Chirp!_Project_Report

ITU BDSA 2025 Group 21

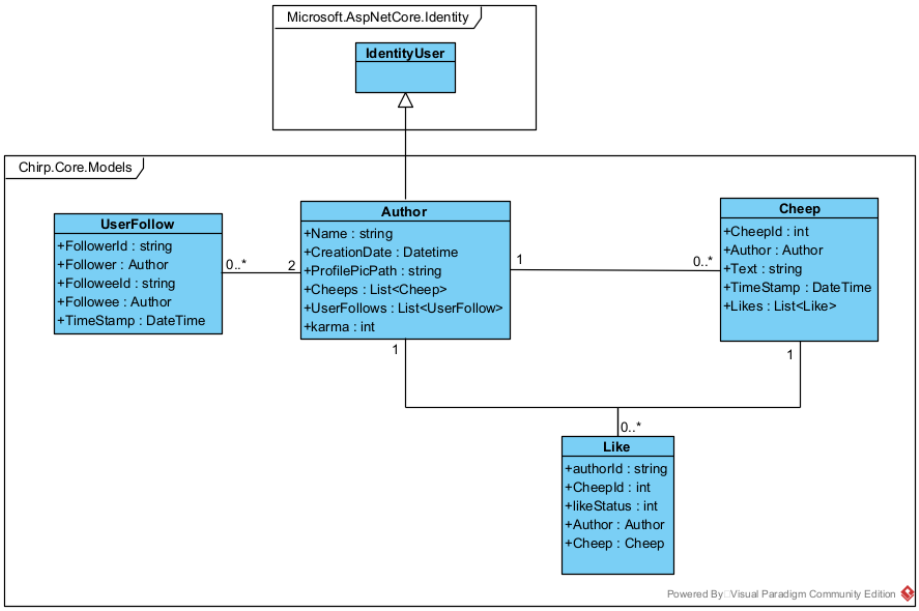
Alfred Damgaard alfd@itu.dk
Alfred Schrøder Oldin aold@itu.dk
Andreas John-Holaus andjo@itu.dk
Anton Thejsen antt@itu.dk
Noah Leerbeck Van Wagenen noav@itu.dk
Philip Bay Quorning phqu@itu.dk

1 Chirp Project Report

1.1 ITU BDSA 2025 Group 21

2 Design and architecture of Chirp!

2.1 Domain model



As illustrated, the Domain Model consists of four domain entities: **Author**,

Cheep, **Like**, and **UserFollow**. The model integrates **ASP.NET Identity**, with **Author** inheriting from `IdentityUser` to provide built-in authentication and authorization, including email and password management.

2.1.1 Author

The central entity of the domain, representing a user of the system.

Properties: - User name - Account creation date - Path to a custom profile picture - Collection of authored cheeps - Collection of followers - **Karma**: a likeness score representing user reputation

In addition, `Author` inherits all properties from `IdentityUser`, such as email and password handling.

2.1.2 Cheep

Represents a post created by an author.

Properties: - **CheepId**: Unique identifier - **Author**: Reference to the author who created the cheep - **Text**: The posted message - **TimeStamp**: Time of creation - **Likes**: Collection of `Like` entities associated with the cheep

2.1.3 Like

Represents a like or dislike issued by an author on a cheep.

Properties: - **AuthorId**: Unique identifier of the author issuing the like - **CheepId**: Unique identifier of the liked cheep - **LikeStatus**: Integer indicating like (`1`) or dislike (`-1`) - **Author**: Reference to the liking author - **Cheep**: Reference to the liked cheep

2.1.4 UserFollow

Represents a follow relationship between two authors.

Properties: - **FollowerId**: Unique identifier of the following author - **FolloweeId**: Unique identifier of the followed author - **Follower**: Reference to the author who follows - **Followee**: Reference to the author being followed - **TimeStamp**: Time when the follow occurred

2.2 Architecture – In the small

As illustrated above, this solution follows the Onion Architecture principle, where all dependencies point inward toward the domain. The goal is to keep business logic isolated from technical and presentation concerns.

