# GitTor

Design Document

Team #15
Simanta Mitra - Client
Isaac Denning - Team Lead
Phu Nguyen - Web Application Lead
Cameron Gilbertson - CLI Application Lead
Tyler Gorton - Record Keeper
Jayson Acosta - Quality Assurance Lead
Seth Clover - Technical Research Lead
sdmay26-15@iastate.edu
https://sdmay26-15.sd.ece.iastate.edu

Revised: 12/3/2025

# Executive Summary

A decentralized Git hosting platform where repositories are synchronized between users via a P2P protocol using torrenting. The purpose of this application has been to eliminate the issue of a single point of failure, which affects other platforms such as GitHub and GitLab. The problems that users identified with centralized platforms were as follows:

1. If the hosting platform goes down, there is no automatic fallback, meaning collaboration on a project is completely halted until an alternative means of sharing is implemented. The switch from one hosting platform to another can take anywhere from a few hours to a few days, depending on the scale and complexity of a project.
2. When cloning a repository, the hosting platform can trivially add/remove any commits from the top of the log without it being obvious. The "author" and commit messages can be identical to the actual state of the repository, with the only difference being the repository content and commit hashes. Something that, in large projects, is rarely checked in depth.
3. By default, Git uses SHA-1 for commit hashes, which has been broken. With enough time and computing power, both of which Microsoft (the owner of GitHub) has, a commit hash collision could be found to inject malicious code in the middle of a project's history. This tactic would be even more challenging to identify than the last, since it can occur at any point in the project's log, not just at the top, and the only difference would be the repository contents.
4. Finally, when the hosting platform is the only means of collaboration, they are the judge, jury, and executioner. Even if they don't change the state of the repositories, they can decide to block one's commits, make malicious commits in a contributor's name, or even delete repositories (ideally, there would still be an instance on someone's machine).

To address all of these issues, we minimize the power of the hosting platform as much as possible. Instead of hosting repositories through GitHub or GitLab, everyone who contributes to it hosts the repository and shares it with others via a P2P protocol. When sharing the repository, GitTor seeds it with a .torrent file and by sharing this .torrent file with someone else, they are able to leech the repository, downloading it directly from the other contributors.

This strategy clearly resolved issue one; even if one computer is off, others are seeding the repository, allowing anyone with the .torrent file to leech it. The solutions to problems two and three are less clear. The .torrent file contains the hash of the contents, among other things. By using this to share repositories, any peer that downloads the repository can verify that every byte matches the original, no matter which seeder it came from. Lastly, we must address issue four. Since the contributors are the ones hosting the repository, they — as a collective — are the judge, jury, and executioner. They control the state of the repository, and it is up to the other contributors to decide if they wish to leech anyone's changes and work off of them.

However, this approach was not without some of its own concerns that need to be addressed. The primary problem with this approach was how to share changes to a repository among the contributors. Since any change to the repository would result in a different .torrent file that needs to be shared with all contributors. While this could be done by any means, email, paper, or even carrier pigeon, none of these would be convenient for GitTor users. Instead, we created a web

application that allowed users to find, view, and obtain .torrent files for repositories. The difference between this and a typical hosting platform is that its power is significantly limited, as it has no control over the repository contents. Additionally, a simple, documented API is available, allowing alternative web applications to be easily created and switched to in minutes.

Another concern was that anyone with access to the repository could modify it, and there would be no way to restrict write access to verified contributors other than through the web application, which we wanted to limit its power as much as possible. To ensure that only authorized individuals can commit to a repository, a file in the repository stores the public GPG keys of those allowed to commit, which is used to verify the validity of the repository. To add another contributor, an already verified contributor must add the new contributor's GPG key to this file. When a new GitTor repository is initialized, it initially contains only the creator's GPG key.

To summarize, the overall structure of this project is two parts:

1. A command-line tool that allows users to easily publish the new state of their repository and seed it for other users to leech. The command-line tool also enables users to validate the commits via their GPG signatures and the valid contributor's keys file.
2. A web application for users to publish their .torrent files to and see other users' repositories with contents and pull requests. The application should also validate the signatures when people publish a new .torrent file and start seeding the repository itself.

As for the tools we used:

- CLI:
    - C for the source code
    - Many C libraries, such as libcurl, glib, argp, libtorrent, etc.
    - Unity C testing library with Gcovr coverage reports and Valgrind memory analysis
    - CppLint and Clang-Tidy
    - Pipeline for rule checks, tests, reports, and builds
- Web App:
    - API:
        - Spring Boot Java source code
        - JUnit integration and unit testing with JaCoCo coverage reports and context profiling
        - Autogenerated OpenAPI specification with documentation
        - Checkstyle rules
    - UI:
        - Angular 21 source code
        - Nginx runtime with reverse proxy to API
        - Cypress end-to-end testing
        - Tailwind styling with ZardUI component library
        - Autogenerated API connection code via OpenAPI specification
        - Prettier and ESLint rules
    - A PostgreSQL relational database
    - A MinIO Simple Storage Service
    - Docker Compose for a Dockerized application stack
    - Pipeline for rule checks, tests, reports, and builds

# Learning Summary

## Development Standards & Practices Used

- RFC 9113 - Hypertext Transfer Protocol -- HTTP/2
- RFC 8446 - The Transport Layer Security (TLS) Protocol Version 1.3
- RFC 9580 - OpenPGP
- OCI runtime-spec - Open Container Initiative Runtime Specification
- OCI image-spec - Open Container Initiative Image Format Specification
- OCI distribution-spec - Open Container Initiative Distribution Specification
- ISO/IEC 9075 - Information technology — Database languages SQL
- ISO/IEC/IEEE 29119 - Software and systems engineering — Software testing

## Summary of Requirements

- GitTor must share repositories via a P2P protocol
- GitTor must not cost any money for users to be able to access/run
- GitTor must not share the contents of your repository with any third party, unless configured by the user to do so
- GitTor must be capable of leeching a repository even when only one seeder exists
- GitTor must be capable of verifying that the commits on a repository have come from authorized contributors
- GitTor must use the GPG protocol for committer authentication
- GitTor must be able to share repositories without using the web application
- The web application must only require one dependency, Docker, to be runnable
- GitTor CLI must be functional on linux
- GitTor web app must be usable and aesthetic on all web platform screen sizes (phone, tablet, personal computer)
- GitTor web app must have a consistent design between pages
- GitTor web app must have dark-mode
- GitTor web app's nested pages are limited to 3 pages deep
- GitTor CLI must use a standard design help menu

## Applicable Courses from Iowa State University Curriculum

SE 3090: Software Development Practices
COMS 3270: Advanced Programming Techniques
COMS 2520: Linux Operating System Essentials
COMS 3630: Introduction to Database Management Systems
SE 3190: Construction of User Interfaces
SE 3170: Introduction to Software Testing

## New Skills/Knowledge acquired that was not taught in courses

Our project involved numerous systems that none of our team members had previously worked with, such as many of our C libraries, the Unity test framework, CppLint, Clang-Tidy, and the Zard-UI components library. Additionally, our project provided all our team members with the opportunity to delve deeper into the Git protocol, a system we had all used but did not have as deep of an understanding with as we do now.

Many of our team members also chose to work on this project as an opportunity to learn Angular since Iowa State's frontend design classes were taught in React.

# Table of Contents

# List of figures/tables/symbols/definitions

# 1. Introduction

## 1.1. Problem Statement

Currently, software development on a team typically involves the following process:

1. Code is stored online through a platform like GitHub or GitLab
2. Members of the team who wish to contribute download the project via this platform
3. They make the desired changes and upload those changes back to GitHub or GitLab

There are many other aspects to modern software development, but for now, these are all that need to be considered. This process seems simple and effective; however, it comes with a lot of trust and reliance on the hosting platform, GitHub or GitLab. What if they stop working and we can no longer share our code? What if they choose to block a team member from contributing? What if they decide to change our team's code without our permission? For those who understand the underlying Git protocol, they may believe that GitHub or GitLab cannot change a project's code without it being obvious; however, this is far from the truth. There are actually multiple ways that platforms like GitHub or GitLab can modify your team's code without it being obvious. These methods can be hard or impossible to describe without first understanding the Git protocol, but for those who already understand Git, these are the two methods:

1. The hosting platform can add/remove commits from the top of the log with the same "author" and commit message as the original, yet with different content and hashes.
2. With sufficient time and computation, the hosting platform can find a SHA-1 commit hash collision. With this, the hosting platform can inject an almost identical commit anywhere in the log, making it even more challenging to find.

To answer the question, "What if my project's hosting platform does...?" Once one of these malicious acts is identified, which may be easier said than done, the team behind this project would need to switch from their current hosting platform to a different one to avoid this issue from continuing. For simple projects, this can be done in a matter of minutes, making it almost a non-issue. However, for large-scale systems, many aspects are tied to their hosting platform, not just the code itself. Because of this, a switch from one hosting platform to another can take anywhere from hours to days. On many professional projects, this amount of downtime without collaboration can cost thousands of dollars and is simply unacceptable.

Another issue with the current state of software development is that the online hosting platforms can read your entire project. On most projects, this is not a concern; however, some have policies preventing information from reaching third parties. In these cases, the solution is quite simple: self-host an instance of GitLab. By doing so, your project never reaches the hands of a third party; however, this can be quite the hassle and requires running a server constantly.

Our project resolves all of these issues. Instead of hosting your project through GitHub or GitLab, you and everyone on your team hosts it and shares it with others via a peer-to-peer protocol. In this system, when a new team member wants to download the project, they can do so directly from other team members. Even if one computer stops working or turns off, the others will still be hosting the project, allowing it to continue being shared. Additionally, the identification for projects, when downloading, include a built-in digital fingerprint that can be used to verify that the contents of the downloaded project have not been tampered with.

As for the issue of the hosting platform blocking a team member from contributing. With GitTor, the only way this can happen is if everyone else on the team unanimously decides not to accept the contributions of one team member. However, this seems like more of an internal team conflict than anything else.

### 1.2. Intended Users

This project can be used by anyone who wants to share their Git projects with others, and all of them would have something to gain. However, there are three specific categories of users that our project is targeting:

Open-Source Teams: These are volunteer maintainers scattered across countries and time zones. They need a free collaboration system that guarantees anyone with authorization can contribute. They would benefit from our system because it is free, with the hosting workload distributed across the network, and the ability to contribute is inherent to the system. While most hosting platforms are currently free to use, this may change in the future, whereas our system cannot have a financial aspect added.

Privacy-Focused Enterprises: These are corporations that handle sensitive information, protected by policies such as HIPAA. They need a collaboration system that does not provide their information to a third party. They would benefit from our system because projects can be shared directly between team members without any third-party access. Unlike current online hosting platforms, where, even when marked private, all the information can be read by the hosting platform itself, which may violate policies.

High-Value Systems: These are applications with large customer bases and high expectations for uptime. They need a collaboration system with a guarantee of no downtime, allowing updates at any time; otherwise, a significant amount of money will be lost. They would benefit from our peer-to-peer approach since each peer provides an independent layer of redundancy, preventing collaboration from ever halting. Systems like GitHub do have redundancy in availability zones; however, they are not independent and always have a chance of downtime if a bug is introduced.

## 2. Requirements, Constraints, And Standards

### 2.1. Requirements & Constraints

#### 2.1.1. Functional Requirements

- GitTor must share repositories via a P2P protocol
- GitTor must not cost any money for users to be able to access/run *(constraint)*
- GitTor must not share the contents of your repository with any third party, unless configured by the user to do so *(constraint)*
- GitTor must be capable of leeching a repository even when only one seeder exists *(constraint)*
- GitTor must be capable of verifying that the commits on a repository have come from authorized contributors
- GitTor must use the GPG protocol for committer authentication

- GitTor must be able to share repositories without using the web application

#### 2.1.2. Resource Requirements

- The web application must only require one dependency, Docker, to be runnable *(constraint)*
- GitTor CLI must be functional on linux

#### 2.1.3. Aesthetic Requirements

- GitTor web app must be usable and aesthetic on all web platform screen sizes (phone, tablet, personal computer)
- GitTor web app must have a consistent design between pages
- GitTor web app must have dark-mode
- GitTor web app's nested pages are limited to 3 pages deep *(constraint)*
- GitTor CLI must use a standard design help menu

### 2.2. Engineering Standards

#### 2.2.1. Importance

Arguably, the most significant importance of engineering standards is establishing an agreed-upon means of collaboration. It's perfectly fine for an isolated system to use its own protocols, but as soon as it needs to interact with other systems, both systems must have an agreed-upon means of doing so. This problem is where engineering standards come in. They establish the agreed-upon means of collaboration that can be applied to all systems. In this way, if a new system wants to collaborate with all others, it only has to implement a few protocols, rather than a specific one for each system.

There are other benefits to engineering standards, like safety and usability, but for us programmers, this is the most important aspect.

