Chirp! Project Report

ITU BDSA 2025 Group 19

Hans Voldby Larsen havl@itu.dk
Nikolaj Severin Due Olsen nsol@itu.dk
Kasper Fjord Grønvold Jensen kfje@itu.dk
Alex Matlok almat@itu.dk

1 Design and Architecture of *Chirp!*

1.1 Domain model

The domain model is a consists of the classes in the core domain of interest, and the relationships between them. The domain model is visually shown in figure 1 as a UML diagram. Only the core entities are shown so other classes of the app concerning eg. infrastructure or web pages are not part of the domain model.

The core domain consists of the entities Author, Cheep and Follow, where an Author can publish multiple Cheeps and each Cheep is authored by exactly one Author.

Authors can follow other authors, which is modeled through the Follow entity representing the follower–followee relationship.

Authentication and authorization are handled via ASP.NET Identity and are therefore not part of the core domain model. The property ApplicationUserId in the class Author (which is used to connect a user indentity with an Author) is therefore not included in the Domain model.

1.2 Architecture — In the small

Figure 2 illustrates the internal architecture of the Chirp! application. The Solution follows a layered architecture with a clear separation between **domain**, **application**, **infrastructure** and **presentation**.

Figure 3 complements the onion architecture overview by showing a UML package diagram that maps the architectural layers to the concrete project structure and illustrates the dependency relationships between them. All dependencies between classes and inhertances are not shown to avoid the illustration to become too overcrowded.

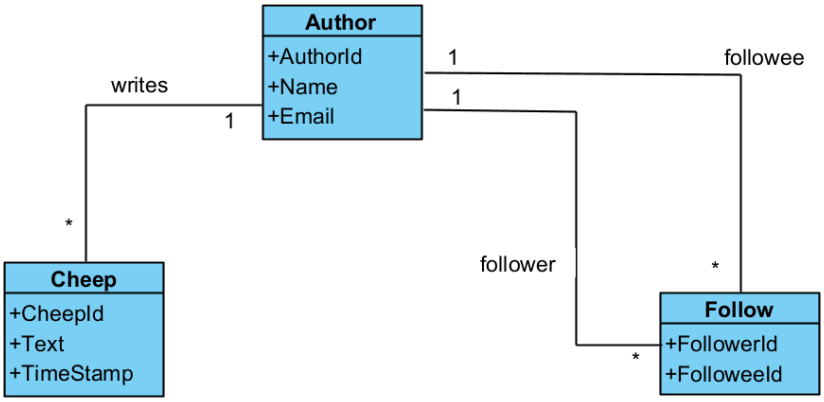

Figure 1: Illustration of the *Chirp!* data model as UML class diagram.

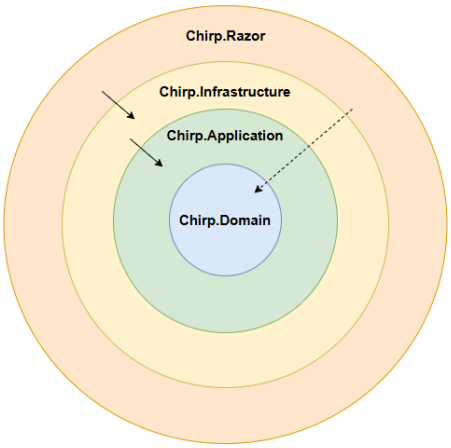

Figure 2: Illustration of the Chirp! onion Archhitecture.

1.2.1 Domain layer

The domain layer (project Chirp.Domain) contains the core domain entities and is completely independent of other layers. The class Author in the domain has a property called ApplicationUserId and is used to associate a domain author with an authenticated user. This property is represented as a primitive value and does not introduce a dependency on the authentication framework.

1.2.2 Appliction layer

The application layer (project Chirp.Application) defines service interfaces and operates exclusively on data transfer objects (DTOs).

The service interfaces define the operations that the presentation layer can use to interact with the application's functionality. For example, the ICheepService interface specifies methods for retrieving cheeps in different contexts, such as paginated lists of cheeps, cheeps authored by a specific author, cheeps associated with a user, and cheeps forming a user timeline. By letting the presentation layer depend abstract interfaces and not concrete implementations of these services, the presentation layer remains decoupled from the underlying infrastructure, allowing service implementations to be replaced or modified without affecting higher layers.

Data Transfer Objects (DTOs) are used to decouple the application and presentation layers from the domain model by exposing only the data required for a specific use case.

1.2.3 Infrastructure layer

The infrastructure layer (project Chirp.Infrastructure) is responsible for data persistence and contains repositories which encapsulate database access logic and isolate the rest of the application from the Entity Framework Core–specific details. Furthermore it provides the concrete implementations of the service interfaces defined in the application layer.

