Chirp! Project Report

ITU BDSA 2025 Group 13

Dimitri Alessandro Nielsen dini@itu.dk
Karl Karl Theodor Ruby krub@itu.dk
Lukas Shaghashvili-Johannessen lush@itu.dk
Matthias Nyman Nielsen mnni@itu.dk
Matthias Schoenning Nielsen mscn@itu.dk

1 Design and Architecture of *Chirp!*

1.1 Domain model

1.2 Architecture — In the small

1.3 Architecture of deployed application

1.4 User activities

1.5 Sequence of functionality/calls trough *Chirp!*

2 Process

2.1 Build, test, release, and deployment

2.1.1 Versioning

Our project used "Calender Versioning" in the beginning, before we were introduced to semantic versioning. The semantic versioning was later automated for our weekly releases, using Release Please from Google. Release Please continously looks at the commits pushed to the main branch, and updates a changelog on a separate branch. This changelog can then be published along with our actual release. This was done for convienece, as we can continue to develop the project without worring about creating our own changelog.

2.1.2 Deployment

Whenever we deploy our code to GitHub, a number of GitHub Actions scripts will be run. These can be found the .github/workflow directory. - coverage.yml:

Runs a code coverage test and fails upon not reaching the set threshold - format.yml: Runs dotnet format, which maintains a certain code standard in our code. These formatting commits have been attributed to our group member natthias - main_bdsagroup13chirprazor.yml: Builds and deploys our code to our Azure Webapp instance, using our GitHub Secrets to access login information. This file was auto-generated by Azure and afterwards customized for our needs by a group member. - release.yml: Builds and publishes our project to our GitHub repository, with builds for different operating systems.

2.1.3 Linear git history

Initially, we did not enforce a linear git history as a requirement for our project, but it was a soft requirement to attempt to keep the history linear. However, after #87381, we decided to enforce a linear history on the project through the git repo settings.

There are many reasons for enforcing a linear history. For one, having a linear history makes managing and maintaining a project easier. It is much easier to understand when a certain feature was added, as each commit is a patch, which cleanly applies atop the previous commit.

It also allows for external tools, such as `git bisect` to traverse the commit graph easier. Since a linear history is just a line, it is much easier for `git bisect` to perform a binary search and find when a regression was introduced. When the commit graph is a large interleaving of commits and merge commits, it can be difficult to perform a binary search. Since the commit graph of a project with a linear graph is a line many tools, such as `git bisect`, which uses binary search to find when a regression was introduced

It also allows for easier code review. Since each commit cleanly applies atop the previous, there is no extra noise generated by merge commits. It is simply a set of patches to apply to the main branch.

Finally, one of the primary reason we used a linear git history was for external tooling. We used an action called release-please for automated changelog generation. This tool recommended a linear history to ease parsing of the commit graph[1].

The workflow for a linear git history is rather simple. When a commit is ready to be merged with the main branch, you `git rebase origin/main`, changes the base of the branch to be the current *HEAD* of the main branch. This might introduce merge conflicts, which you resolve as normal. Once the conflicts have been resolved, and the PR has been approved, it can be merged with `git merge --ff-only`.

[1]Release Please documentation about linear history

2.2 Team work

2.2.1 Trunk-based development

2.3 How to make *Chirp!* work locally

2.4 How to run test suite locally

3 Ethics

3.1 License

3.2 LLMs, ChatGPT, CoPilot, and others