#### 2.2.2. Relevant Standards

- RFC 9113 - Hypertext Transfer Protocol -- HTTP/2
  HTTP/2 is the second major version of the core communication protocol of the web. Overall, this protocol defines how clients and servers exchange requests and responses on top of the TLS or TCP protocol. This version intends to improve upon its predecessors by being more efficient with its use of network resources and reducing latency. It accomplishes this by updating the HTTP standard to support field compression concurrent exchanges on the same connection.
  Our application utilizes HTTP/2 for all communication between the CLI and web applications, as well as for requests between the UI and API.
- RFC 8446 - The Transport Layer Security (TLS) Protocol Version 1.3
  TLS is an encryption standard for client/server communication that prevents eavesdropping, tampering, and message forgery from anyone but the two intended parties. Version 1.3 is an improvement in that it is faster, utilizes a more secure cryptographic method, and encrypts a greater portion of the initial handshake process.
  All our HTTP requests in GitTor are built on top of TLS to provide necessary encryption.
- RFC 9580 - OpenPGP

OpenPGP is a standard of internet encryption that includes tools like GPG. GPG specifies how to generate a public-private key pair, create a digital signature of authenticity, and how to validate those signatures.
Since the Git protocol already integrates GPG signatures, we utilize this feature to authenticate and authorize commits.

- OCI runtime-spec - Open Container Initiative Runtime Specification
  This specification defines the configuration, execution environment, and lifecycle of a container. It ensures that applications running inside a container have a consistent environment, regardless of the machine on which they are running.
  Since Docker runs on the Open Container Initiative, with layers of abstraction, we rely on this standard to host the web application images.
- OCI image-spec - Open Container Initiative Image Format Specification
  This specification defines the structure of an OCI image, which stores all the information needed for the runtime to create, start, and stop a container running the application.
  Since Docker runs on the Open Container Initiative, with layers of abstraction, we rely on this standard to create the web application images.
- OCI distribution-spec - Open Container Initiative Distribution Specification
  This specification defines how OCI images can be shared between systems through a push and pull schema. It also describes how images can be identified through tags to facilitate easier retrieval.
  Since Docker runs on the Open Container Initiative, with layers of abstraction, we rely on this standard to retrieve images to build off of when creating our own.
- ISO/IEC 9075 - Information technology — Database languages SQL
  This standard provides a set of rules for the Structure Query Language (SQL) used often in relational databases. The standard mainly specifies what syntax, semantics, data structures, and behavior that SQL should demonstrate. This provides consistency on what database service or company is using SQL.
  In our project, we use Postgres for our database, and that uses SQL, so we need to follow this standard to get uniform SQL.
- ISO/IEC/IEEE 29119 - Software and systems engineering — Software testing
  This standard is meant to provide a framework for the structure of software testing. Specifically, it aims to provide a clear layout for consistency, quality, and transparency for how testing is planned, designed, and reported.
  Throughout our project we have kept this standard in mind in order to create a comprehensive test suite with over four hundred tests to date.

# 3. Project Plan

## 3.1. Project Management/Tracking Procedures

For our project, we employed an agile management style to deliver working software early and often, which enabled us to refine features and respond quickly to issues as they arose. This style also ensured that both our CLI and web application evolved together as new requirements and challenges emerged.

To track our progress, we used a GitHub Kanban board, which integrated well with our GitHub repositories, linking branches and pull requests. This Kanban board included task stages: draft, to-do, in progress, in review, and done.

## 3.2. Task Decomposition

To solve the problem at hand, we continuously broke down the development of GitTor into multiple tasks and subtasks, with interdependencies that enabled smooth collaboration among team members. Due to our Agile structure, we didn't have our project decomposed into all of its tasks and subtasks from the beginning. Instead, this was to be done iteratively throughout the development cycle.

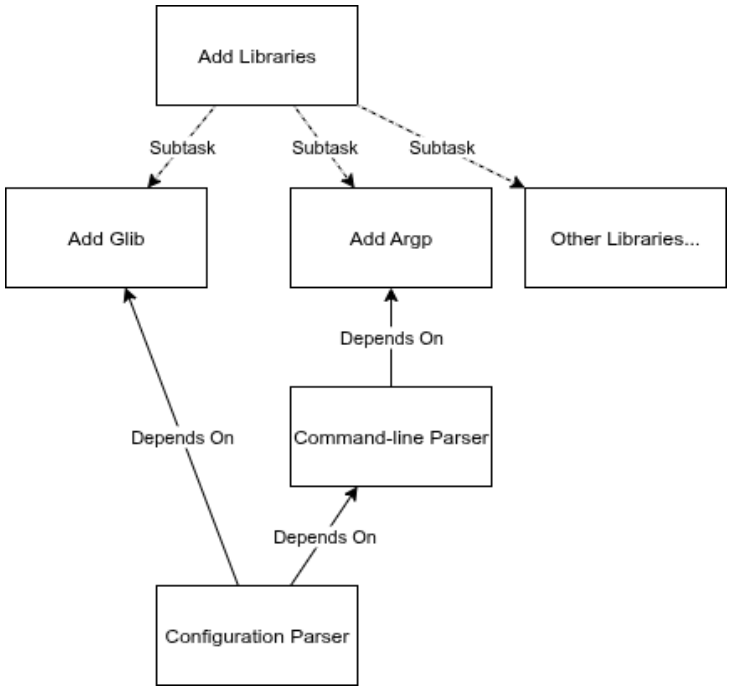


***Figure 1.*** *Example task decomposition.*

An example of this task decomposition can be seen in the implementation of our configuration file parser in the CLI. Since the parser utilized the Glib library, this task depended on adding Glib, which was a subtask of adding necessary libraries. There also needed to be a way to call the configuration file parser, which required the command-line parser that used the argp library. Once again, adding this library was a subtask of adding necessary libraries.

## 3.3. Project Proposed Milestones, Metrics, and Evaluation Criteria

Our project was broken down into four major milestones that we would be working towards throughout the year.

- **Foundations** - Build out the project environments with all known dependencies integrated. This requires a command-line parser for the CLI, as well as the addition of dependent libraries. The web application requires a connected Docker Compose structure with all containers collaborating properly.
- **Seeding and Leeching** - The API must enable the upload and retrieval of torrent files, and the CLI must utilize these endpoints to seed and leech the repository properly. The UI must be sufficiently developed to allow users to access repositories.
- **Security** - The CLI and API integrate GPG checks to validate that all commits originate from authorized contributors. The web application enables different levels of repository visibility. The UI continues to be developed with additional functionality, including the ability to view repository contents online.
- **Polish and Usability** – The CLI and web application have been refined to ensure that all required functionalities are implemented effectively, providing a smooth and user-friendly experience.

## 3.4. Project Timeline/Schedule

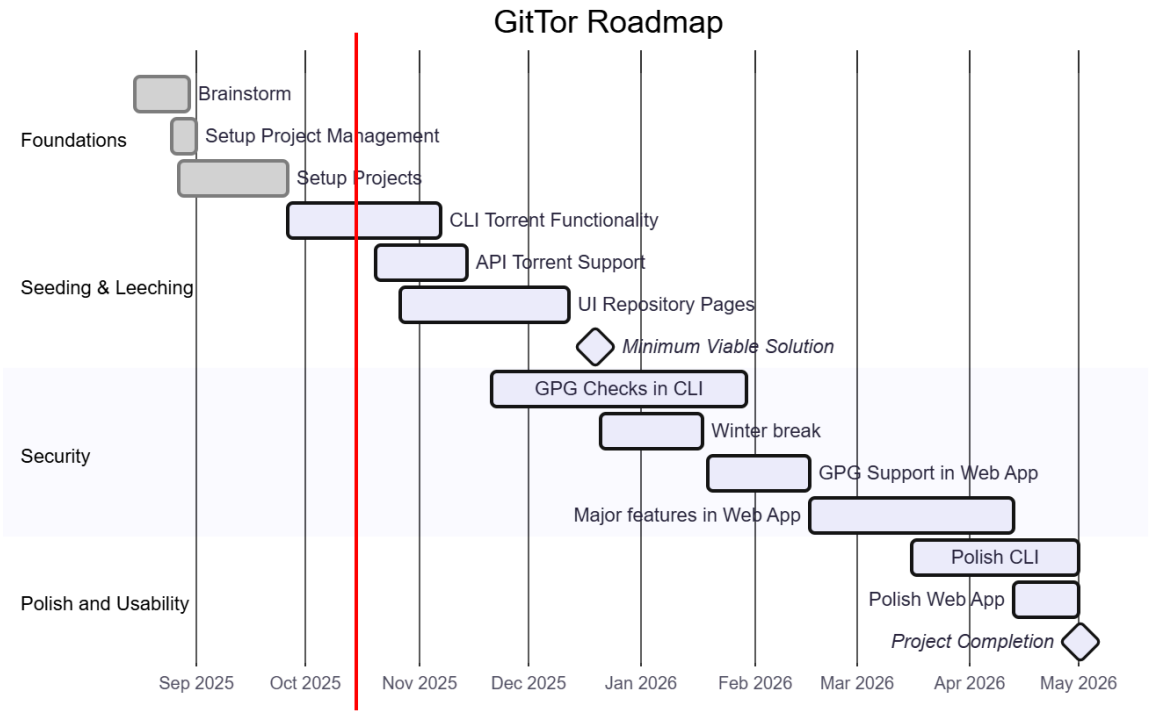


***Figure 2.*** *Gantt chart.*

## 3.5. Risks and Risk Management/Mitigation

| Risk | Probability | Severity | Mitigation Strategy |
|---|---|---|---|
| *CLI Torrent Functionality* | | | |
| Torrent library incompatibility | 0.2 | High | Research and experiment with |

| | | | |
|---|---|---|---|
| | | | multiple torrent libraries |
| Performance issues with large repositories | 0.4 | Medium | Implement performance benchmarks during early development.<br>Investigate optimization techniques |
| *API Torrent Support* | | | |
| API Performance Bottleneck | 0.3 | Medium | Consider scaling strategies (caching layer, load balancing) |
| Incompatibilities with CLI | 0.25 | Medium | Establish versioning strategy |
| *UI Repository Pages* | | | |
| UI/UX Complexity | 0.5 | High | Create wireframe plan before implementation |
| *GPG Checks in CLI* | | | |
| Unexpected Integration Complexity | 0.4 | High | Prototype GPG integration early |
| *GPG Support in Web App* | | | |
| Secure Key Handling in a Web Environment | 0.6 | High | Follow strict security protocols for server key storage and handling |

***Table 1.*** *Risk management and mitigation.*

Inarguably, our biggest single "issue" that took up the most time by itself was getting Windows support for the command line interface. However, this is misleading, since we never specified this as a requirement, knowing it would very well cause headaches in the future and may even cause us to drop Windows support. As such, we never put it on our risk chart since we knew that if it ever became too much of a bottleneck, we could give up Windows support and continue.

Beyond that, we were very successful at avoiding large-scale issues that could have slowed our project by keeping the big picture in mind and tackling complex issues as early as possible. In doing this, we had already decided how almost every area of our codebase would work after the first few months; beyond that, we still needed to implement them, though.

## 3.6. Personnel Effort Requirements

| **Major Task** | **High-Level Description** | **Related Sub Tasks*** | **Time Estimate** | **Time Actual** |
|---|---|---|---|---|
| Brainstorm | Beginning Stages of the GitTor project. Laid out our project design | - Come up with ideas<br>- Pitch our ideas to | 20 hours | 20 hours |

| | | | | |
|---|---|---|---|---|
| | | faculty | | |
| Set up Project Management | Setting up our project environments | - Set up a GitHub Repository<br>- Set up our Kanban and Discord | 15 hours | 15 hours |
| Setup Projects | Setting up the framework for GitTor | -Setting up the framework for the CLI<br>- | 30 hrs | 30 hrs |
| CLI Torrent Functionality | Develop a working CLI that can functionally work as a Git service | -Research Torrent Integrations<br>- Torrent Demo | 100 hrs | 150 hrs |
| API Torrent Support | Develop a working API to be able to call our CLI | - Create a design for the API with methods<br>- Implement the design to work with our CLI | 60 hrs | 80 hrs |
| UI Repository Pages | Develop a working UI for our repository for the WebApp | - Choosing a UI<br>- Implementing basic functionality for users to use the UI Repository Page (such as login) | 90 hrs | 100 hrs |
| Minimum Functioning Implementation | System Integration between the UI and the CLI. | - Have a functioning CLI<br>- Have a functioning WebApp<br>- Integrate the CLI and WebApp together | Aggregate of previous = 315 hrs | Aggregate of previous = 395 hrs |
| GPG Checks in the CLI | Enable GPG Checks for the CLI | - Parse GPG signatures file<br>- Validate GPG signature<br>- Backtrack through commits validating | 70 hrs | 100 hrs |
| GPG Support in the WebApp | Enable GPG Support for the WebApp | - Parse GPG signatures file<br>- Validate GPG signature<br>- Backtrack through commits validating | 60 hrs | TBD |

| | | | | |
|---|---|---|---|---|
| Polish CLI | Finish the CLI while creating readable and maintainable code | - Complete the CLI<br>- Add READMEs<br>*TBD | 80 hrs | TBD |
| Polish Web App | Finish the WebApp while creating readable and maintainable code | - Complete the WebApp<br>- Add READMEs<br>*TBD | 200 hrs | TBD |
| Documentation | Create final documentation pieces | - Create a documentation explaining GitTor in either an extra document or as a Wiki | 30 hrs | 30 hrs |

***Table 2.*** *Major task outline.*

For the first few major tasks, our time estimates were perfect because we had already completed them. After that, we went over time on every task because we did not expect the effort each step would require. While we knew the overall design and understood what would be needed to complete these tasks, we did not expect how much time they would take.