The data persistence infrastructure is centered around the ChirpDbContext, which serves as the Entity Framework Core database context and defines the mapping between the domain entities and the underlying relational database. The ChirpDbContext is kept lightweight and contains only the aggregate entities from the domain layer (Author, Cheep, and Follow).

The ChirpDbContext inherits from IdentityDbContext, which integrates ASP.NET Identity into the persistence layer and is responsible for managing the database tables related to authentication and authorization (e.g., user accounts, roles, and claims).

Supporting components such as database migrations and the DbInitializer are included to manage schema evolution and initial data seeding.

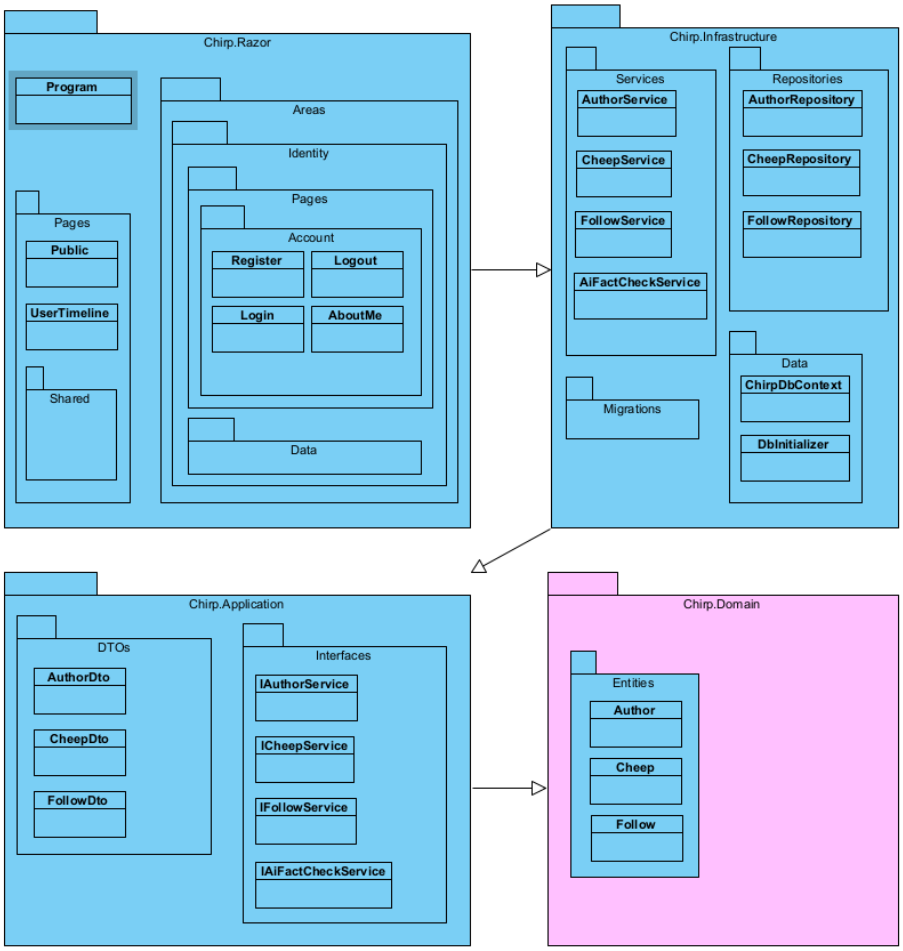

Figure 3: Illustration of the *Chirp!* Archhitecture.

The service implementations (e.g., AuthorService, CheepService, FollowService, and AiFactCheckService) implements the behavior specified by the corresponding application-layer interfaces. These services coordinate domain entities and repositories to fulfill application use cases while remaining hidden behind the abstractions defined in the application layer.

1.2.4 Presentation layer

The presentation layer (project Chirp.Razor) contains all components related to user interaction and request handling. It includes the Razor Pages that render the web-based user interface, the application entry point (Program.cs) where the system is configured and composed, and the authentication-related pages generated through ASP.NET Identity scaffolding. Together, these components are responsible for presenting data to users, handling input, and delegating application logic to the underlying layers.

Ideally, the presentation layer should depend only on service interfaces and data transfer objects (DTOs) defined in the application layer. In practice, some Razor Pages also reference domain entities directly, resulting in limited direct dependencies on the domain layer. While this slightly weakens the intended separation, the overall responsibility boundaries between layers remain clear.

1.2.4.1 Program.cs The Program.cs file defines how the system is assembled at runtime. It is responsible for configuring dependency injection, registering services, setting up authentication and authorization.