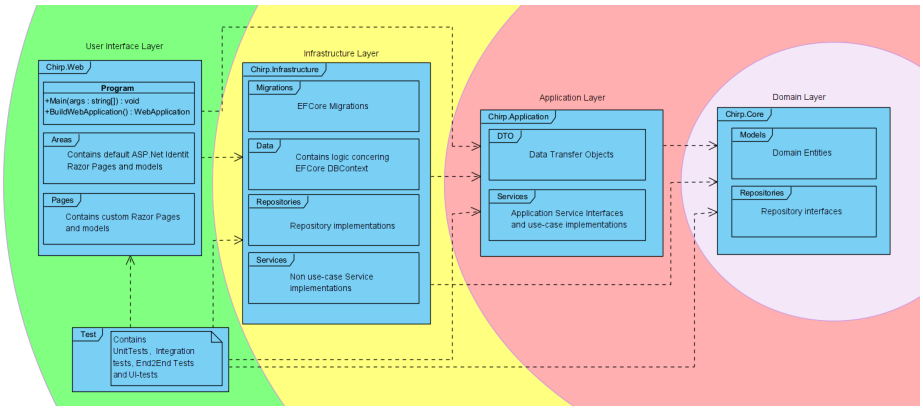


Figure 1: Architecture

2.2.1 Domain Layer

The core of the application.

- Domain entities
- Repository interfaces
- Business rules

Characteristics: - Dependency-free - Independent of frameworks and infrastructure - Represents pure business logic

2.2.2 Application Layer

Coordinates use cases and application workflows.

- Application services and use cases
- Service interfaces
- Data Transfer Objects (DTOs)
- Mapping between domain entities and DTOs

Characteristics: - Depends only on the Domain layer - Contains no infrastructure or UI code

2.2.3 Infrastructure Layer

Handles technical and external concerns.

- EF Core DbContext and migrations
- Repository implementations
- Framework- and third-party-dependent service implementations (e.g. ASP.NET Identity)

Characteristics: - Implements interfaces from inner layers - Depends on Domain and Application layers

2.2.4 User Interface Layer

Handles presentation and user interaction.

- Razor Pages and Page Models
- Dependency Injection setup
- Application startup and configuration

Characteristics: - Depends on the Application and Infrastructure layer - Contains no business logic

2.2.5 Test Layer

Validates the system across all layers.

- Unit, Integration, End-to-End, and UI tests
- Not part of the runtime architecture

2.3 Architecture of deployed application

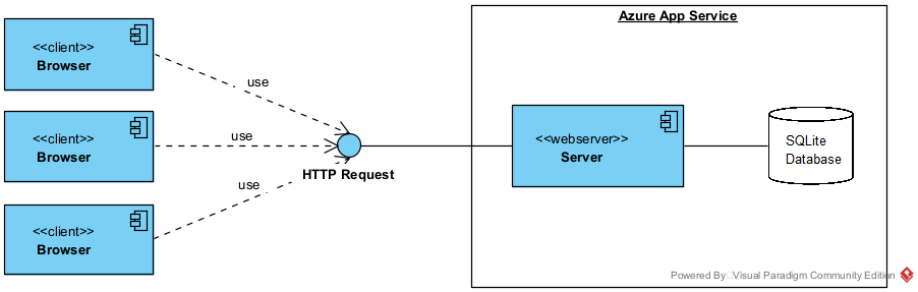


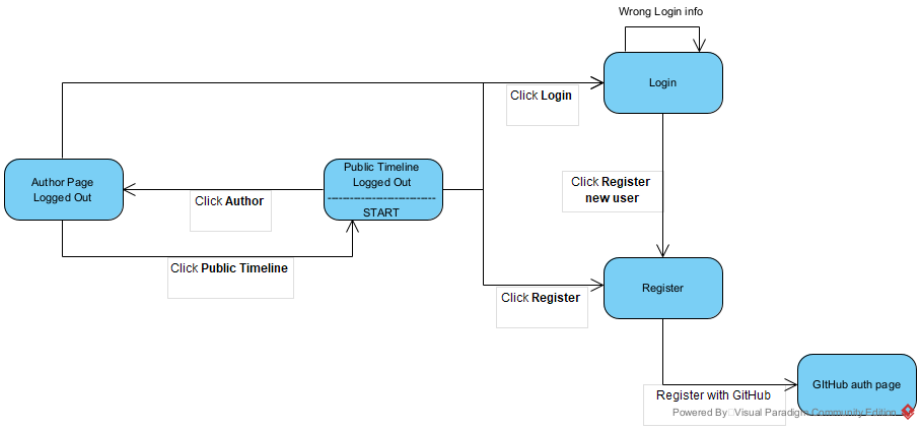
Figure 2: Deployed Architecture

The application follows a **client–server architecture**.

The server is implemented as a web application and deployed on **Azure App Service**. It exposes the required interfaces and API endpoints that enable communication between the client and the server.

3 User Activities

3.1 Acessibility for a non authorized user



When a user enters the website for the first time, they are placed on the public timeline.

In this state, the user can: - View the complete public timeline. - Navigate to and read any individual user's timeline. - Navigate to Register / Login

However, while unauthenticated, the user cannot perform any interactive actions such as posting cheeps, following authors, or liking/disliking cheeps.