It is always hard to estimate time, even when there are no hiccups in the process; underestimates still arise. To mitigate these issues, we should have added a buffer to every task beyond our original estimate.

### 3.7. Other Resource Requirements

Our project is entirely software-based, and we don't require any physical resources. We are hosting the web application on one of the team member servers, although that is not a requirement.

# 4. Design

## 4.1. Design Context

### 4.1.1. Broader Context

The GitTor project aims to give programmers a decentralized, secure, and robust way to share their code with others. GitTor is aimed towards programmers as an alternative to major repository hosting platforms, such as GitHub and GitLab, as our project removes the need for a centralized server owned by a large corporation like Microsoft. The removal of the centralized server also creates layers of redundancy and removes a single failure point, which many other repository hosting sites suffer from.

| **Area** | **Description** | **Examples** |
|---|---|---|
| Public health, safety, and welfare | Our project provides a secure way for users to store and share their code | • HTTPS communication between the server and users |

| | | |
|---|---|---|
| | repositories. This will provide peace of mind that their code is safe and secure. | • P2P secure communications between users<br>• GPG Checks for Repository Editing |
| Global, cultural, and social | Our project is open source, so this provides users the ability to modify and tailor our product to their specific needs. We believe this better aligns with the beliefs of future users. | • Open Source Software<br>• Decentralized Network |
| Environmental | N/A | N/A |
| Economic | Our project is an open-source software product. This will make it available to hobbyists as well as professionals. | • Our product is free to use |

***Table 3.*** *Societal considerations.*

#### 4.1.2. Prior Work/Solutions

Our project is an alternative to common repository hosting platforms such as GitHub [15] and GitLab. Ours will be a decentralized version that will use P2P connections to share files. According to Coursera, "P2P networks are useful for applications that require decentralized collaboration, resource sharing, or secure and transparent transactions." [13]. Using this protocol prevents the need for a centralized server that needs to be controlled and managed.

One project that is similar to what we are doing is a product called Radicle [14]. Radicle is also a decentralized P2P repository sharing platform. Our project is different from Radicle because we are going to have extra features, like a usable GUI that the user can use instead of the CLI. Also, our project will have a webpage that will provide users with a different way to look over their repositories. Also, GitTor will have Windows support, unlike Radicle.

#### 4.1.3. Technical Complexity

GitTor's multi-layered architecture involves multiple domains of software and computer engineering across the CLI and web application components, each leveraging a unique set of principles. The CLI application requires low-level systems programming in C to implement BitTorrent protocol integration, cryptographic verification using GPG signatures, and persistent background service management for seeding operations. The web application employs distributed systems principles through its service-based architecture with loosely coupled components: an Angular frontend implementing reactive UI patterns, a Spring Boot API handling RESTful state management, PostgreSQL managing relational data, and MinIO providing scalable object storage. Additionally, the security model itself presents considerable challenge, as it replaces traditional centralized access control with a cryptographic chain of trust using GPG signatures, eliminating a single point of failure found in current centralized platforms.

### 4.2. Design Exploration

#### 4.2.1. Design Decisions

A few key design decisions have significantly shaped the GitTor project architecture. First, we chose to implement the CLI in C over higher-level languages to minimize resource overhead and

maximize compatibility with existing Git tooling, though this increased development complexity. Second, we decided to separate the seeding functionality into a persistent background service rather than embedding it in the CLI process, which enabled continuous repository seed availability even when users aren't actively running commands but added inter-process communication challenges. Third, we adopted Docker Compose for the web application deployment to ensure consistent environments and simplify scalability, though this introduced a dependency that some developers may find cumbersome compared to native installation methods. Each decision involved trade-offs between usability, performance, and implementation difficulty that we evaluated based on our target user needs and project constraints.

#### 4.2.2. Ideation

Several distinct options were evaluated against the functional and technical needs outlined previously. We considered embedding seeding as a periodic operation within the CLI, implementing a persistent background service to handle seeding independently of the CLI, running seeding actions at fixed intervals through scheduled system services such as cron or systemd, requiring users to manually initiate seeding via the CLI, and offloading the seeding operations to a web server that would forego the project's Peer-To-Peer principles. Each approach was weighed based on its feasibility of implementation and user convenience, while considering maintaining a decentralized architecture and managing development complexity.

#### 4.2.3. Decision-Making and Trade-Off

The qualitative differences between the brainstormed options rested on four main factors: availability, user effort, complexity, and the ability to maintain a decentralized platform. The analysis showed that persistence, which is the ability for seeding operations to always be available, was integral for GitTor to be a reliable Peer-To-Peer repository distribution system. Reducing user effort was also key, since requiring manual intervention or complex setup would deter users from switching from other established platforms that do not suffer from those problems. Accepting a slightly higher degree of implementation complexity was deemed justifiable if it allowed for other categories of consideration to be accounted for, because the current scope of the project has allowed for an expansion of its expectations after discussion with our project advisor and client. Although embedding seeding in the CLI or relying solely on scheduled system services would be easier for user comprehension, they either lacked persistence or introduced more work for users or increased the potential for user error. Centralizing seeding in the web application directly opposes the project's core goals, so it was also dismissed. Based on these considerations, we chose to implement seeding as a persistent background process independent of the CLI. This solution introduces greater technical complexity but aligns best with the open-source, peer-powered vision for the project.

### 4.3. Final Design

#### 4.3.1. Overview

GitTor is built from two main parts: a command-line tool and a web application. The command-line tool allows users to share repositories directly. The web application helps users find the latest repository states. The command-line tools communicate over a Peer-To-Peer BitTorrent protocol, while the web application connects using HTTPS through a REST API [8], [9].

The command-line tool depends on a parser to read user input and trigger specific operations. Each operation manages different aspects of repository sharing, configuration, and authentication. The design focuses on keeping the tool lightweight and direct. It supports sharing repositories over a Peer-To-Peer network without unnecessary features. The goal is to provide essential functionality without overcomplicating the user experience.

The web application delivers a broader experience for users. It enables users to search for repositories, explore code bases, and view pull requests. The front end is built with Angular and hosted on Nginx, which also proxies requests to the API. The API runs with Spring Boot and connects to a PostgreSQL database to handle structured data [7]. For larger or unstructured data, such as repository previews, the system uses Minio Simple Storage Service.

All components run inside Docker containers managed with Docker Compose [10], [11], [12]. This setup isolates services, simplifies deployment, and improves scalability. The design creates a clear separation between the user interface, API, and storage, ensuring the system remains organized and efficient.

### 4.3.2. Detailed Design and Visual(s)

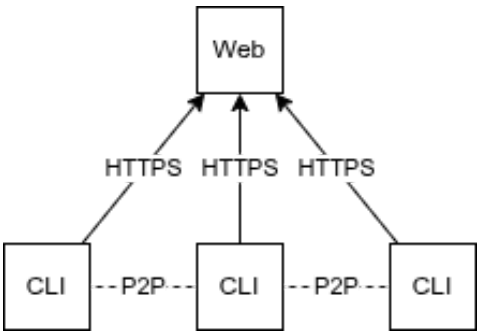


***Figure 3.*** *High level communication diagram.*

To understand the design of GitTor, we must start at a high level and gradually work our way down through the components. At the top level, GitTor can be thought of as two parts: a command-line tool that allows users to share repositories, and a web application that enables users to find the latest state of repositories. Communication between different command-line tools will be facilitated via a Peer-To-Peer BitTorrent protocol, and communication with the web application will be over HTTPS with the REST API.

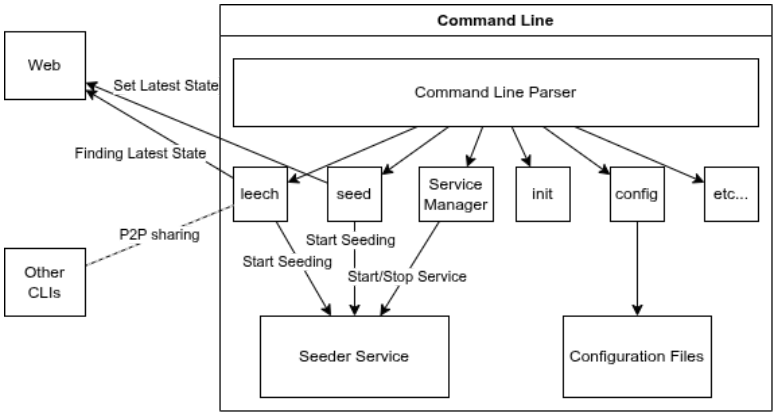


***Figure 4.*** *CLI system architecture diagram.*

Within the command-line tool, the true complexity of the application begins to reveal itself. Firstly, a command-line parser is used to interpret the user's input and invoke the necessary functionality, which has been split into multiple components. The most important of these components are the leech, seed, and service manager.

- The leech component determines the latest state of the repository using the web API, retrieves it from other contributors, and then instructs the seeder service to begin seeding.
- The seed component updates the state of the local repository according to the current changes, tells the seeder service to start seeding this new state, and then informs the web API of the state.
- The service manager starts and stops the seeder service, which is a separate process from the command-line tool.

To understand why we need this separate process, we must first realize that the basic command-line program must start and stop regularly as users call it. However, to properly seed repositories on a Peer-To-Peer network, there must be a process actively seeding at all times.

Along with all these components of GitTor's command-line tool, there are many others to support the different sub-commands of GitTor and their various options.

| **Sub-Command** | **Description** | **Git Equivalent** |
|---|---|---|
| Init | Initializes a new GitTor repository in the current directory. | Init |
| Leech | Downloads a repository from the torrent network. This can be either creating a new repository with an identifier or updating the existing repository in the current directory. | Clone / Pull |
| Seed | Uploads the repository to the torrent network so that others may leech it. | Push |
| Devs | Manages what developers are allowed to contribute to this repository. | N/A |

| | | |
|---|---|---|
| Verify | Verifies that all commits to this repository came from authorized developers. | N/A |
| Config | Read and write local and global GitTor configurations. | Config |
| Login | Authenticate user with the web application | N/A |

***Table 4.*** *CLI sub-commands.*

After reviewing the design for the command-line tool, one might assume that the UI must only support retrieving, uploading, and updating repositories, as well as maybe some form of authentication. If this were the case, the API could be implemented in only a few lines of code. However, this would not provide a well-rounded experience for most of our user base, who require the ability to find others' repositories, navigate through their code base, and view pull requests. For this, we need a much more stable design than simply a few lines of code.

The design of our application involves an Angular user interface hosted on top of Nginx, which will also serve as a proxy to the API. As for the API, it will run with Spring Boot to manage requests and establish a connection to our PostgreSQL relational database. However, not all our data will be structured or compact, like storing the repositories needed for previewing, so there will be a Minio Simple Storage Service for the API to offload this data to. This design involves a significant amount of structure and networking to manage, so we will utilize Docker Compose to host all these services within their own containers.

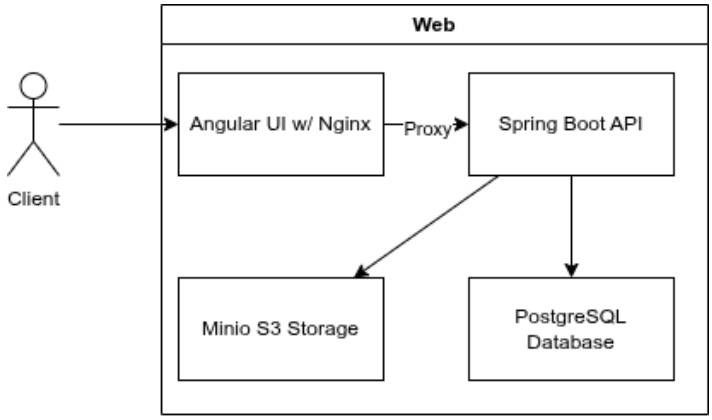


***Figure 5.*** *Web application system architecture diagram.*

### 4.3.3. Functionality

The functionality of GitTor can be described in stages. For this description, we will assume two users exist, Alice and John.