Within Program.cs, application-layer service interfaces are bound to their concrete infrastructure-layer implementations, and repositories, database contexts, and identity services are registered with the dependency injection container. The file also configures external authentication via GitHub OAuth and integrates ASP.NET Identity for user management. Also authentication, authorization, static file handling, and session management is configured here. By centralizing these concerns, Program.cs keeps configuration and wiring separate from application logic and presentation concerns.

1.2.4.2 Custom Razor Pages The custom Razor Pages developed for Chirp! implement the core user-facing functionality of the application. These pages handle displaying timelines and creating new cheeps. Each page is responsible for handling HTTP requests, invoking application-layer services, and rendering the resulting data in the user interface.

The Razor Pages primarily interact with the application layer through service interfaces such as ICheepService, IAuthorService, and IFollowService, which encapsulate application logic and data access. In some cases, domain entities are accessed directly from the presentation layer, reflecting pragmatic trade-offs made during development. Nevertheless, the overall flow of control remains from the presentation layer toward the application and infrastructure layers.

1.2.4.3 Razor Pages Generated via ASP.NET Identity Scaffolding

In addition to the custom pages, the presentation layer contains Razor Pages generated using ASP.NET Identity scaffolding. These pages are organized under the Areas/Identity structure and provide functionality related to authentication and account management, including login, logout, registration, and access control.

These scaffolded pages rely on the ASP.NET Identity framework and operate largely independently of the application's domain and application layers. By using the standard scaffolding approach, authentication and authorization concerns are handled using well-established framework components, reducing the need for custom implementation while keeping identity-related functionality isolated from the core application logic.

The "About Me" Page is a custom made Page but is put together with the Identity Pages since it has a tight connection to the user account.

1.3 Architecture of deployed application

Figure 4 illustrates the deployment architecture of the Chirp! application and shows how the system is composed at runtime. The application is deployed as a client–server web application consisting of a web browser client, a server-side ASP.NET Core application hosted on Azure App Service, a local SQLite database, and external services accessed over HTTPS.

The client side consists of a standard web browser running on a client computer. Users interact with the application through rendered HTML pages and submit requests via forms and links. All communication between the client and the server takes place over HTTPS, ensuring secure transmission of data.

The server side is hosted within an Azure App Service environment and runs a web application that handles incoming HTTP requests, executes application logic, and renders Razor Pages. The server acts as the central coordinator of the system, mediating all communication between the client, the database, and external services.

Data persistence is provided by a SQLite database that is deployed within the same Azure App Service environment as the web application. As SQLite is a file-based database, it is accessed locally by the web application without network communication. The database stores both application-specific data (such as authors, cheeps, and follow relationships) and authentication-related data managed by ASP.NET Identity.

In addition to local persistence, the deployed application integrates with two external services over HTTPS. GitHub OAuth is used to support user authentication via OAuth 2.0, allowing users to sign in using their GitHub accounts. During authentication, the web application communicates directly with GitHub's OAuth endpoints, while the client browser interacts only with the Chirp! server. Furthermore, the application communicates with the OpenAI API over HTTPS using a REST-based interface to perform AI-assisted fact checking of user-submitted

content. Both external services are accessed exclusively by the server and are not part of the deployed system itself.

Overall, the deployment architecture follows a client–server model in which the browser acts as a client, the application hosted on Azure App Service serves as the central runtime component, and all persistence and external integrations are handled server-side.

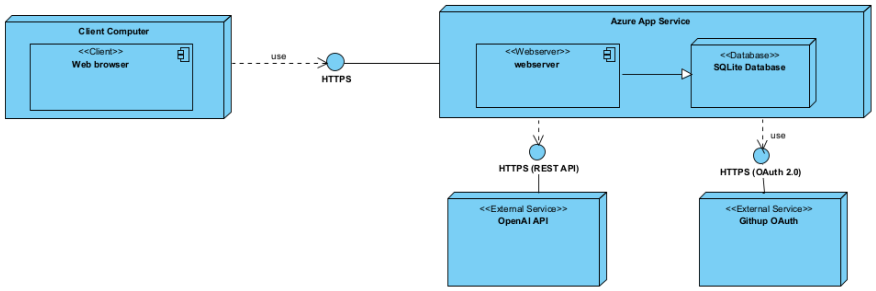

Figure 4: Architecture of the deployd aplication.

1.4 User activities

1.5 Sequence of functionality/calls trough *Chirp!*

Figure 5 illustrates the runtime flow of an unauthenticated request through the Chirp! application using a UML sequence diagram. The diagram shows starts with an HTTP request from a web browser and ending with a fully rendered HTML page returned to the client.