This section represents the default read-only experience.

3.2 Login & Registration Flow

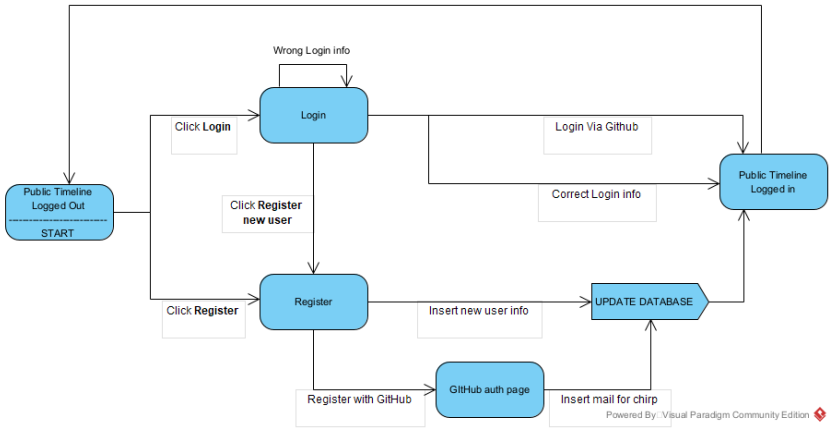


Figure 3: RegistrationFlowImage

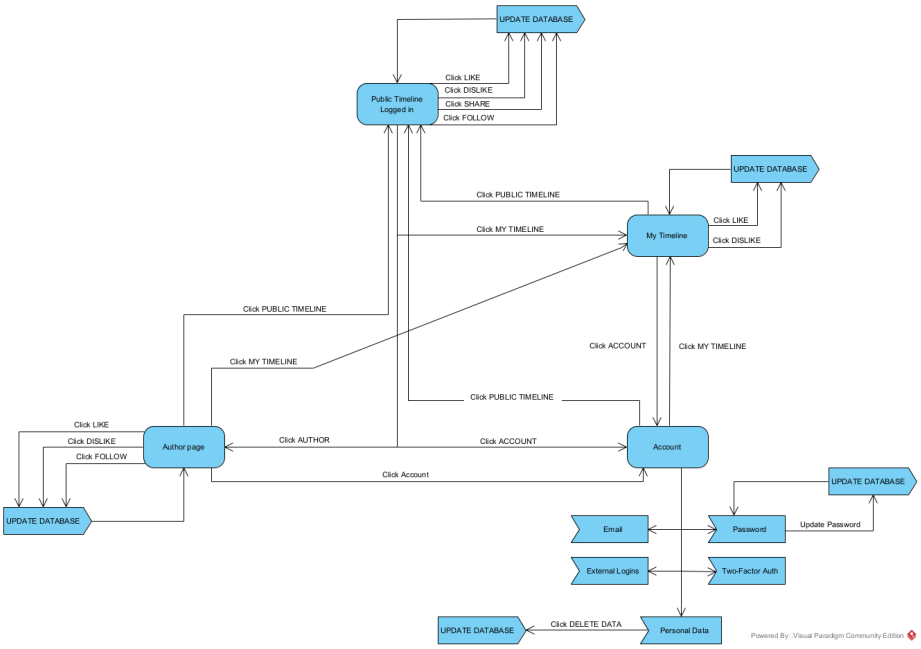
From the public timeline, an unauthenticated user may choose to log in or register as a new user.

The user can: - Navigate to the login page to access an existing account. - Navigate to the registration page to create a new account.

Both login and registration can be completed using: - Traditional credentials (e.g email, username and password) - GitHub Authentication

Successfull authentication redirects the user to the public timeline.

3.3 Acessibility for a logged in user

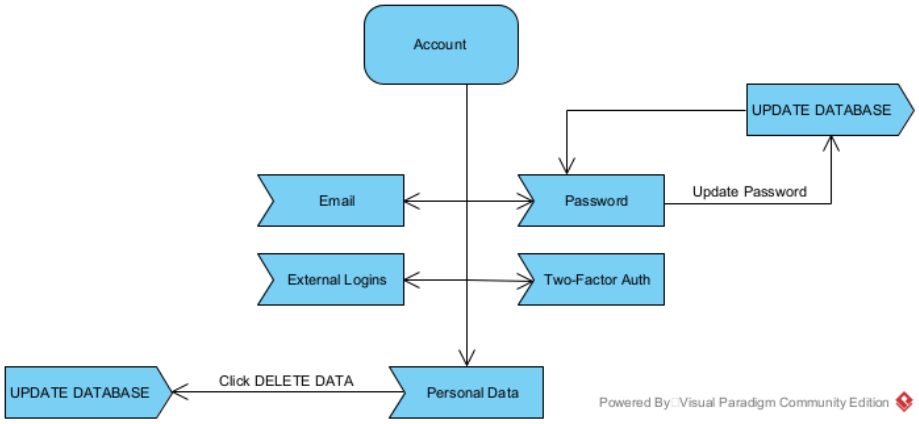


After a successful login or registration, the user is redirected back to the public timeline, now as an authenticated user.