- Alice initializes a new repository on her system.
- Alice makes a few Git commits, adding code and authorizing other contributors, such as John.
- Alice tells her GitTor CLI to seed the repository, which in turn causes a sequence of events:
    - Alice's GitTor CLI tells Alice's seeder service to begin seeding the repository.

        - Alice's GitTor CLI notifies GitTor Web of the new repository and how to torrent it.
        - GitTor Web begins to leech the repository, allowing it to display a preview of the code.
- John finds Alice's new repository on GitTor Web and decides he wants to contribute to it.
- John tells his GitTor CLI to leech Alice's repository.
- John adds Git commits with even more code.
- John tells his GitTor CLI to seed the new state of Alice's repository, which again causes the same sequence of events:
    - John's GitTor CLI tells John's seeder service to begin seeding the repository.
    - John's GitTor CLI notifies GitTor Web of the new state of the repository and how to torrent it.
    - GitTor Web begins to leech the new state of the repository, allowing it to display a preview of the code.
- Alice sees this change, and decides to leech it onto her machine.

### 4.3.4. Areas of Challenge

Over the development of GitTor, we have encountered many challenges that we have had to overcome.

One area of difficulty we identified in our initial design was that the CLI would need a separate service to manage repository seeding to maintain reliability. We, however, did not correctly identify how challenging it would be to create this other program with proper inter-process communication. The main source of difficulty was that we needed the seeder service to handle multiple inter-process connections at once, because — while unlikely — a system call could issue multiple CLI commands at once, and the seeder service must still be able to handle this. Which means the seeder service must be written as two processes, one for seeding and one for handling connections, with multithreading for each connection. To add to this complexity, we wanted to maintain Windows support, which meant all of this code must be written as operating system agnostic, greatly reducing the number of tools and system calls available.

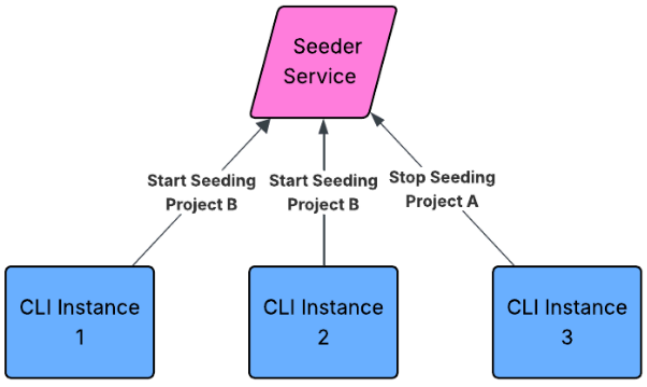


***Figure 6.*** *Seeder service architecture diagram.*

Another area of our project that took a lot of our team's time was creating an overall feel for our web application that we were happy with. We both wanted it to be similar enough to GitHub's design that anyone familiar with GitHub could move over without any issues, while also being

distinct in our own way. Our web application underwent many design iterations, and whenever a new page was added, it had to be evaluated individually to ensure it fit the overall design. Our team did not expect how much time and effort would go into simply getting a well-designed website, ignoring all functionality.

## 4.4. Technology Considerations

**CLI Technologies**

*C Programming Language*

- Strengths: Low-level control, minimal runtime overhead, excellent compatibility with existing Git tooling and system APIs, widely available on Linux systems
- Weaknesses: Manual memory management increases development complexity and bug potential, fewer built-in abstractions compared to higher-level languages, steeper learning curve for team members less experienced with systems programming
- Trade-offs: We chose C over Python/Rust to minimize resource consumption and to maximize performance for large repository operations. The development complexity is justified by the performance requirements.

*GPG/OpenPGP Libraries*

- Strengths: Industry-standard cryptographic verification, Git already uses GPG signatures, so integration is simpler
- Weaknesses: API complexity, key management requires careful error handling

**Web Application Technologies**

*Angular (Frontend)*

- Strengths: TypeScript provides type safety, component-based architecture, strong ecosystem with Angular CLI and testing tools
- Weaknesses: Steep learning curve, larger bundle sizes compared to lighter frameworks
- Trade-offs: Choose Angular over React/Vue for its opinionated structure and built-in features. The bundle size concern is acceptable given our target user base has reliable internet.

*Spring Boot (Backend API)*

- Strengths: Mature Java ecosystem, excellent dependency injection, built-in security features, strong database integration
- Weaknesses: JVM memory overhead, potentially slower startup times
- Alternative Considered: Express.js would have lighter resource usage, but Java's type system and tooling better support our API reliability requirements.

# 5. Testing

For our application, testing is one of our highest values, as it ensures the system is functioning correctly with minimal to zero manual effort. However, since our application is split into three parts —CLI, API, and UI —we need a different test suite for each, tailored specifically to that part. While each of these tests function differently, their overall goal will be the same: to ensure that the requirements for the developed feature are met. The sections below outline the various forms of testing we used and explain how they are best suited for each part of our application.

For each task on our Kanban board, we marked the form of testing that needed to be implemented and guaranteed that those tests were created and passed before merging. This helped us keep track of our project and ensured old features weren't unintentionally broken in the implementation of a new feature.

## 5.1. Unit Testing

In general, our project avoided unit testing where possible, as the more accurately the tests simulate the entire environment, the better they are. This isn't to say we didn't have unit testing, as it's often the best option available; however, when integration testing was a viable alternative, it was used instead.

Our CLI is an excellent case where unit testing was the best option available. Developing high-level tests, such as integration tests, is nearly impossible for low-level languages like C, on which the CLI was built. For this reason, the CLI's tests were written in a framework called Unity, which got its name because it was designed around unit testing. While our tests here check the function's output, they were primarily focused on preventing memory leaks and infinite loops more than anything else. To handle this, our tests were run on top of Valgrind, which analyzed the memory usage and detected any errors.


***Figure 7.*** *CLI test coverage report.*

## 5.2. Interface Testing

The best example of interface testing in GitTor comes from our Simple Storage Service. The API requires storing large files and/or unstructured data, and that's where the Simple Storage Service (S3) comes in. It's a system that allows the program to upload, download, or delete these objects. The only complication is that the API actually uses three different S3 implementations, depending on the runtime: in-memory, in files, or in MinIO.

Since we didn't want the program to behave differently across runtimes, we needed a test suite to ensure that none of the implementations differed at the interface level. This test suite didn't even guarantee that they functioned "properly"; it just checked that they all behaved the same when given the same input. In creating this, we actually found some cases where specific characters were allowed in one implementation but not in another, which we later fixed.

## 5.3. Integration Testing

The most important area for us to do proper integration testing was the API. It was necessary to ensure that all controllers functioned as expected, which would have been extremely tedious to do manually, especially as the system grew and changed. For this reason, almost all of our API tests were examples of integration tests.

**api**

| Element | Missed Instructions | Cov. | Missed Branches | Cov. | Missed | Cxty | Missed | Lines | Missed | Methods | Missed | Classes |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| api.services | | 93% | | 80% | 24 | 144 | 16 | 264 | 9 | 106 | 0 | 9 |
| api.mapper | | 77% | | 47% | 22 | 44 | 28 | 99 | 2 | 20 | 0 | 5 |
| api.services.storage | | 89% | | 92% | 1 | 25 | 12 | 78 | 0 | 18 | 0 | 3 |
| api.exceptions | | 85% | | n/a | 6 | 30 | 7 | 40 | 6 | 30 | 0 | 5 |
| api.controllers.users | | 98% | | 85% | 2 | 32 | 0 | 83 | 0 | 25 | 0 | 3 |
| api.controllers | | 98% | | 94% | 1 | 34 | 1 | 78 | 0 | 25 | 0 | 5 |
| api | | 37% | | n/a | 1 | 2 | 2 | 3 | 1 | 2 | 0 | 1 |
| api.utils | | 92% | | n/a | 1 | 3 | 1 | 15 | 1 | 3 | 0 | 1 |
| api.configs | | 98% | | n/a | 1 | 12 | 1 | 16 | 1 | 12 | 0 | 2 |
| api.components | | 100% | | 100% | 0 | 15 | 0 | 33 | 0 | 11 | 0 | 2 |
| api.entities | | 100% | | n/a | 0 | 2 | 0 | 1 | 0 | 2 | 0 | 1 |
| Total | 257 of 3,492 | 92% | 44 of 178 | 75% | 59 | 343 | 68 | 710 | 20 | 254 | 0 | 37 |

***Figure 8.*** *API test coverage report.*

Developed using JUnit, the tests sent REST requests to the controller layer, which called the service layer, repository layer, and finally the database/S3. For these tests to run correctly, all pieces of the API needed to be integrated and running together. However, most of the time, we didn't want to set up an entire PostgreSQL database or MinIO instance to test the application. In these cases, our system simply replaced them with an H2 database and an in-memory S3.

## 5.4. System Testing

When possible, system testing in the form of End-to-End testing is the golden standard, as it ensures your entire system can collaborate effectively to achieve the desired result. In the context of our application, this was most achievable with the web user interface and its interaction with the API.

To create these tests, we used Cypress, which ran the entire web user interface and interacted with it as an automated user, ensuring the proper information appeared on screen. This approach provided confidence that both the frontend and backend were functioning together correctly under real-world conditions.

## 5.5. Regression Testing

To ensure that new changes did not break existing features, the entire test suite needed to be run, which would have been very tedious. To manage this, we have set up GitHub workflows that run all our tests on each new commit to the repository, along with other checks. For a pull request to be merged into the main branch, all checks had to pass. Along with ensuring new changes did not break existing features, it also verified that the latest tests created with this feature also passed.

## 5.6. Acceptance Testing

For a new feature to be added to our system, it had to go through a pull request, which first verified that all automated checks passed. After that, however, it also had to be reviewed by another team member, who would either approve it or request changes. Only when all checks passed, and another team member had approved it could it be merged into the main branch.

Every two weeks, our team met with the client to showcase all the changes made to the system. At this time, if any new feature had modifications requested, a new issue would be created on the Kanban board, and the cycle continued.

## 5.7. Security Testing

In every system, security testing takes a different form. For our system, and given our resources, the best form of security testing we could realistically have was primarily static code analysis. In each of our environments: CLI, API, and UI, we set up static code analysis, which searched our codebase for common security vulnerabilities as well as other issues. As mentioned before, we used Valgrind to perform memory analysis during our CLI testing, which can also help prevent many security issues.

In the end, however, no form of security testing is complete, and there will always be potential for security threats, even if it's just built into the libraries the project depends on. Our system could have benefited from more security testing, but at some point, we decided it was good enough and focused our efforts elsewhere.

## 5.8. User Testing

Admittedly, user testing is the most difficult form of testing for our team. Since our own ambition initially inspired the project and our team invested its time in development rather than marketing, we do not have a community lining up to try it.

That being said, we have always tried to put ourselves in users' shoes during development, which is especially helped by the fact that each member of our team is an ideal user of GitTor, as we are software developers. Along with this, we have our meeting with our advisor, where he can offer a fresh perspective on our application and provide recommendations as needed.

### 5.9. Results

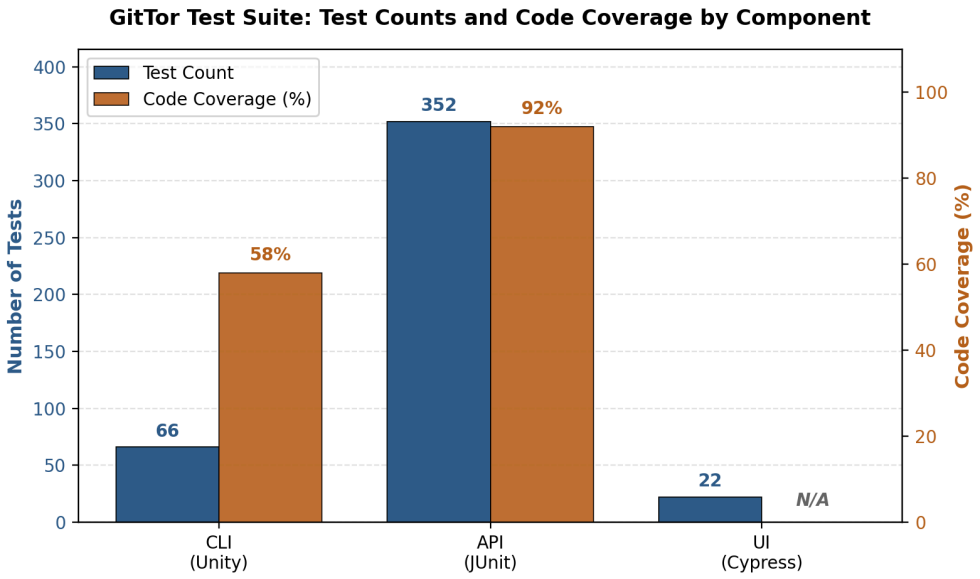


***Figure 9.*** *Test suite analysis per component*

In the end, we created 22 Cypress tests for the UI, 66 Unity tests for the CLI, and 352 JUnit tests for the API. We have coverage reports for the CLI and the API, indicating 58% and 92%, respectively. The reason for the CLI's supposedly low coverage percentage is that restrictions are preventing torrenting testing on GitHub runners. However, we manually tested all those areas and confirmed they work without issue.