The interaction begins when a web browser sends an HTTP GET request to the root endpoint of the application. The request is received by the ASP.NET Core runtime, which is responsible for routing incoming requests to the appropriate Razor Page based on the configured routing rules. In this case, the request is routed to the Public Razor Page.

The Public Razor Page coordinates the retrieval of data required to render the page. To obtain the list of cheeps for the public timeline, it invokes the CheepService through its application-layer interface. The service call represents an application-level operation that encapsulates the business logic required to fetch cheeps independently of persistence concerns.

The CheepService delegates data access to the CheepRepository, which is responsible for interacting with the persistence layer. The repository uses Entity Framework Core, represented by the ChirpDbContext, to query the underlying SQLite database. The database returns a collection of cheep entities, which are

mapped to data transfer objects (DTOs) and send back through the repository and service layers to the Razor Page.

Once the data has been retrieved, the Public Razor Page returns a Page() result to the ASP.NET Core runtime. The runtime then renders the associated Razor view using the populated page model, producing a complete HTML document. Finally, the rendered HTML is sent back to the web browser as an HTTP 200 OK response.

Although several operations in the diagram are implemented using asynchronous methods in C#, they are modeled as synchronous calls in the sequence diagram, as the calling components await their completion before continuing execution.

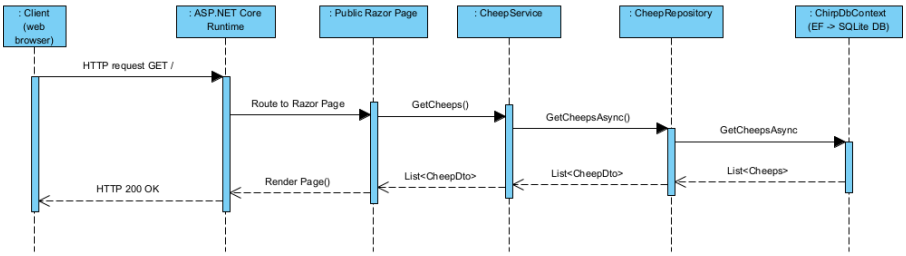

Figure 5: Sequence diagram of unauthorized user.

2 Process

2.1 Build, test, release, and deployment

GitHub Actions was used as a means for continous integration and deployment in the project.

The integration workflow runs on every push and pull request targeting the main branch. It checks the repository, installs .Net 8, restores dependencies, and builds the solution, installs Playwright for the test project, starts the Razor app locally and then runs the test suite. During testing, some configuration is required and is thus provided through environment variables such as OAuth client credentials from Github Secrets, furthermore the application is run in a dedicated testing environment to isolate test behavior from production behavior.

The deployment workflow runs on every push to main and can also be triggered with manually using workflow_displatch. For deployment, a seperate workflow builds and tests the Razor project in Release configuration and then publishes it to a folder which is uploaded as an artifact. A deploy job then donwloads said artifact, logs into Azure via azure/login, zip the output, and deploys it the the Azure Web App using azure/webapps-deploy. Seperating the build and deploy

steps into two seperate jobs ensures that the deployed package is exactly the one that was produced by the build job

For the last workflow, the release workflow, releases are handled with a tag-based workflow triggered by pushing a version / release tag matching the formart of 'v..*'. This workflow reruns build and tests as an extra quality gate and then publishes self-contained, single-file builds for Windows, macOS, and Linux. The outputs are zipped and attached to a GitHub Release created automatically.

Together, all of these workflows ensure a coherent structure in which all changes are thoroughly validated before integration, automatically deployed to production, and released for the supported platforms.

All of this can be seen in the UML activity diagrams below

2.2 Team work

The team's collaboration and development process was centered around the use of GitHub issues as the primary unit of work. Planned features, bug fixes, and improvements were represented as GitHub issues. This ensured transparency in task ownership.

The team followed a consistent workflow for progressing from an issue to a completed feature in the main branch:

- Review and complete the previous pull request Before starting new work, team members reviewed any open pull requests. Once approved, pull requests were merged using the Squash and merge.
- Assign the next issue A team member then assigned themselves to the next available GitHub issue on the project board. This made responsibility for each task explicit and reduced the risk of duplicated work.
- Create a feature branch A new branch was created from the main branch, typically named according to the issue or feature being implemented.
- Implementation on the branch Development was carried out entirely on the feature branch. The developer implemented the required functionality and verified it locally before publishing changes.
- Push branch to remote Once development was complete, the branch was pushed to the remote repository, typically using the Publish Branch functionality in GitHub Desktop.
- Create pull request A pull request was created targeting the main branch. The pull request included a short description of the changes and referenced the relevant issue.
- Automatic issue closure The pull request description included the text closes #X, where X is the GitHub issue number. This ensured that the issue was automatically closed once the pull request was merged.

