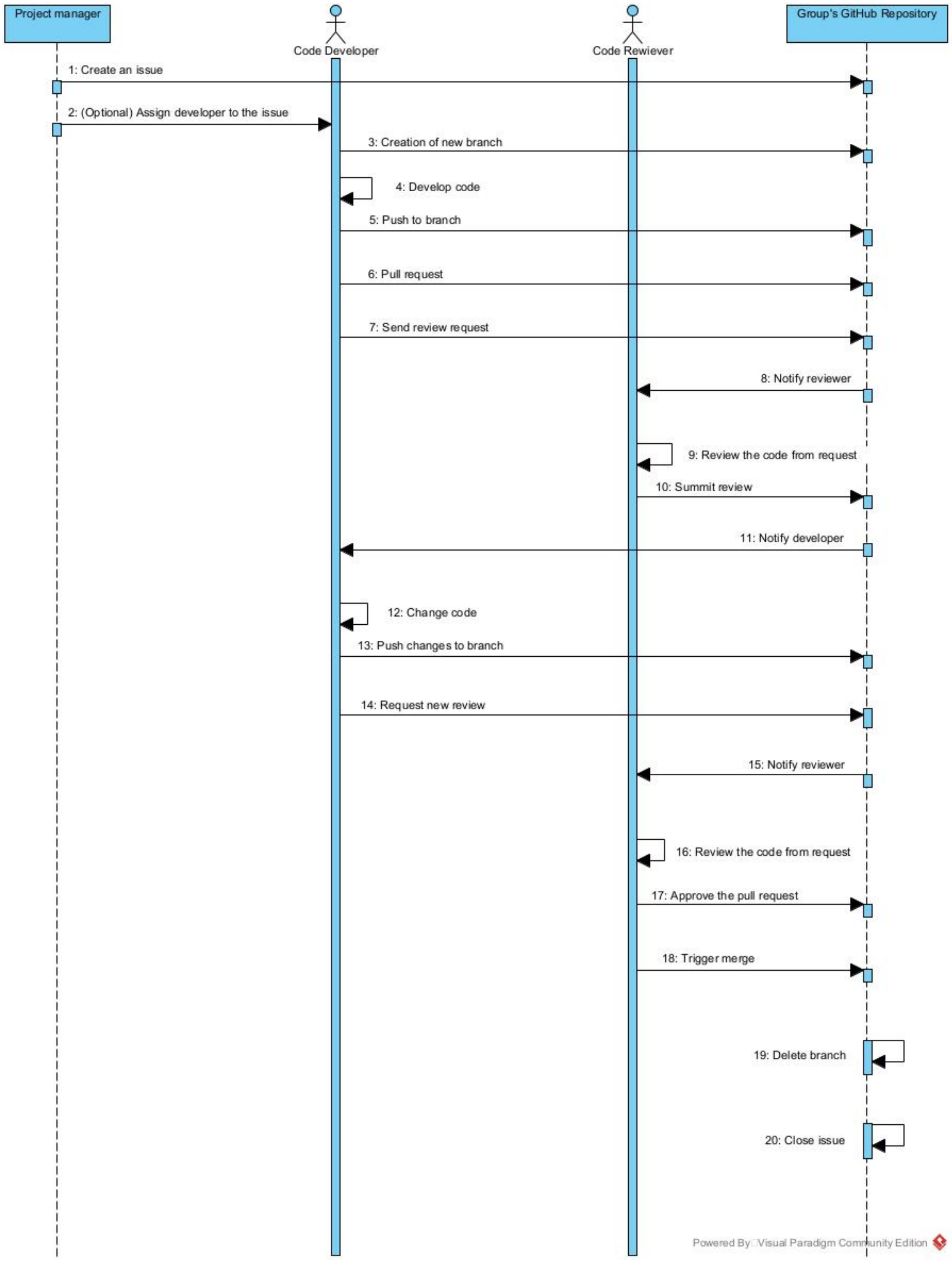


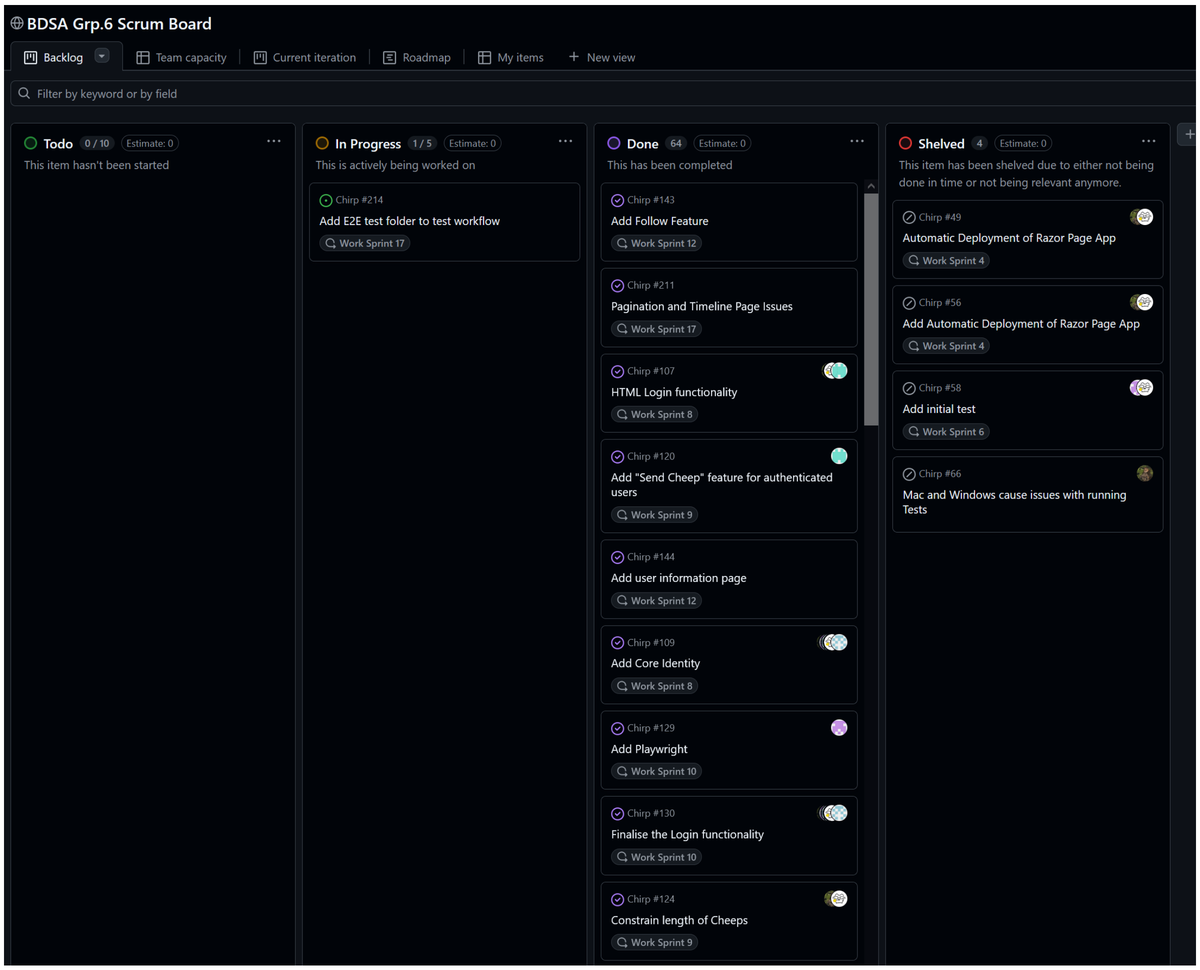
Figure 14: Team Work Sequence Diagram

As illustrated in the picture, our process had the following main components:

- **Creating an issue:** During the first two steps of the sequence, the project manager creates a new issue. This issue is based on either the requirements presented in the given week or on bugs found during development. After creating the issue, including a description and success requirements, a developer can be assigned to it.
- **Code development:** The issue is then picked up by a developer from the team. In the next steps of the sequence, the developer creates a new designated branch for the issue, develops the feature, pushes the code to the GitHub repository, and finally requests review of the implemented code.
- **Code review:** GitHub then notifies the team about the pull request, and a reviewer reviews the code. The review is based on the success criteria defined when the issue was created. The reviewer submits the review through GitHub, which then notifies the developer whether the pull request was approved or if changes are requested. If changes are requested, the developer updates the code, pushes the changes and sends a new review request through the GitHub repository. If the pull request is approved, the sequence continues to the merge step.
- **Closing the issue:** Once the pull request is merged, the branch is deleted and the issue is closed. The process starts again, with the creation of a new issue.

2.2.1 Snippet of Final Scrum Board

We used a Scrum Board to keep track of current issues and their status.