In this state, the user retains all read-only capabilities but additionally gains the ability to: - Post new cheeps. - Follow or unfollow other users. - Like or dislike cheeps.

Additionally an authenticated user can navigate to: -User timeline -Account

3.4 Account Management & Profile Settings



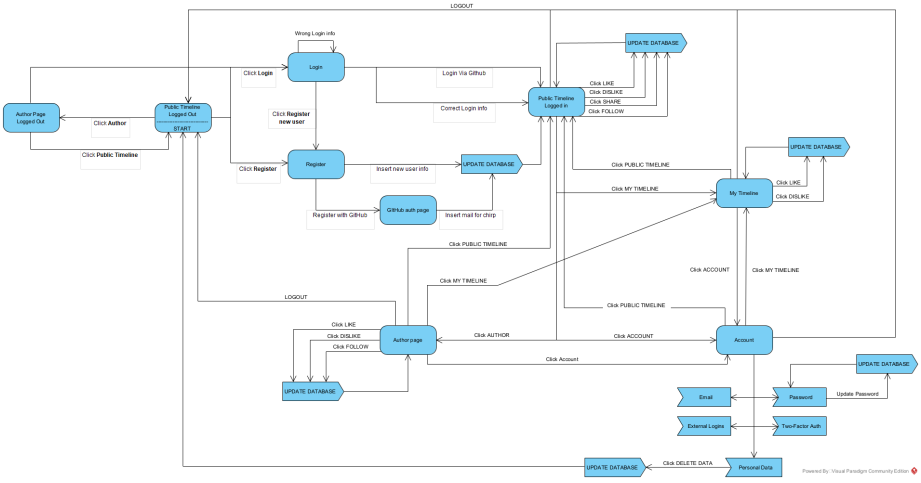
Authenticated users may access the account page, which contains personal profile information.

From this page, the user can: - View account details such as email and password. - Change their profile picture. - Change their password. - Download all personal data. - Delete their account and associated data.

If the user deletes or downloads their data, they are: - Logged out automatically. - Redirected back to the public timeline in an unauthenticated state.

This section represents identity, privacy and lifecycle management.

3.5 Full state chart for logged in and out users

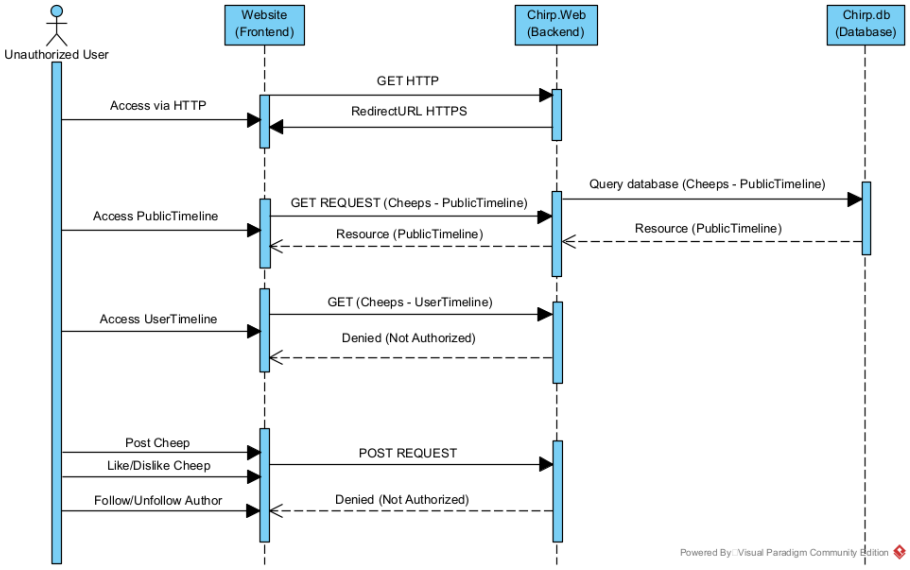


The following UML statechart represents the available “options” a user has depending on where on the website they are located, and whether they’re logged in or not.