Due to our extensive testing, we have identified only one major bug that made it to production, caused by an inconsistency between mobile Safari's rendering and that of all other browsers. The remaining issues on our Kanban marked as bugs were due to problems in our development environment rather than the product itself.

## 6. Implementation

### 6.1. Project Setup

The start of our project began with creating two separate repositories: one for the command-line tool and one for the web application. The web repository was split into folders for frontend and backend. From there, we initialized the environments, referencing previous projects we'd worked on: C for the CLI, Spring Boot for the API, and Angular for the UI.

Once we had these bases to work from, we established the must-haves for our development process, which included linting, style checks, autoformatting, testing, coverage reports, and CI/CD. To view the tools used for each environment, refer to the appendices. Although this was seemingly complete within the first month, in reality, this has been an ongoing process of improvement as we have found more areas of our development environment in need of improvement.

## 6.2. Command Line Interface

```
isaac@isaac-asus:~$ gittor --help
Usage: gittor [OPTION...] COMMAND [ARGUMENTS...]
COMMANDS:

  init     Create an empty GitTor repository
  leech    Clone a GitTor repository into a new directory
  seed     Share the current state of the repository
  devs     Manage who can contribute to this repository
  verify   Verify all commits are from authorized developers
  config   Get and set GitTor local or global configurations
  service  Manage the GitTor service

OPTIONS:

  -p, --path=PATH            The path to the gittor repository
  -?, --help                 Give this help list
      --usage                Give a short usage message

Mandatory or optional arguments to long options are also mandatory or optional
for any corresponding short options.
```

***Figure 10.*** *CLI main help message.*

With the command-line interface, we identified many libraries that we would need at some point in the project and decided to add them as soon as possible. This process was trivial in Linux, but it introduced numerous issues when attempting to compile for Windows. While Windows support was not a requirement for our project, it was still desirable, as it would open us up to a larger user base. With considerable effort, we eventually successfully made all libraries functional on both Linux and Windows.

Once we had these libraries to work with, we needed a command-line parser to handle the numerous sub-commands of GitTor, as well as any parameters and flags. To ensure that our team could easily collaborate on the command-line interface without stepping on others' toes, we templated all the sub-commands and their parsers.

While this was happening, we wanted to demonstrate the feasibility of our application to our client, so we used a combination of prototyped C code and external tools to share a repository with torrenting between computers on separate networks. This prototype still required many manual steps, but it showed that what we were attempting to do was entirely possible.

```
·$ gittor tor
seeding 4211 kB/s 276445 kB (100%) downloaded (111 peers)
saving session state

done, shutting down
```

***Figure 11.*** *Prototyped torrenting feature.*

Next, we wanted to complete the configuration sub-command as we knew it would be used later by other sub-commands to read users' custom configurations. This task introduced extra complexity to our system, as we are managing both local repository configurations and global user configurations.

The end of the first semester marked the implementation of the seeder service, a separate service process for continuous seeding. This process needed to support inter-process communication from the CLI with multiple connections at a time. The only way to solve this was with a combination of multiprocessing and multithreading. However, the main complexity of this feature was that it should be functional on both Linux and Windows, if possible. Because of this, the only proper way to manage inter-process communication was through ports and sockets, which introduced a whole new layer of complexity.

Towards the beginning of the second semester, the CLI development team established the necessary connections to the already built API endpoints for authentication, uploading torrents, downloading torrents, etc. With this task, a new library needed to be added to the CLI to address a need that had not been identified when all the other libraries were added: JSON parsing. Adding this library required, once again, a lot of work to ensure Windows support.

Once the endpoint connections were complete, the leeching and seeding sub-commands could be finalized. As these were the main commands of our application, they had been slowly developed throughout the year. However, the last puzzle piece was in place, so that now, by simply calling a GitTor command, a repository could be shared entirely over P2P connections.

### 6.3. Web Application

Unfortunately, unlike the CLI, the setup for the web application repository was not limited to the items discussed in the project setup section. We knew we wanted our web application to run on top of Docker, so we developed multi-stage container files to build the API and UI runtime images, which would run with the Temurin JRE and Nginx, respectively. Once we had the image build instructions, we created Compose files to manage all the necessary containers, including the database, Simple Storage Service, and the API and UI containers [7], [10], [11], [12].

Another piece of setup we wanted was the ability to generate API connection code in the UI automatically. We used the OpenAPI specification generated by Spring Boot, along with the OpenAPI command-line interface, to automatically generate TypeScript code for our frontend with data transfer objects and API endpoints.

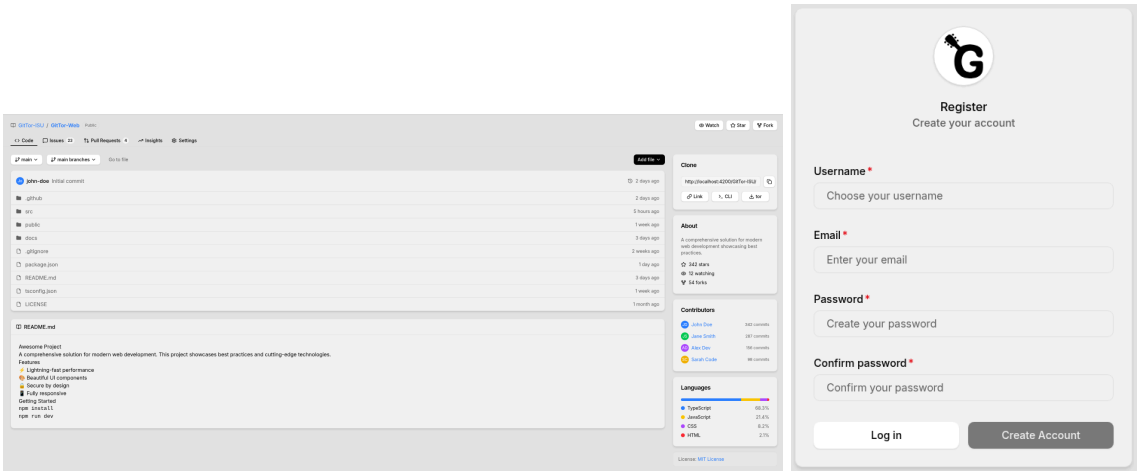


***Figure 13.*** *Web application repository and login pages.*

Once our web application was finally set up and ready for features to be added, we created the most fundamental features for any API: users, roles, authorities, and authentication. While this may sound simple, it required roughly 24 endpoints and 125 tests [6]. We then began work on our project-specific endpoints, which involved retrieving, uploading, and updating repository torrent files.

While all this progress was being made on our backend, our frontend team was busy designing a rough draft for the aesthetic and layout of our application. These designs underwent multiple iterations before approval, taking longer than initially estimated.

After the designs were finalized, the frontend underwent restructuring, retheming, and setup of layouts and fundamental components to prepare it for the implementation of these designs. This also helped prevent accidental duplication of work by implementing these necessities ahead of all the future tasks.

Once everything was set up, the UI developers were quick to develop the many necessary pages for controlling the web application. The implementation of these followed a three-stage process. First, the design was implemented with entirely dummy data. Once that was approved, the functionality would be added by connecting the webpage to the API. Finally, testing was added to prove it worked properly and ensure future changes would not break this feature [6].

## 6.4. Unimplemented Features

While we were able to accomplish many things this semester, we would be lying if we didn't say our original goals were slightly unrealistic and that we came up short in a few areas. While we did complete the main functionality of our project, to share Git repositories via a P2P network, some of the more minor areas had to be reduced.

The largest of these unimplemented features is the repository validation subcommand in the CLI. In development, we successfully used a file containing public GPG keys to validate the authenticity of a repository commit; however, the library we used lacked Windows support. We spent many hours trying to build the library from source on Windows and exploring alternatives, but we

realized this was not the only issue we had to deal with. Due to the nature of our authenticity checks, the validation command would have to rollback the repository through each commit, checking its validity before either reaching an already validated commit or the beginning of the repository. We made plans to avoid this issue, such as storing the authorized public GPG keys in the commits, perhaps in the description, so we wouldn't have to rollback the commits, but at this point, we were reaching the end of the second semester. Given how much effort would be needed to implement this feature, not to mention getting Windows support if we chose to keep it, we decided to focus our efforts elsewhere. That being said, we have a very clear idea of a few different ways this could be implemented and have tested every part of the system; we just did not have enough time to implement it properly.

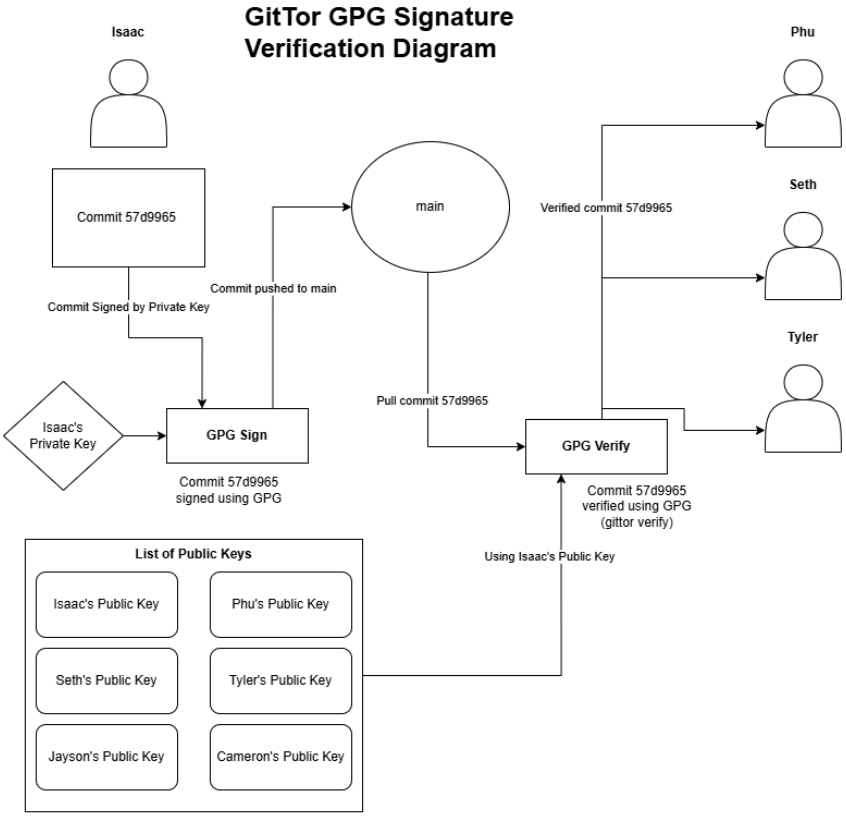


***Figure 14.*** *GPG validation interaction diagram.*

Another area where our implementation did not meet the expectations set by our design is our API. A big goal for our project was for users to be able to access repository contents through the UI, which would require the API to fetch the repository whenever changes are made. We knew of this going into the project, but did not fully appreciate the level of work required to implement torrenting features until we began implementing them in the CLI. On top of realizing how much work we had ahead of us with this task, it also could not be effectively completed until the CLI successfully seeded repositories so that the API could leech them. And by the time we had finalized the CLI seeding, we recognized that there simply wasn't enough time to complete the API torrent functionality.

Overall, though, we reached our goal of sharing Git repositories via a P2P network using a command-line tool. The only areas we came up short were in quality-of-life features, such as viewing repository content online and full GPGP validation integration. And even these are areas we know exactly how to complete if we had more time.

## 6.5. Design Analysis

The GitTor design successfully delivers on its promise of a decentralized architecture that removes the central hosting platform as a single point of failure. However, the implementation highlights a clear trade-off between total decentralization and synchronization speed.

### 6.5.1. Strengths

The system excels at providing a resilient, "un-killable" repository state. By utilizing torrent files for sharing, we have effectively stripped the web application of its power to manipulate repository history.

- **Decentralized Discovery:** The web application provides a clean, documented API and UI for finding repositories without actually owning the data.
- **Integrity Assurance:** The torrent protocol's native hashing ensures that every byte of a repository is verified upon download, preventing the "silent commit injection" issues common with centralized platforms.
- **Evidence:** We have been able to use GitTor to share the GitTor source code repositories between contributors via an entirely decentralized approach.