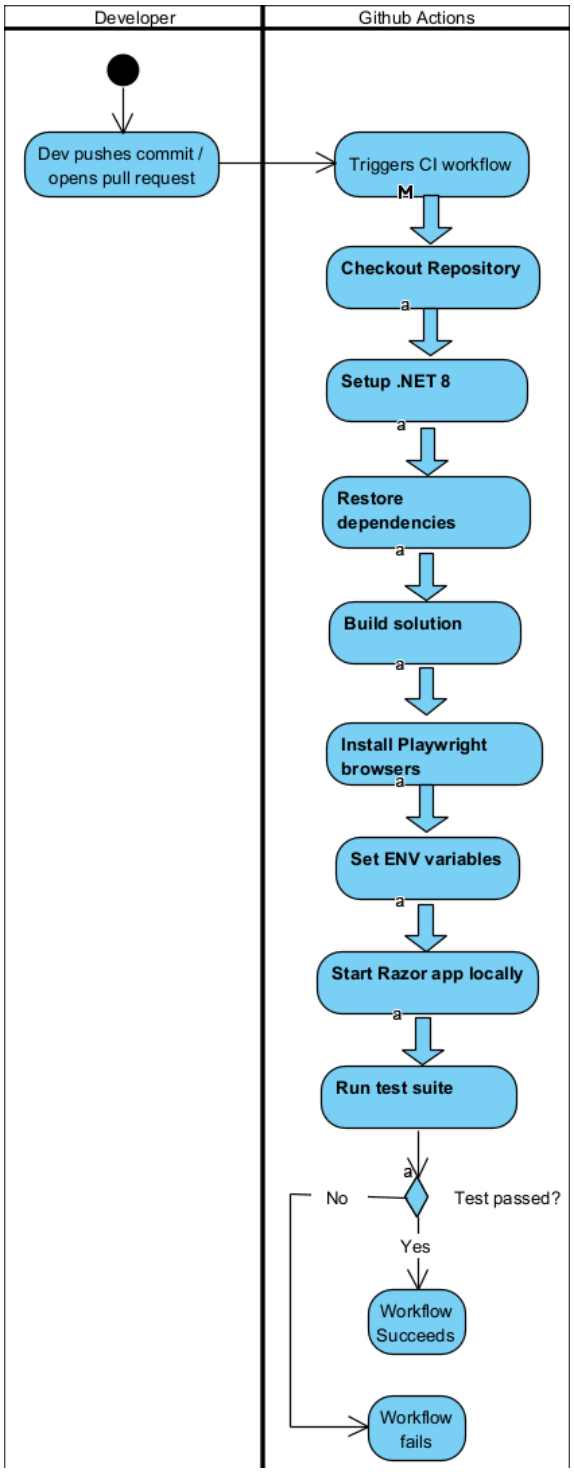

Figure 6: Illustration of the integration workflow as a UML activity diagram.