3.6 Sequence of functionality/calls through Chirp!

The below diagrams of sequences illustrates a sequence of calls in the Chirp! application initiated by a user, for both an unauthenticated and an authenticated user.

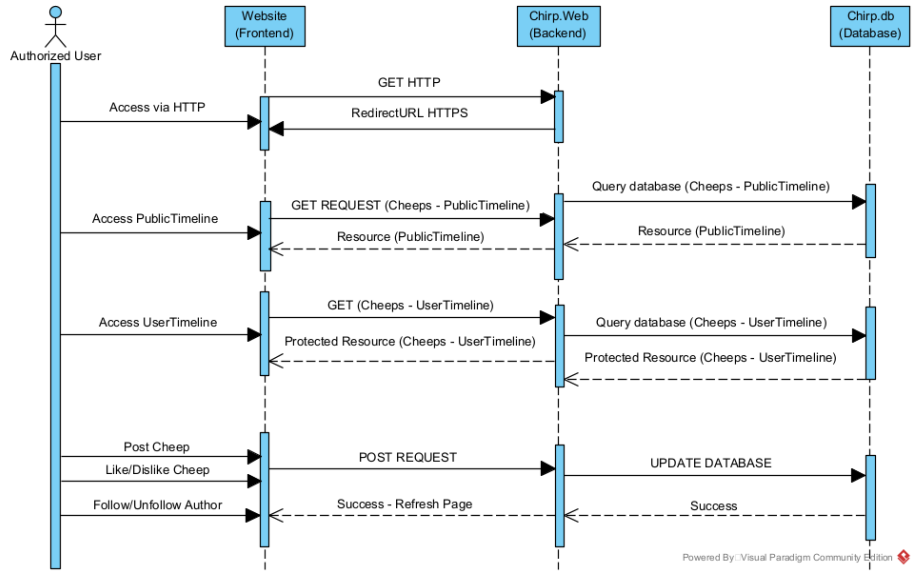
3.6.1 Unauthorized user



The diagram illustrates the sequence of calls made by an unauthorized user accessing the **PublicTimeline**, including redirection to **HTTPS**, which is permitted without authentication.

It also shows attempts to access the **UserTimeline**, like and post cheeps, and follow or unfollow other users. These actions are restricted to **authorized users only**.

3.6.2 Authorized User



The diagram illustrates the sequence of calls made by an **authorized user** accessing the **PublicTimeline**, including redirection to **HTTPS**.

It also shows authorized actions such as accessing the **UserTimeline**, liking and posting cheeps, and following or unfollowing other users.

4 Process

4.1 Build, test, release and deployment

The above diagram shows an overall activity UML illustrating the overall CI/CD pipeline of Chirp. The model shows how pull requests are validated through automated testing, merged into the main branch, deployed to production, and optionally released as versioned artifacts. Below we dive into each workflow process.

4.1.0.1 Continuous Integration Workflow The above figure expands the continuous integration from the previous model. When a pull request is opened or updated, the workflow restores dependencies, builds the application, and executes the automated test suite. If any step fails, the workflow ends and the pull request cant be merged. CodeFactor is used for static code analysis, and identified quality issues must be resolved before the pull request can be accepted