### 6.5.2. Weaknesses

The primary drawback of the current implementation is that GitTor is marginally slower than traditional Git workflows. Unlike the near-instantaneous handshakes of HTTP or SSH, the torrenting process takes a few seconds to perform peer discovery and establish connections to seeds. While the transfer itself is fast, the initial startup lag is noticeable to the user.

A potential solution to the latency issue is a Hybrid Model, where torrenting serves as a robust backup to traditional HTTP or SSH methods. In this scenario, users would enjoy the high speed of a central server for daily tasks, with the P2P protocol serving as an automatic fallback if the central server goes offline or attempts to block a user.

Our team actually considered this hybrid approach from the very beginning of the project. However, after a group vote, we decided to commit exclusively to the torrenting protocol. We chose this path to clearly demonstrate the viability of a fully decentralized Git environment and push the limits of P2P functionality, even though we knew it would result in worse latency than the traditional approach.

By focusing solely on torrenting, we ensured that every part of our development was dedicated to solving the challenges of decentralization rather than relying on existing centralized architectures.

# 7. Ethics and Professional Responsibility

## 7.1. Areas of Professional Responsibility/Codes of Ethics

| **Areas of Responsibility** | **Definition** | **Relevant Code of Ethics** | **Project Application** |
|---|---|---|---|
| Work Competence | Create a quality product. | 3. PRODUCT: Ensure products meet the highest professional standards. | We have selected mature tools and applied rigorous testing to ensure the quality of our product holds up to our original vision.<br>As newer versions of these tools were released, we migrated the project to keep it up to date. |
| Financial Responsibility | Create it at a reasonable price for its value. | 2. CLIENT & EMPLOYER: Act in the client's best interest. | Our project is entirely open source ensuring that our users never have to pay a dime for it. |
| Communication Honesty | Be honest to your users. | 4. JUDGMENT: Integrity in professional judgment.<br>6. PROFESSION: Advance the integrity and reputation of the field. | All our bugs and issues are publicly posted to our projects GitHub ensuring users are aware of all issues that may affect them. |
| Health, Safety, Well-Being | Minimize safety risks of the product. | 1. PUBLIC: Act consistently with public interest. | We are helping the well-being of our users by giving them the peace of mind that their repositories cannot be tampered with by a third-party. |
| Property Ownership | Respect the ownership of property of others. | 2. CLIENT & EMPLOYER: Act in the client's best interest.<br>7. COLLEAGUES: Support colleagues. | Our design prioritizes personal control of data so that our users do not have to upload any of their personal information to a third party.<br>We continued this by |

| | | | |
|---|---|---|---|
| | | | limiting the amount of information users must provide when registering on the web. |
| Sustainability | Minimize environmental risks of the product. | 1. PUBLIC: Act consistently with public interest.<br>6. PROFESSION: Advance the integrity and reputation of the field. | Our decentralized approach removes reliance on large data centers, reducing the environmental impact of repository collaboration. |
| Social Responsibility | Ensure the product benefits society. | 1. PUBLIC: Act consistently with public interest.<br>6. PROFESSION: Advance the integrity and reputation of the field. | GitTor addresses many societal concerns about centralized control, censorship, and manipulation of open-source projects. |

***Table 5.*** *Professional Responsibility and code of ethics.*

Of these responsibilities, our team performed very well in terms of Communication Honesty. Almost all of our project's communication has been publicly viewable, including our development process. Because of this, our users can see everything we've done, including our mistakes, and make a very informed decision on whether to use our project.

An area where our team needs improvement is Sustainability. While GitTor's decentralized design inherently removes reliance on large data centers, we have not yet fully addressed the long-term environmental footprint of our application. The torrenting protocol introduces a large amount of overhead that will need to be evaluated.

## 7.2. Four Principles

| **Broader Context Area** | **Beneficence** | **Nonmaleficence** | **Respect for Autonomy** | **Justice** |
|---|---|---|---|---|
| **Public Health, Safety, Welfare** | Protects users from repository tampering, supply chain attacks, and untrusted code, improving digital safety and peace of mind. | Local security or usability mistakes could still expose users to risk; misconfigured GPG keys may allow accidental invalid commits. | Users control which commits to trust and propagate, maintaining agency over their repositories. | No single platform can block or manipulate repositories, promoting fair access and protection for all users. |
| **Global,** | Supports global | Network reliance | Contributors can | Benefits and |

| | | | | |
|---|---|---|---|---|
| **Cultural, Social** | collaboration across cultures and communities by enabling decentralized participation without central censorship. | and complexity could disadvantage users in regions with limited bandwidth or technical expertise. | choose whom to collaborate with and which repositories to access, supporting self-directed participation. | access to repository data are distributed equitably among all contributors, regardless of location or affiliation. |
| **Environmental** | Reduces dependence on large, energy-intensive data centers through torrenting. | Users' local devices consume additional power for seeding, producing some environmental cost. | Users can decide how much to seed and when, controlling their personal environmental footprint. | Workload is distributed across peers, preventing unfair environmental burden on any single server or organization. |
| **Economic** | The project is open-source, preventing reliance on paid services, lowering costs for individuals and organizations. | Extra bandwidth or storage usage may increase personal costs for contributors who seed frequently. | Users choose whether to seed or leech repositories, preserving control over their resources. | Provides equal access to software tools and collaboration opportunities without favoring wealthier participants. |

***Table 6.*** *Four principles.*

For our project, the most critical pair for us is Public Health, Safety, Welfare, and Beneficence. By ensuring that repository content cannot be tampered with silently and that commit authenticity is cryptographically verified, we improve users' digital safety and trust. We provide this benefit by requiring all commits to be signed with GPG keys, validating contributors against the repository's authorized keys file, and distributing repositories via P2P torrenting with hash verification to prevent unauthorized modifications.

However, the pair that we are most lacking in is Environmental and Nonmaleficence. While the system reduces reliance on large centralized servers, contributors still consume personal power to seed repositories. This negative impact is partially mitigated by allowing users to control when and how much they seed, as well as by distributing the workload across multiple peers. However, it is difficult to measure the true scale of this impact without a large user base and a network of repositories.

## 7.3. Virtues

**Honesty -** Consistently communicating truthfully and transparently about intentions, progress, and limitations.

**Accountability -** Taking ownership of self assigned tasks, setting personal deadlines, and reliably following through on commitments.

**Diligence -** Performing work carefully and thoroughly, ensuring quality and correctness beyond the minimum requirements.

## 7.4. Reflections

**Isaac Denning** - I feel the virtue I have best demonstrated is accountability. As the team lead, if I am not able to be accountable for my own tasks, I will not be able to lead our project to success. So, Throughout the year, I have consistently self-assigned tasks and completed them in a timely manner to ensure the continued success of our project.

The virtue that I still need the most improvement in is my diligence. I want to ensure that my contributions are as good as they can be, but I often make mistakes and forget changes I intended to make. Thankfully, other team members frequently catch these mistakes during the review process.

**Seth Clover -** Honesty was the virtue that I demonstrated best this year. I made a conscious effort to communicate to my team what I was working on and to be transparent about when I was stuck or falling behind on my deadlines. Being open with my individual progress allowed me to get help when I needed it and allowed me to stay grounded in my personal intentions and goals.

With a strong display of honesty, my next step is to improve on my accountability. I want to be able to set more concrete goals for myself and complete them on time more consistently. Improving on this virtue will allow me to maintain momentum in the project's development process and make me more reliable to my teammates.

**Tyler Gorton** - In my work on our project, I think I have most successfully exhibited the virtue of diligence. I've always prided myself on paying attention to the details and holding my work to a high standard, because I believe that is critical to providing the best experience to users. This year I dedicated myself to writing high quality code and testing it thoroughly in order to make my contributions as beneficial as they could be.

However, this ties into one of my most significant weaknesses in my senior design work, which is accountability. Several times this year I set a target or deadline for myself which I failed to meet due to some obstacle, and in many of these cases the setback was related to a minor issue that may have been better handled after completing more significant components of the task. Often it's more important to integrate the core of a feature so it can be evaluated with the project as a whole before finalizing the details, which I will make an effort to do in the future.

**Jayson Acosta -** The virtue I have demonstrated most is diligence, as I have consistently surpassed the requirements on every webpage I have worked on. While I could stop once the page meets the requirements, I continue to work on improving its style and color until I am happy with it.

This leads into the virtue I need to work on most, which is accountability. By going above and beyond on each page, I often go past the original estimated timeline. In the future, I will strive to be more transparent with the team about any delays I anticipate due to this issue.

**Cameron Gilbertson** - The virtue that was most apparent in my work was honesty. During the year I communicated with my teammates on my progress, good or bad. This shows that even when I could have not told the truth I owned up to my limited progress and provided reasons and my plan to get back on track. This virtue is important to me because when people do not communicate their progress honestly it can create a false sense of progress for the entire group that could have been corrected if the truth had been known

The virtue that I believe I could improve on would be accountability. This is important because no one wants a micromanager who is constantly nagging about completing tasks. But the freedom from a micromanager requires a certain amount of accountability from each team member to complete their tasks. The way I could improve my accountability would be to treat my deadlines with more urgency and take initiative to complete my tasks on time.

**Phu Nguyen** - The virtue that has best represented my senior design work so far is diligence. This virtue is vital because quality work helps develop a more robust codebase that can withstand the test of time, even if another team takes over. I have demonstrated this virtue by putting effort into ensuring the work I am doing is clear, utilizing up-to-date practices, and having it reviewed by multiple eyes.

One virtue that I want to improve on is accountability. This virtue is important because it demonstrates responsibility and benefits not only yourself but also everyone on my team. To achieve this virtue, I aim to stay ahead on assignments and be more involved in setting goals for specific features in the future.

# 8. Conclusion

## 8.1. Summary of Progress

Our team successfully developed and implemented the core functionality of GitTor, a decentralized Git environment that eliminates the single point of failure inherent in traditional hosting platforms. By creating a peer-to-peer (P2P) command-line interface and a lightweight web application, we achieved our primary goal: enabling users to share and synchronize Git repositories entirely through a torrent-based network.

The finalized design consists of two distinct components: The command-line interface and the web application.

Written in C, the CLI handles the heavy lifting of repository synchronization. By leveraging libraries such as libtorrent and libcurl, the CLI allows users to publish the state of their repositories, generate torrent files, and seed the contents to the P2P network.

The Web App was built with a Spring Boot Java API and an Angular 21 frontend. This application acts as a bulletin board. It allows users to publish their torrent files and discover other repositories without the platform ever having control over the actual repository contents. We utilized an autogenerated OpenAPI specification to ensure seamless communication between the UI and API, backed by a PostgreSQL database and MinIO for robust object storage. The entire stack was successfully containerized using Docker Compose.

To ensure the reliability and security of this decentralized architecture, we implemented many different styles of testing within a continuous integration pipeline:

In the CLI, we used the Unity C testing library alongside Gcovr for comprehensive coverage reports. Crucially for a network-facing C application, we utilized Valgrind for memory analysis to ensure leak-free execution, alongside CppLint and Clang-Tidy for strict rule checking.

As for the web application, we tested it using JUnit integration tests and unit tests, with JaCoCo for coverage reports. The Angular UI was validated with Cypress end-to-end (E2E) testing to ensure smooth user workflows, and formatted with Prettier and ESLint.

The most significant milestone of our progress was proving the viability of our own concept. By the end of the development cycle, we successfully used GitTor to share the GitTor source code repositories among our contributors via an entirely decentralized, P2P approach.

While certain quality-of-life features, like in-browser repository viewing and automated GPG validation rollbacks, were scoped down due to time constraints, the core objective was unequivocally met. We can collaborate on Git repositories without a central point of failure.

## 8.2. Value Provided

The GitTor architecture provides immense value by fundamentally shifting the trust model of version control from a centralized corporate entity to a verifiable, cryptographically secure peer-to-peer network. In the broader context of modern software development, where projects rely heavily on platforms like GitHub or GitLab, GitTor addresses the critical vulnerabilities that arise when a single platform acts as a single point of failure and the ultimate authority over a repository's state.

Our design successfully tackled the four primary problems we identified with centralized platforms:

Eliminating the Single Point of Failure (Availability): If a central hosting platform experiences an outage, collaboration typically halts until the servers are back online or a complex migration is performed. GitTor solves this by making the repository "un-killable." As long as at least one contributor's machine is seeding the repository, the project remains available.

Cryptographic Integrity: Centralized platforms can silently manipulate commit history or exploit SHA-1 collisions to inject malicious code. By distributing repositories via torrent files, GitTor leverages BitTorrent's native piece-hashing. Every byte of the repository is verified against the hash stored in the torrent file upon download. This ensures that users receive the exact, unaltered repository regardless of which peer they leech from, completely preventing any hosting platform from secretly modifying the code.

Censorship Resistance: Traditional platforms can unilaterally block users, delete repositories, or lock access. By stripping the GitTor Web App of any actual control over repository contents—relegating it to a simple, documented discovery tool for torrent files—we return ownership to the developers. Contributors host the repository collectively, deciding for themselves which changes they wish to fetch and build upon.