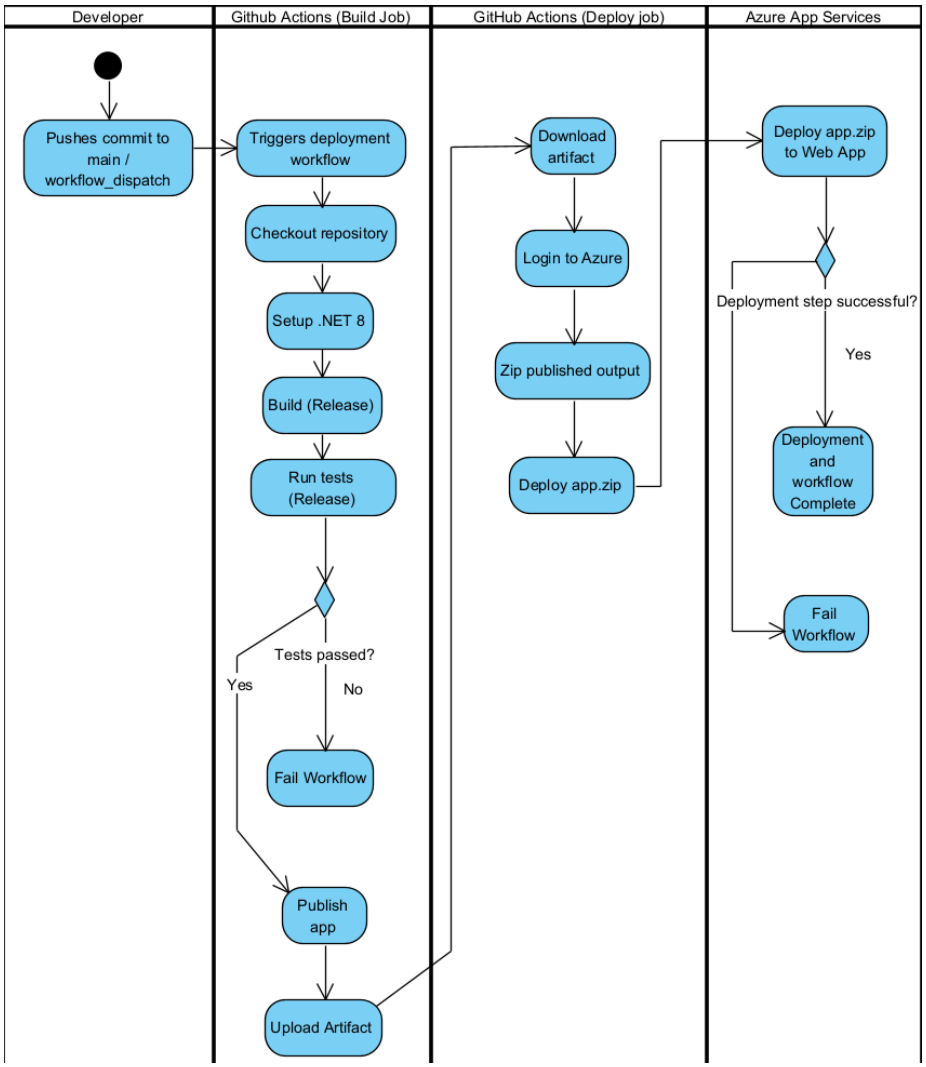

Figure 7: Illustration of the deployment workflow as a UML activity diagram.