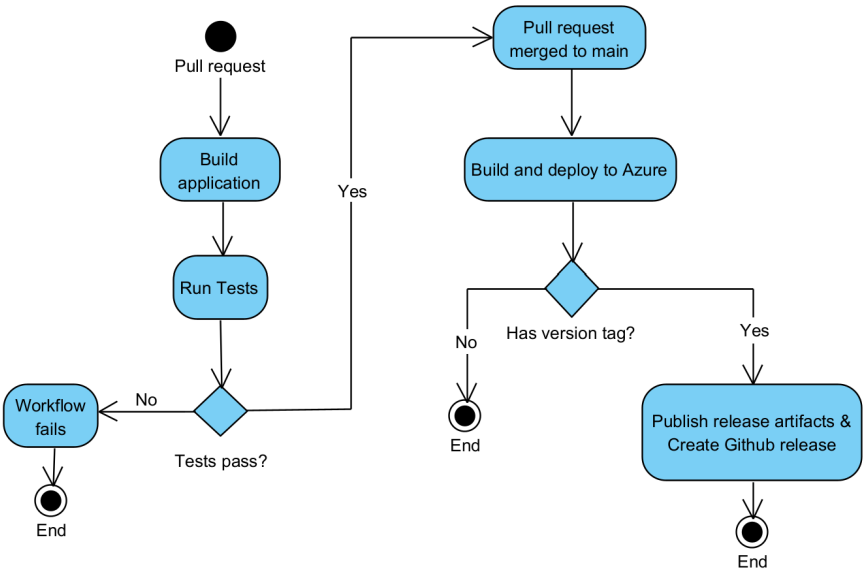


Figure 4: githubWorkflow

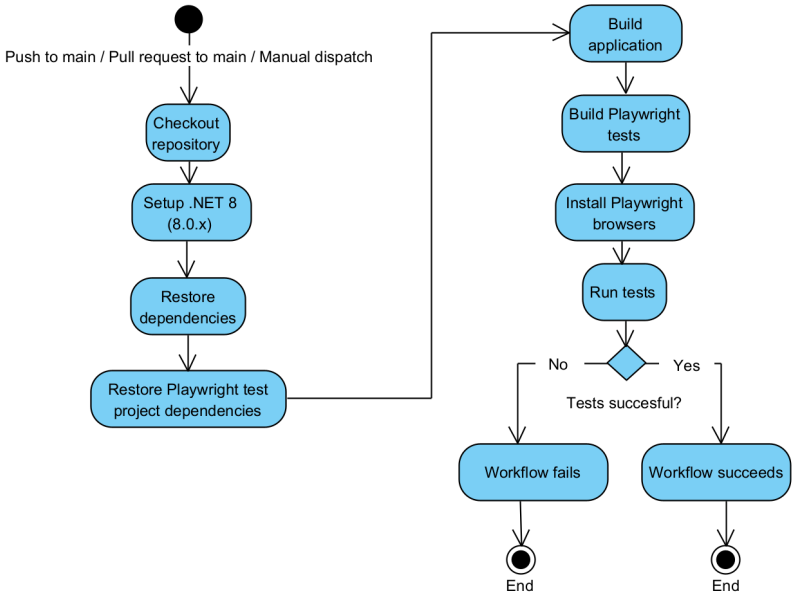
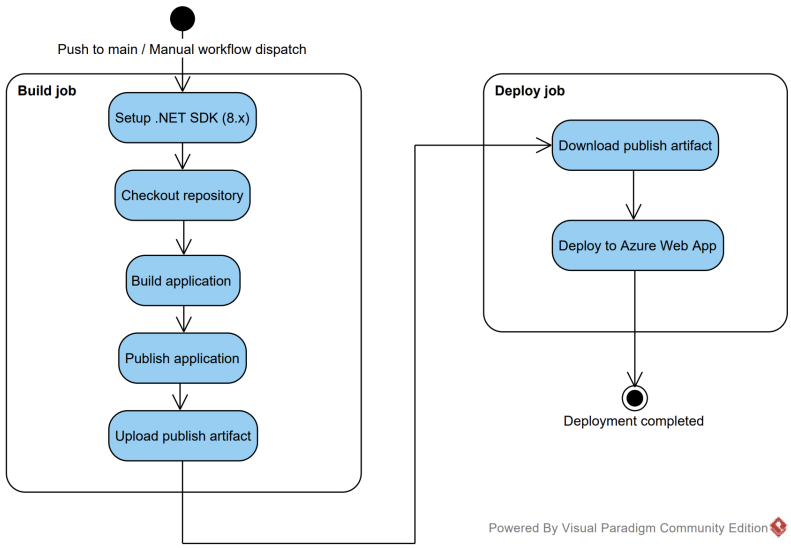


Figure 5: Automatic test build - github workflow



4.1.0.2 Deployment Workflow

The above figure expands the deployment stag. Once changes are merged into the main branch, the workflow builds the application and deploys it to an Azure Web App. This ensures that production always reflects a tested and integrated version of the codebase.