The most compelling evidence of GitTor's value is our successful deployment of the tool to manage and share the GitTor source code itself. By relying entirely on our CLI and P2P network to synchronize our work, we proved that a completely decentralized workflow is not just theoretical, but functionally viable.

Furthermore, our design deliberately accepts a specific trade-off to provide this value. While GitTor experiences marginally slower synchronization speeds compared to the near-instantaneous handshakes of HTTP or SSH (due to the necessary peer discovery and BitTorrent handshake overhead), this latency is a calculated exchange for total decentralization. In a broader context where data sovereignty and protection against corporate censorship are becoming increasingly critical to the open-source community, GitTor provides a highly resilient alternative that prioritizes absolute security and user autonomy over marginal speed gains.

## 8.3. Next Steps

While our work this semester successfully established the foundation of a fully decentralized Git environment, several key areas provide a clear roadmap for future development. Expanding on these features would transition GitTor from a functional proof-of-concept into a highly competitive, production-ready version control system. If another team were to pick up this project, we recommend focusing on the following three initiatives: GPG validation, API leeching, and A hybrid model.

A core element of our security design was restricting write access to verified contributors via a public GPG key file. However, fully integrating this validation into the CLI was halted due to library compatibility issues on Windows and the computational overhead of having to "rollback" through the commit history to verify authenticity step by step.

Future development should implement our theorized workaround: storing the authorized public GPG keys directly within the commit metadata, such as the commit descriptions or Git trailers. This approach would allow the CLI to validate the chain of trust incrementally without performing an expensive repository rollback. As for the cross-platform issue, it would take time and effort to get Windows support for the cryptographic library, but it could be done.

Currently, our web application serves as a decentralized discovery tool for torrent files, but lacks the quality-of-life feature to display raw repository code in the browser. We realized during development that we would not have enough time after implementing seeding in the CLI to support leeching on the API and code display in the UI.

In the future, the API needs to be integrated with a torrenting client, allowing the backend to leech repositories as they are updated and store them in S3. Implementing this is vital for bridging the gap between GitTor and the user-friendly interfaces developers expect from modern tools, allowing for online code reviews and branch comparisons without granting the platform control over the data.

Lastly, to prove the viability of a purely decentralized system, we committed exclusively to the torrenting protocol. As anticipated, this introduced latency during the BitTorrent peer discovery and handshaking phases, making GitTor slightly slower than traditional Git workflows. A follow-up

project should explore a "Hybrid Model." In this design, GitTor would utilize high-speed HTTP or SSH connections to a central server for routine, day-to-day tasks. However, the system would retain the P2P protocol as an automatic, seamless fallback. This would provide the best of both worlds: the immediate responsiveness developers are accustomed to, backed by the absolute resilience and censorship resistance of a decentralized network.

# 9. References


[1] GitTor-ISU, "GitHub - GitTor-ISU/GitTor-Cli: GitTor-Cli Implementation," *GitHub*, 2025. https://github.com/GitTor-ISU/GitTor-Cli

[2] GitTor-ISU, "GitHub - GitTor-ISU/GitTor-Web: GitTor-Web implementation," *GitHub*, 2025. https://github.com/GitTor-ISU/GitTor-Web

[3] "GitTor • GitTor-ISU," *GitHub*, 2025. https://github.com/orgs/GitTor-ISU/projects/1

[4] D. Huigens, J. Winter, and Y. Niibe, "OpenPGP," Jul. 2024, doi: https://doi.org/10.17487/rfc9580.

[5] J. Callas, L. Donnerhacke, H. Finney, D. Shaw, and R. Thayer, "OpenPGP Message Format," *www.rfc-editor.org*, Nov. 2007, doi: https://doi.org/10.17487/RFC4880.

[6] 14:00-17:00, "ISO/IEC/IEEE 29119-1:2022," *ISO*. https://www.iso.org/standard/81291.html

[7] 14:00-17:00, "ISO/IEC 9075-1:2023," *ISO*. https://www.iso.org/standard/76583.html

[8] E. Rescorla, "The Transport Layer Security (TLS) Protocol Version 1.3," Aug. 2018, doi: https://doi.org/10.17487/rfc8446.

[9] "HTTP/2," Jun. 2022, doi: https://doi.org/10.17487/rfc9113.

[10] The, "The OpenContainers Distribution Spec," *https://opencontainers.github.io*, 2021. https://specs.opencontainers.org/distribution-spec/

[11] The, "The OpenContainers Image Spec," *https://opencontainers.github.io*, 2021. https://specs.opencontainers.org/image-spec/

[12] The, "The OpenContainers Runtime Spec," *https://opencontainers.github.io*, 2021. https://specs.opencontainers.org/runtime-spec/

[13] C. Staff, "What Is a Peer-to-Peer Network?," *Coursera*, Oct. 24, 2024. https://www.coursera.org/articles/peer-to-peer

[14] "Radicle: the sovereign forge," *Radicle.xyz*, 2026. https://radicle.xyz (accessed Mar. 29, 2026).

[15] GitHub, "GitHub," *GitHub*, 2013. https://github.com


# 10. Appendices

## 10.1. Operation Manual

GitTor is a project with many parts that can be used in many different ways, depending on the user's needs. This user manual has been separated into the main sections of our project that users will interact with and covers the most common usages, but not everything that can be done with GitTor.

### 10.1.1. GitTor CLI Installation

#### 10.1.1.1. Linux Installation

Before installing GitTor on Linux, you must first install all the dependencies:

- build-essential
- libtorrent-rasterbar-dev
- libglib2.0-dev
- libgit2-dev
- libcurl4-openssl-dev
- libjson-glib-dev

While this installation process is distribution-specific, if you are on a Debian-based distribution, the installation can be done with these commands:

```
sudo apt update
sudo apt install -y build-essential \
    libtorrent-rasterbar-dev \
    libglib2.0-dev \
    libgit2-dev \
    libcurl4-openssl-dev \
    libjson-glib-dev
```

As of now, no prebuilt binaries are provided for any Linux distributions. So, once you have all the necessary dependencies for GitTor, you can build the program from source with this command:

```
git clone https://github.com/GitTor-ISU/GitTor-Cli.git
cd GitTor-Cli
make
make install
```

During installation, the program may or may not prompt you to run a command to add GitTor to your PATH. Once this is complete, you should be able to call GitTor directly without any issues.

```
gittor --help
```

If you are encountering any issues installing GitTor on your system, please refer to the devcontainer configuration to help troubleshoot any discrepancies.

#### 10.1.1.2. Windows Installation

To install GitTor on Windows, download the binary from the most recent build artifact here. Extract this archive to a location on your system where you won't delete it, and add its path to your PATH environment variable. There are many tutorials online that can show how to edit your PATH environment variables.

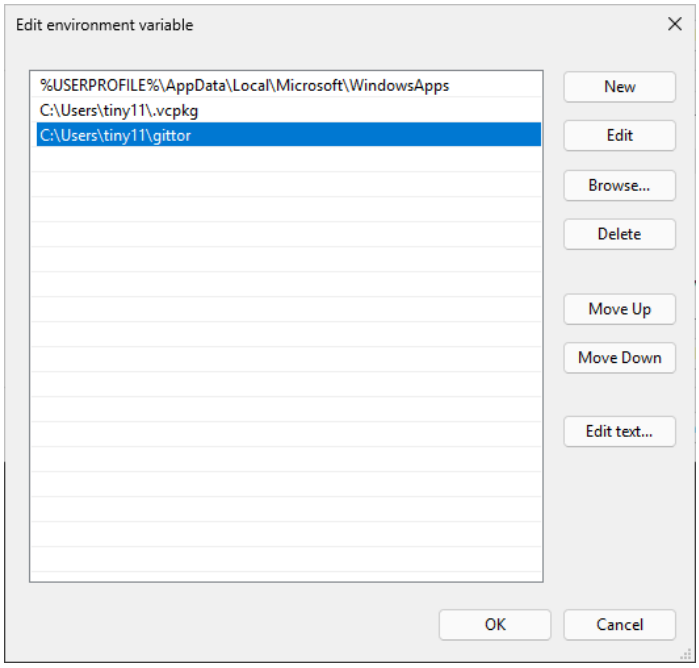


***Figure 15.*** *Windows environment variables editor.*

From there, you should be able to call GitTor directly without any issues.

```
gittor --help
```

### 10.1.2. GitTor CLI Configuration

GitTor supports many custom configurations that should ideally be set up before you begin using the application; however, these can be changed at any time. In order to configure GitTor, open up a file named '.gittorconfig' in your HOME directory and begin by adding this content:

```
[network]
port=12345
api_url=https://gittor.rent/api
tracker1=https://tracker.moeblog.cn:443/announce
tracker2=https://tr.nyacat.pw:443/announce
tracker3=https://tr.highstar.shop:443/announce
tracker4=https://tracker.gcrenwp.top:443/announce
```

Some will recognize this format as the same as the '.gitconfig' configuration file for Git. This design choice was intentional to make an easier transition for our users already familiar with Git.

The 'port' configuration value is the port that GitTor uses for seeding repositories. While not entirely necessary, seeding repositories is much more effective when you have a public-facing port that can be configured here. However, we acknowledge that it is often outside the reach of everyday users to port forward their machine, but when possible, it makes leeching much faster for other users.

The 'api_url' configuration value is used to specify which instance of the GitTor Web application you wish to connect to for finding and uploading repository torrents. Here we have shown the URL to our GitTor web's API; however, if you decide to use your own deployment or someone else's, that would be configured here.

The trackers are a list of URLs to public torrent trackers, which are used to orchestrate connections between seeders and leechers for any given torrent. This list of tracker URLs can be anywhere from one to one hundred trackers, which will be used whenever you are creating a new torrent (i.e., pushing new content to a repository). The trackers shown here are just a few that our team used with some reliability throughout our development process, though it is entirely up to the user which trackers they want to use for their own torrents.

### 10.1.3. GitTor CLI Usage

GitTor has many sub-commands, many flags, and many parameters. So, while all possible uses of the CLI cannot be expressed here, the typical usage looks like this:

1. First, the repository is initialized, and in this case, the optional folder parameter is used with 'project.'
2. The user enters the repository and makes some basic changes.
3. These changes are applied using the typical git add and commit commands.
4. The repository is seeded for others, and since the web app does not recognize this project yet, the user is prompted to enter a name and description.

```
isaac@tux-dev:~/Documents$ gittor init project
isaac@tux-dev:~/Documents$ cd project/
isaac@tux-dev:~/Documents/project·main
·$ echo "Hello, World!" > file.txt
isaac@tux-dev:~/Documents/project·main
·$ git add file.txt
isaac@tux-dev:~/Documents/project·main
·$ git commit -m "Hello message"
[main 8af6dc6] Hello message
 1 file changed, 1 insertion(+)
 create mode 100644 file.txt
isaac@tux-dev:~/Documents/project·main
·$ gittor seed
GitTor service started.
Torrent name: project
Torrent description (optional): Simple hello project
Seeding Repository!
Repository ID: 26671ac0e1a590bba36f97d7ac9c29e51382254f
Torrent File:
/home/isaac/.config/gittor/repos/26671ac0e1a590bba36f97d7ac9c29e51382254f.to
rrent
Magnet Link:
'magnet:?xt=urn:btih:8b45d8b3ebdcad3b8f9733145fe1daedb094b500&xt=urn:btmh:12
```

```
20eac8460ca2bbe1b0430791a1e66f51fe22789d3320dfe6fdef78df699f5f1af2&tr=https%
3a%2f%2ftr.highstar.shop%3a443%2fannounce&tr=https%3a%2f%2ftracker.gcrenwp.t
op%3a443%2fannounce&tr=https%3a%2f%2ftr.nyacat.pw%3a443%2fannounce&tr=https%
3a%2f%2ftracker.moeblog.cn%3a443%2fannounce'
```

In the output, we see three different identifiers for the seeded repository: the repository ID, torrent file, and magnet link. Any one of these could be provided to another user to leech the repository and make changes; however, each has its trade-offs.

The repository ID is by far the shortest, simplest, and, as a bonus, it doesn't change when new updates are made to the repository. The trade-off is that it relies on the web application being functional, since it effectively translates these repository IDs into the most recent torrent files.

As for the magnet link and torrent file, they are quite similar in most ways except that the magnet link is a single string of text that can be easily shared with others rather than an entire file.

The recommended workflow is to use repository IDs, as they are the simplest and will only cause issues if the web application goes down. However, if there is an issue, use the magnet links until it is resolved, then return to the repository IDs. In the end, though, these are just different identifiers that can be transitioned between easily and serve primarily to give our users more options than anything else.