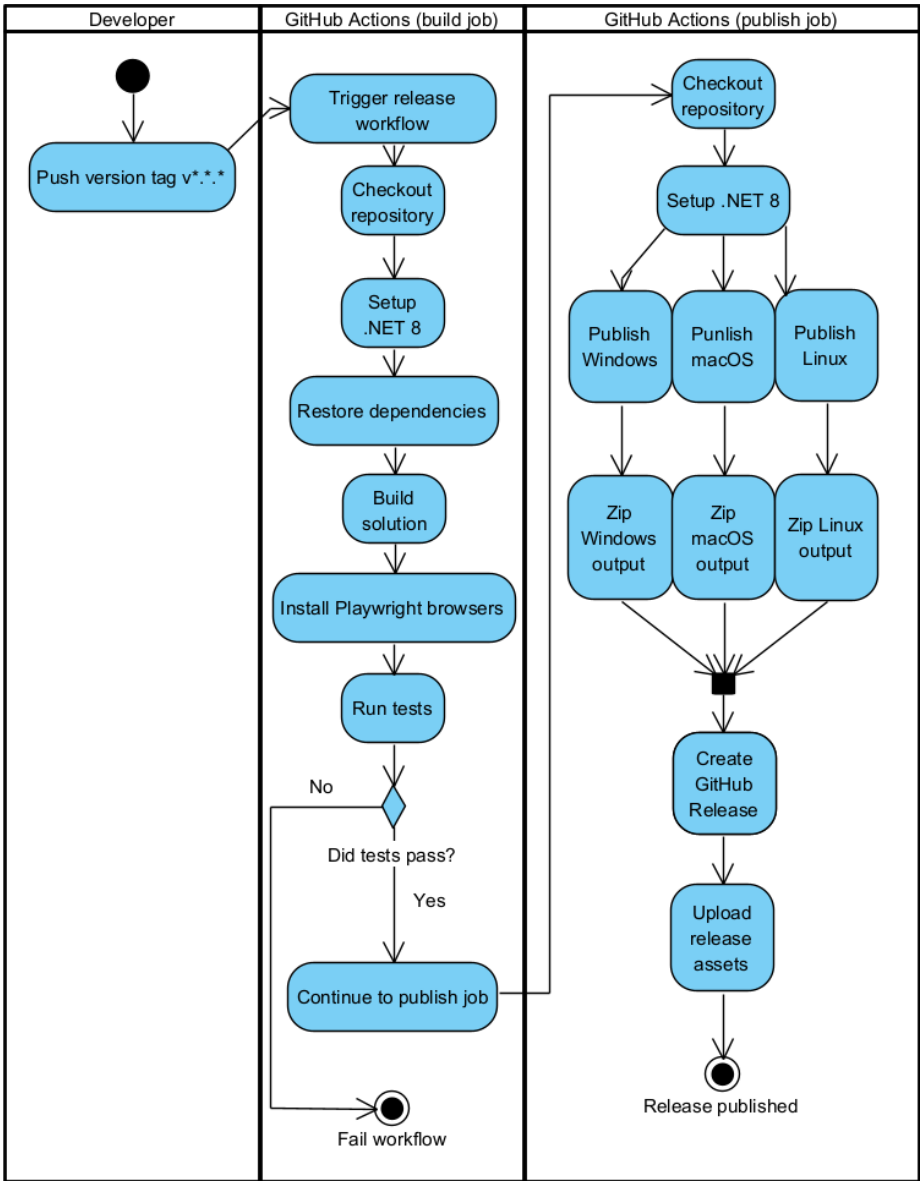

Figure 8: Illustration of the release workflow as a UML activity diagram.

- Review coordination via Discord After opening a pull request, the developer notified the team via Discord to request a review. This helped ensure timely feedback and reduced idle time.

The development process relied on team members self-assigning to open issues, which introduced certain challenges. Some team members were unable to participate for extended periods due to external circumstances, resulting in phases where active development was carried out by fewer team members than originally planned. In a more realistic project setting, a more deliberate assignment of issues would likely have ensured a more structured process and a more even distribution of the workload.

By the end of the project, only a small number of issues remained open, as the required application features had largely been implemented. Some compiler warnings remain and should ideally be addressed. Additionally, further testing and refactoring of parts of the codebase would be necessary to improve the overall quality and maintainability of the application. These potential improvements were intentionally not added as issues, as they were unlikely to be completed within the project timeframe. Figure 9 shows the state of the project board at the conclusion of the project.

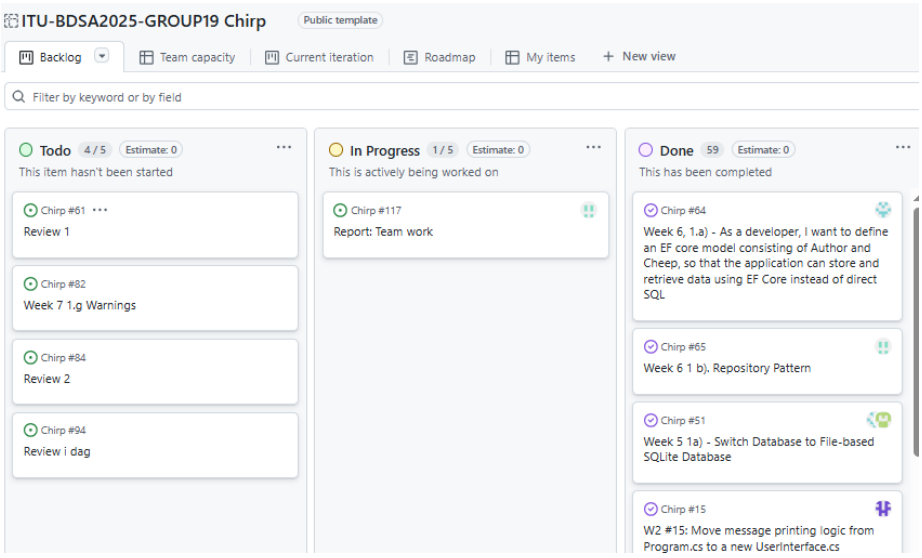

Figure 9: project board at the conclusion of the project.

2.3 How to make *Chirp!* work locally

First clone the repository by opening a command prompt and typing *git clone https://github.com/ITU-BDSA2025-GROUP19/Chirp <foldername>*. Next, navigate to the folder "Chirp.Razor" < "src" and execute the following *dotnet*

run. This will start the application and open a port on your computer. Copy the URL displayed in the terminal *http://localhost:5273* and paste it into your browser to explore the application.

2.4 How to run test suite locally

Automated UI tests for Chirp! were implemented using Playwright and NUnit. They verify core user interactions, including user registration, following/unfollowing users, and timeline behavior. The tests ensure that key functionality works as expected and that the UI reflects the correct state for different user actions. To run the UI tests locally, do the following: 1. navigate to the folder "Chirp.Razor" < "src" and execute the following *dotnet run* 2. in another terminal, navigate to the folder "Chirp.UI.Tests" < "test", and execute: *dotnet test*

The Chirp project contains multiple test suites targeting different layers of the system. Together, these tests help us provide the confidence we need that the application behaves as expected at all levels.

2.4.1 Running the test suites

All test projects can be executed using the .NET CLI.