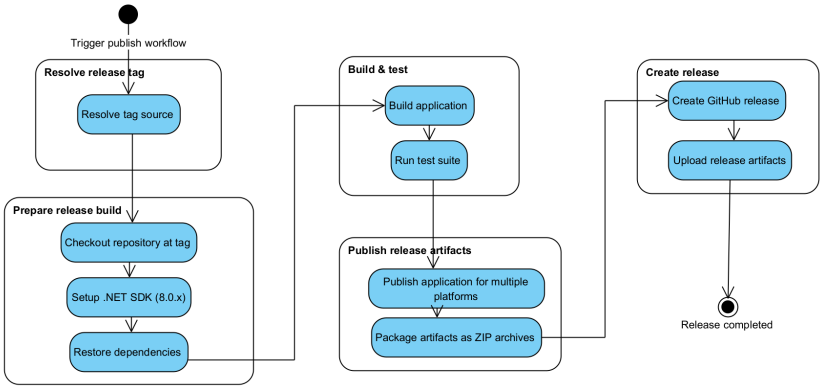
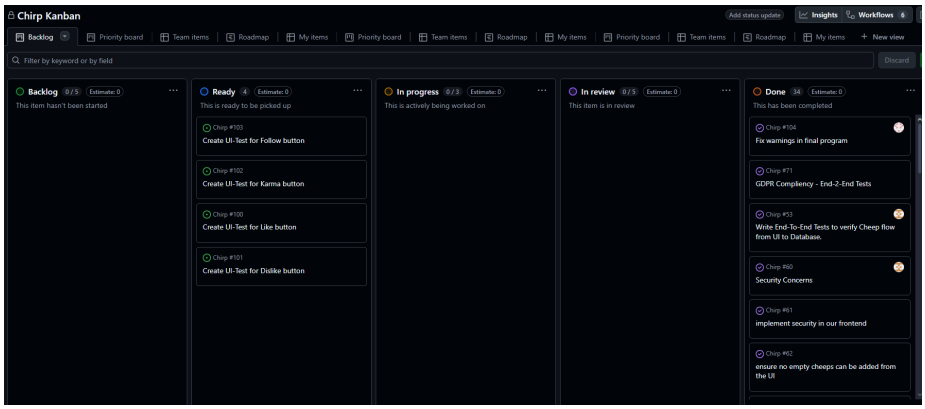


Figure 6: Publish - workflow

4.1.0.3 Release Workflow The figure above expands the release stage. When a merge is associated with a version tag, the publish workflow is triggered, producing versioned release artifacts and creating a corresponding GitHub release.

4.2 Team work

4.2.1 Project Board



Above shows our project board. All wished features and functionalities have been implemented.

We where not able to finish UI-tests of the latest features in time, being UI Tests foer the like, dislike, follow and karma buttons.

4.2.2 Trunkbased Development

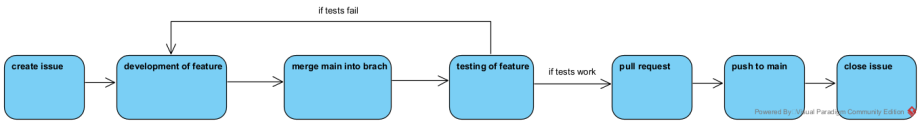


Figure 7: TrunkbasedDevelopment

During our development of Chirp we have followed a trunk-based development workflow. Short-lived branches were created for individual tasks and features, each branching from the main trunk. Development was carried out on these branches with frequent, small commits documenting progress. Once a task is complete a pull request was opened targeting the main branch. Project members reviewed the changes to ensure correctness and adherence to project standards. After approval, branches were merged into main, and the corresponding issues were closed. This workflow allowed us to maintain a stable main branch while supporting rapid development and collaboration.

4.2.3 Full workflow process

The model above is a workflow UML that shows an abstraction of the full workflow process, used throughout the development. This model was created early on in the project, to create a uniform and structured way of developing features. Development begins with Github issue with a structure based on a user

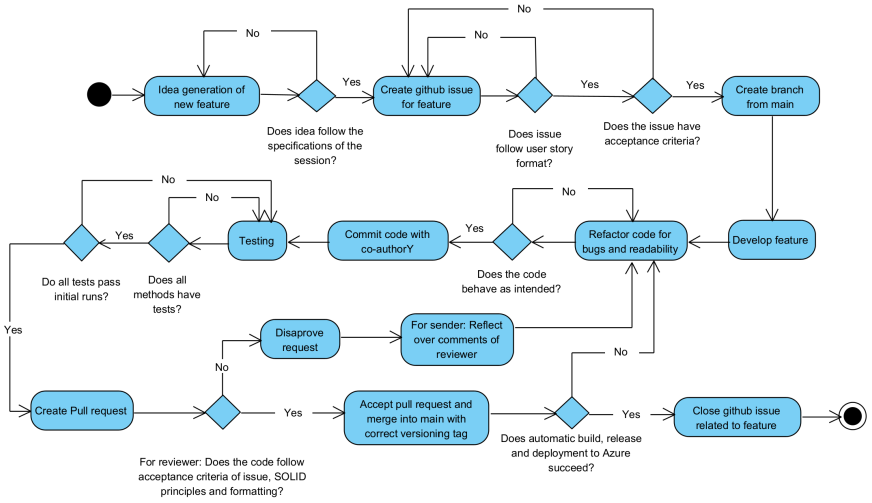


Figure 8: developmentProcessWorkflow

story, containing a description and acceptance criteria. This is kept non-technical and purely user-experience or stakeholder oriented. Each issue should aim to be consice enough to be implemented, reviewed and merged through a single short-lived feature branch. This is not a requirment, but compliments the idea of trunk-based Git branching strategy. For each issue, a feature branch is created with a name that reflects the issue or the feature being worked on. When the feature has been developed and tests are added and passed, a pull request is created for peers to review. The reviewer should ensure that the acceptance criteria and SOLID principles are covered. Once the Github workflow finishes, the related issue is closed.