BDSA Grp.6 Scrum Board

Backlog | Team capacity | Current iteration | Roadmap | My items | New view

Filter by keyword or by field

Todo 0 / 10 Estimate: 0
This item hasn't been started

In Progress 1 / 5 Estimate: 0
This is actively being worked on

- Chirp #214 — Add E2E test folder to test workflow — Work Sprint 17

Done 64 Estimate: 0
This has been completed

- Chirp #143 — Add Follow Feature — Work Sprint 12
- Chirp #211 — Pagination and Timeline Page Issues — Work Sprint 17
- Chirp #107 — HTML Login functionality — Work Sprint 8
- Chirp #120 — Add "Send Cheep" feature for authenticated users — Work Sprint 9
- Chirp #144 — Add user information page — Work Sprint 12
- Chirp #109 — Add Core Identity — Work Sprint 8
- Chirp #129 — Add Playwright — Work Sprint 10
- Chirp #130 — Finalise the Login functionality — Work Sprint 10
- Chirp #124 — Constrain length of Cheeps — Work Sprint 9

Shelved 4 Estimate: 0
This item has been shelved due to either not being done in time or not being relevant anymore.

- Chirp #49 — Automatic Deployment of Razor Page App — Work Sprint 4
- Chirp #56 — Add Automatic Deployment of Razor Page App — Work Sprint 4
- Chirp #58 — Add initial test — Work Sprint 6
- Chirp #66 — Mac and Windows cause issues with running Tests

Figure 15: Final Scrum Board

2.2.2 Unresolved Tasks

We did not manage to cover all test cases for the initial part of the project (Chirp.Razor). We have prioritized getting Azure to work, but encountered issues with it during the final phases of the Follows functionality. Currently, we believe it to be functioning, but our Quota is exceeded. We hope to be able to test it around the date of delivery, when Quota resets.

2.3 How to make *Chirp!* work locally

2.3.1 Prerequisites

- .NET 8 SDK
- Git

2.3.2 Clone Repository

Open a terminal and navigate to preferred directory. Then, run:

```
git clone https://github.com/ITU-BDSA2025-GROUP6/Chirp.git
cd Chirp
```

The project files should now be visible in the terminal.

2.3.3 Restore Dependencies

Install all required dependencies and tools for the project by running:

```
dotnet restore
```

A confirmation message should indicate that all packages have been restored successfully or that all projects are up-to-date for restore.

2.3.4 (Optional) Add Open Authentication Secrets

Initialize User Secrets Storage Run the following in the terminal:

```
dotnet user-secrets init
```

Set GitHub Secrets

```
dotnet user-secrets set "authentication:github:clientId" "<ClientId>"
dotnet user-secrets set "authentication:github:clientSecret" "<ClientSecret>"
```

Set Google Secrets

```
dotnet user-secrets set "authentication:google:clientId" "<ClientId>"
dotnet user-secrets set "authentication:google:clientSecret" "<ClientSecret>"
```

2.3.5 Run Application

From project root directory, start the application by running:

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

The terminal should display messages indicating that the application is building and starting.

2.3.6 Access the Application

Once the application is running, open a browser and navigate to:

```
https://localhost:5273
```

The *Chirp!* web interface should now be running locally.

2.4 How to run test suite locally

After cloning the project repository, you can run the entire test suite or individual types of tests using the following steps:

2.4.1 1. Install Required Browsers for Playwright

The E2E tests use Playwright, so browser binaries need to be installed locally. From the project root, run:

```
pwsh bin/Debug/net8.0/playwright.ps1 install --with-deps
```

If `pwsh` is not available, install PowerShell first.

This step is required for the E2E tests to run correctly.

For any issues with Playwright or browser installation, see the official guide: `https://playwright.dev/dotnet/docs/intro`

2.4.2 2. Run All tests:

From the project root directory, run the command:

```
dotnet test
```

This will run all test cases across the project, including unit, integration, and end-to-end tests.

2.4.3 3. Run Specific Test

Each type of test can also be run separately from the project root:

- **Unit tests:** `dotnet test test/UnitTests`
- **Integration tests:** `dotnet test test/IntegrationTests`
- **E2E tests:** `dotnet test test/End2End`

3 Ethics

3.1 License

During this project, we have learned about the importance of licensing and the ethics of open source development. This has added new understanding and new perspectives to the reality surrounding the Chirp! application. For licensing, we have chosen the MIT License. This choice is based on the simplicity of the license, as well as the its broad use and familiarity. Having this license means that others are free to use, modify and distribute our application, as long as the original license and copyright notice are included.

3.2 LLMs, ChatGPT, CoPilot, and others

ChatGPT and Copilot have been utilized in the making of this project. The LLMs were used for debugging and understanding different frameworks used. Deployment to Azure was especially difficult, and Copilot aided in correctly resolving the issues based on Azure logs. Additionally, we had it support us in coding the UI aspect of the program.

In using the LLMS, we were critical of its outputs, and aware of its potential detrimental consequences to our learning. As such, we strived to use it as a knowledgeable "peer", bouncing theory back and forth, instead of direct answers.