Some of the tests can be run directly from the repo root without starting the web application, whereas other tests require a running instance of the application to work.

```
dotnet test test/Chirp.Infrastructure.Tests
dotnet test test/Chirp.Razor.Tests
```

The above commands, run from the repo root, execute quickly and do not depend on external services or a running server.

The UI and end-to-end tests use Playwright and therefore require a working running instance of the Chirp application to work. Following the steps at "How to make *Chirp!* work locally" you should have a working instance of the application and can then use the following commands for the tests dependent on the application:

' 'dotnet test test/Chirp.UI.Tests dotnet test test/Chirp.EndToEnd.Tests' "

2.4.2 Description of test suites

2.4.2.1 Infrastructure Tests These tests focus on the repository layer of the application. They use EF Core with an in-memory database to isolate each test case and ensure that core data access logic is correctly implemented.

The tests include:

AuthorRepository tests, which verify retrieval of authors by name and email, as well as enforcement of uniqueness constraints. **CheepRepository tests**, which validate that cheeps are correctly stored, returned in order, filtered by author, and that new authors are created automatically when a cheep is posted by an unknown author.

2.4.2.2 Razor integration tests These tests verify that the ASP.NET Core Razor application is correctly wired and responds to HTTP requests correctly. They use WebApplicationFactory to start the application in a controlled environment for testing.

These tests ensure that dependency injections, middleware configurations, and database setups work correctly at startup.

2.4.2.3 UI Tests The UI tests use Playwright to validate user behavior through a real browser environment. They simulate a series of realistic user interactions and thus verify visible outcomes in the rendered HTML.

The tests cover:

- Visibility of the cheep input box
- Enforcement of the 160-character limit
- User registration and login
- Follow and unfollow interactions
- Filtering of timelines
- Display of user information

2.4.2.4 End-to-end tests The end-to-end tests validate the complete system behavior across all layers.

The tests simulate situations such as:

- Logging in as an existing user
- Creating a new cheep
- Verifying that a cheep appears in the timeline
- Verifying that the cheep is persisted

These tests provide a strong level of confidence and assurance by confirming that multiple layers work together correctly.

3 Ethics

3.1 License

3.2 LLMs, ChatGPT, CoPilot, and others

LLMs have been used throughout the development of this project for a variety of tasks. The extent to which LLMs were used varied among team members,

reflecting different working styles and preferences. Below is an overview of how LLMs contributed to the project.

- **Debugging an troubleshooting.** LLMs (primarily ChatGPT and GitHub Copilot) have been particularly helpful in debugging and troubleshooting. In addition to suggesting fixes, they often explained why errors occurred. This was especially valuable when working with high-level frameworks such as EF Core and ASP.NET Identity, where error messages can be difficult to interpret. Without ChatGPT, resolving these issues would often have required extensive searching across online forums and documentation. Furthermore, GitHub Copilot's built-in analysis of pipeline error logs proved useful when CI builds failed.
- **Writing code.** ChatGPT is very keen on writing code. However, since the primary purpose of this project was educational, prompts were deliberately structured to discourage copy-paste-ready solutions. In some cases, LLM-generated code was still used—for example, when generating unit tests. In such instances, ChatGPT was credited as a co-author in the corresponding commit. In a non-educational or production setting, LLMs would likely be used more extensively for writing code.
- **Generating commit messages.** In Git Desktop the built-in Copilot has been used to generate commit messages. The commit messages have typically been used as-is or with only a few edits.
- **Personalized tutoring**. ChatGpt has been helpful in explaining concepts, answering follow-up questions, and providing feedback on ideas. The ability to iteratively ask questions and receive explanations tailored to the user's level of understanding significantly accelerated learning.
- **Writing**. Not being native english speakers, ChatGPT has been helpful in reformulating sentences so they appear more professional without "fluff".

There were occasions where ChatGpt provided bad advice. It seems sometimes it fixates on solving a specific problem by creating a very advanced solution and overlooking more obvious ones. But overall the use of LMM has sped up the deevelopment and gaining experience with these tools was itself a valuable outcome.

As discussed in the lectures, LLMs consume significant amounts of electricity and water, and their use should therefore be considered within an ethical and environmental context. However, given the growing importance of AI in software development, it is difficult to imagine a future where software developers do not rely on such tools. Choosing not to learn how to use LLMs due to environmental concerns may, in practice, exclude developers from the field altogether. While changes in individual behavior are important, they are unlikely on their own to resolve the environmental impact of LLMs — just as concerns have not eliminated air travel. The more realistic solution will likely come from future technological advances that make AI systems more energy- and resource-efficient.