4.2.4 Pair Programming and Commit Attribution

During the project, a significant portion of the development was carried out through pair programming sessions using Visual Studio Live Share. In these sessions, two or more team members worked together simultaneously on the same code, discussing design decisions and implementing solutions collaboratively.

Although Git supports the use of Co-Authored-By tags, these were not applied consistently or always correctly during development. As a result, the commit history does not fully reflect the actual distribution of contributions, as many commits represent collaborative work rather than individual effort. The commit log should therefore be understood as a technical record of changes, not as an exact measure of individual contribution. Knowledge sharing and joint problem solving were intentional parts of the development process and played a significant role in the final solution. This is also reflected in a large amount of commits, where only 1 is credited for the commit. Examples include: 133f7ee, 3cba4c0,

386f3b1, d6e8216...

4.3 Setting up and running Chirp! locally

4.3.1 Step 1: Clone the repository:

Clone the Chirp! Repository to your local machine using Git.

4.3.2 Step 2: Ensure the correct .NET version is installed:

Verify that .NET SDK 8.0 is installed on your system. The project targets .NET 8.0 and is not compatible with earlier versions.

4.3.3 Step 3: Restore project dependencies:

From the root of the solution, restore all required dependencies by running this .NET CLI command:

```
dotnet restore
```

4.3.4 Step 4: Configure the authentication secrets:

Before running the application, configure the GitHub authentication secrets using the .NET user-secrets feature. These secrets are required for authentication to function correctly.

```
dotnet user-secrets set "Authentication:GitHub:ClientSecret"
"bcae20e854008588fdc845fe904f43d7a204c424"
dotnet user-secrets set
"Authentication:GitHub:ClientId" "Ov23li7jj1fMsnWTWXzO"
```

4.3.5 Step 5: Build and run the application

Nagivate to the Chirp.Web project directory and start the application by running:

```
dotnet run
```

4.3.6 Step 6: Access the application

Once the application has started successfully, the console will display a localhost URL. Open this URL in a web browser to access the Chirp! application.

4.3.7 Running Test Suite locally

To validate the system locally, the automated test suite can be executed by running

```
dotnet test
```

from the root of the Chirp! solution. This command will build all the required projects. To run the Playwrights tests, you must first install playwright using the following commands in your terminal:

```
npm install -D @playwright/test
npx playwright install
```

Then when you use

```
dotnet test
```

The playwright tests will also run.

5 Ethics

5.1 License

MIT License

Copyright (c) 2025 ChirpGroup21

Permission is hereby granted, free of charge, to any person obtaining a copy of this software and associated documentation files (the "Software"), to deal in the Software without restriction, including without limitation the rights to use, copy, modify, merge, publish, distribute, sublicense, and/or sell copies of the Software, and to permit persons to whom the Software is furnished to do so, subject to the following conditions:

The above copyright notice and this permission notice shall be included in all copies or substantial portions of the Software.

THE SOFTWARE IS PROVIDED "AS IS", WITHOUT WARRANTY OF ANY KIND, EXPRESS OR IMPLIED, INCLUDING BUT NOT LIMITED TO THE WARRANTIES OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE AND NONINFRINGEMENT. IN NO EVENT SHALL THE AUTHORS OR COPYRIGHT HOLDERS BE LIABLE FOR ANY CLAIM, DAMAGES OR OTHER LIABILITY, WHETHER IN AN ACTION OF CONTRACT, TORT OR OTHERWISE, ARISING FROM, OUT OF OR IN CONNECTION WITH THE SOFTWARE OR THE USE OR OTHER DEALINGS IN THE SOFTWARE.

5.2 LLMs, ChatGPT, CoPilot, and others

During the development of Chirp, we made limited and deliberate use of Large Language Models (LLMs) as a supportive learning and reference tool. LLMs were primarily used to help us understand various .NET libraries, third-party frameworks, and other unfamiliar technical aspects, providing explanations, clarification of documentation, and general guidance. They were not used to

generate raw application code, and no LLM-generated code was incorporated into the final solution.

Copilot was also used as an initial aid for the documentation process, offering preliminary suggestions and structure. These suggestions were carefully reviewed and fully rewritten or adapted by the project group to ensure accuracy, clarity, and alignment with the implementation. Overall, LLMs served as a supplementary resource rather than a development tool, with all design decisions, implementation work, and final outputs produced entirely by the project team.