As far as the leeching side is concerned, this is typically what that would look like:

```
isaac@tux-dev:~/Documents$ gittor leech \
26671ac0e1a590bba36f97d7ac9c29e51382254f project
GitTor service started.
Leech complete. Saving session state...
Closing leeching client...
isaac@tux-dev:~/Documents$ cd project/
isaac@tux-dev:~/Documents/project·main
·$ cat file.txt
Hello, World!
isaac@tux-dev:~/Documents/project·main
·$ git log
commit 8af6dc666076f24bbd43160cb54d070d1092f077 (HEAD -> main, origin/main,
origin/HEAD)
Author: Isaac Denning <117934828+idenning2003@users.noreply.github.com>
Date:   Wed Apr 22 16:13:20 2026 -0500

    Hello message
```

Another important thing to note about leeching is that once a repository has been leeched, the user can simply run 'gittor leech' within the repository, without any identifier, and the repository ID will be automatically used to retrieve the latest state via torrenting.

The last command used frequently is the login command, which connects the CLI to the API. This is a basic command, but since session tokens regularly expire, it may need to be run daily.

```
isaac@tux-dev:~/Documents/project·main
·$ gittor login
Email or username: isaac
Password:
Login successful.
```

While not likely to be used often, there is one final command that may be useful to know: the service subcommand. This allows users to check on and manage the seeder service's state, which is best practice to restart after any changes to the '.gittorconfig' file.

```
isaac@tux-dev:~/Documents$ gittor service status
up
isaac@tux-dev:~/Documents$ gittor service stop
GitTor service stopped.
isaac@tux-dev:~/Documents$ gittor service status
down
isaac@tux-dev:~/Documents$ gittor service start
GitTor service started.
isaac@tux-dev:~/Documents$ gittor service restart
GitTor service stopped.
GitTor service started.
```

### 10.1.4. GitTor Web Self Host

Self-hosting the web application requires only one dependency, Docker. Docker is a large tool with different installation instructions depending on your environment, so please refer to Docker's official installation instructions for your system.

Once Docker  has been successfully installed on your system, the application can be started with a very simple set of commands:

```
git clone https://github.com/GitTor-ISU/GitTor-Web.git
cd GitTor-Web
./tools/run-docker-prod.sh
```

At the end, the program will output the administrator's login information:

```
API_ADMIN_USERNAME: admin
API_ADMIN_EMAIL: admin@gittor
API_ADMIN_PASSWORD: m7rlsNej69AU3HVS
```

### 10.1.5. GitTor Web Usage

Once the web application is self-hosted, you can access it via `https://localhost/` and any device with connection can access it through the host device IP unless specified otherwise. After you connect to the UI, it provides an interface to access all API functionality, from account management to viewing and downloading repositories.

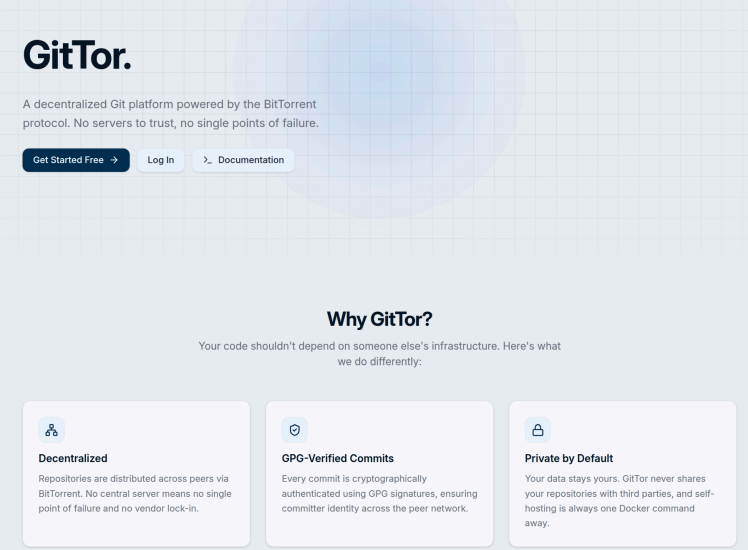


***Figure 16.*** *Web application landing page.*

Assuming the latest version of GitTor is being used, interaction begins on the main landing page. This page gives a brief overview of GitTor and direct links to everything a visiting user would need: links to register (for first-time users) and log in (for returning users), and a link to the documentation on setting up the GitTor CLI, which can be viewed without any authorizations.

When registering or logging in, you will need to provide information such as a username, email address, and password to access additional features on GitTor. Login sessions are stored for up to a week, so inactivity for more than a week will require the user to reauthorize on their next visit.

GitTor works smoothly on most browser-enabled devices, delivering a consistent experience across various screen sizes. It includes a responsive sidebar, or a top navigation bar on smaller screens, that provides easy access to all key features of the GitTor web application, including the repository list page, documentation page, and settings page.

Once you have logged in, you will be redirected to a page listing your repositories. This page can also be navigated to using the home or repositories buttons. This list will likely be empty unless you have already seeded a repository using the GitTor CLI connected to the same API. On this screen, you can view and sort your repositories, and retrieve the repository ID for cloning via the CLI.

The documentation page can be navigated to from the sidebar about button or the landing page documentation button, and it provides all the details on how to use GitTor, including a link to clone the GitTor CLI repository on your own system.

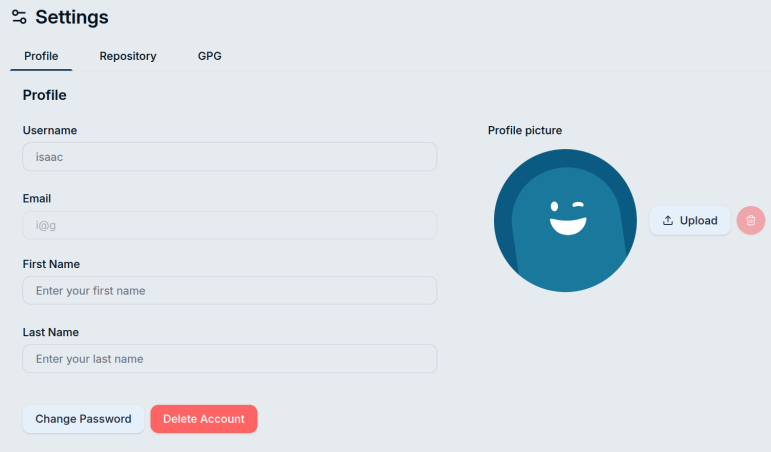


***Figure 17.*** *Web application user settings page.*

Lastly, the settings menu can be accessed from the sidebar settings button or profile picture icon. In the profile tab, you have the option to update any of the information provided during registration except for your email address. In addition, you can update the default profile picture to a custom one of your choice. You can revert to the GitTor-provided default profile picture by clicking the delete button next to the profile picture. In the repository tab, you can edit the metadata of any of your existing repositories, as well as delete them from the database.

## 10.2. Alternative/initial version of design

Throughout the development of GitTor, our understanding of the BitTorrent protocol, network latency, and cryptographic validation deepened significantly. As a result, several of our initial architectural designs were revised, scrapped, or scoped down. Below are the major alternative versions we considered, along with the reasoning behind their revisions.

### The Hybrid Synchronization Model

From the very beginning of the project, we considered a "Hybrid Model" to mitigate peer discovery latency. In this version, the system would default to traditional, high-speed HTTP or SSH connections to a central server for daily commits and pulls. The torrenting protocol would only activate as an automatic fallback if the central server went offline or attempted to block a user's access.

This approach was abandoned following a group vote early in the semester. We realized that relying on a centralized server for primary operations would allow us to bypass the hardest technical hurdles of peer-to-peer networking. We chose to commit exclusively to the pure torrenting protocol to demonstrate the viability of a fully decentralized environment clearly. Scrapping the hybrid model forced us to dedicate every part of our development to addressing the challenges of decentralization, even though we knew it would lead to worse latency.

### Commit-by-Commit GPG Rollback Validation

To prevent unauthorized users from pushing changes to a repository, we designed a validation system using a file containing public GPG keys. The initial design for the CLI validation command involved "rolling back" the repository through each commit, checking its cryptographic validity step by step until it reached either an already-validated commit or the very beginning of the repository's history.

This approach failed to meet our specifications for two reasons. First, the specific cryptographic C library we used lacked Windows support, severely limiting cross-platform compatibility. Second, the computational overhead of physically rolling back repository states was far too intensive. We revised this design into a theoretical approach—storing the authorized public GPG keys directly within the commit metadata or descriptions so validation could happen incrementally without rollbacks. However, due to time constraints, GPG validation was not implemented.

**Web UI Code View**

Our original project goals for the Spring Boot and Angular Web App closely mirrored GitHub's functionality: users would be able to view raw repository contents, file trees, and pull requests directly in the web browser.

Before learning more about the intricacies of the BitTorrent protocol, we did not fully appreciate the level of work required to implement torrenting features within a Java backend. Furthermore, this API functionality was bottlenecked; it could not be developed until the CLI's seeding mechanics were completely finalized. By the time the CLI was reliably seeding, we realized there wasn't enough time to build a torrent client into the API and add the code view on the UI.

## 10.3. Other considerations

| Component | Tool | Purpose |
|---|---|---|
| CLI Code | C | Low level CLI logic |
| CLI Build | Makefile | Build and link CLI binaries |
| CLI Libraries | Libtorrent, Glib, Gio, Libgit2, libcurl | Networking, torrenting, Git integration, utility functions, etc. |
| CLI Testing | Unity | Unit tests for CLI functionality |
| CLI Coverage | Gcovr | Measure coverage of tests and generate reports |
| CLI Memory Analysis | Valgrind | Detect memory leaks and issues |
| CLI Static Analysis | CppLint & Clang-Tidy | Enforce C coding standards |
| API Code | Java | High level API logic |
| API Build | Maven | Build and dependency management |

| | | |
|---|---|---|
| API Framework | SpringBoot | REST API framework with database integrations |
| API Specification | OpenAPI | Auto-generate API docs, client & server stubs |
| API Testing | Junit | Integration testing |
| API Coverage | Jacoco | Measure coverage of tests and generate reports |
| API Static Analysis | Checkstyle | Enforce Java coding standards |
| UI Code | Typescript/ HTML | Frontend logic and structure |
| UI Framework | Angular | SPA frontend framework |
| UI Libraries | Talwind & ZardUI | Styling and UI components |
| UI Runtime | Nginx | Web server plus reverse proxy to API |
| UI Testing | Cypress | End-to-end testing |
| UI Static Analysis | Eslint | Enforce coding standards and linting |
| DB | PostgreSQL | Relational database storage |
| S3 | Minio | Large object storage for repositories/torrents |
| Containerize | Docker Compose | Manage and Deploy multiple containers together |
| CI/CD | GitHub Actions | Automated checks, testing, builds, and deployments |

***Table 7.*** *Development tools used.*

### 10.4. Code

**Kanban Board:** https://github.com/orgs/GitTor-ISU/projects/1

**CLI Repository:** https://github.com/GitTor-ISU/GitTor-Cli

**Web Repository:** https://github.com/GitTor-ISU/GitTor-Web

## 11. Team

### 11.1. Team Members

- Jayson Acosta
- Seth Clover

- Isaac Denning
- Cameron Gilbertson
- Tyler Gorton
- Phu Nguyen

## 11.2.Required Skill Sets for Your Project

Development in the CLI requires skills in:

- **C Programming:** Strong knowledge of memory management, pointers, and low-level design.
- **C Libraries:** Able to integrate and use C libraries for complex functionality.
- **Unit Testing:** Can use unit testing to ensure proper functionality of features.
- **Build Management:** Understands Makefile syntax and can use it to compile the project with the necessary libraries.

Development of the web application requires skills in:

- **Java & Spring Boot:** Can use Java with Spring Boot to create a complete REST API.
- **Angular 20:** Knowledge of modern component-based UI design principles using Angular.
- **OpenAPI:** Understanding of OpenAPI specifications and how they can be used to maintain API integration.
- **End-to-End Testing:** Able to use Cypress testing to develop a complete test suite.

General knowledge requirements:

- **P2P & BitTorrent:** Knowledge of the P2P BitTorrent process of sharing data.
- **Agile Principles:** Can use iterative development practices to complete a software project.

## 11.3.Skill Sets covered by the Team

Development in the CLI requires skills in:

- **C Programming:** Everyone
- **C Libraries:** Isaac Denning and Seth Clover
- **Unit Testing:** Everyone
- **Build Management:** Isaac Denning and Seth Clover

Development of the web application requires skills in:

- **Java & Spring Boot:** Everyone
- **Angular 20:** Phu Nguyen, Isaac Denning
- **OpenAPI:** Isaac Denning, Phu Nguyen
- **End-to-End Testing:** Phu Nguyen, Isaac Denning, and Seth Clover

General knowledge requirements:

- **P2P & BitTorrent:** Isaac Denning, Phu Nguyen
- **Agile Principles:** Everyone

## 11.4. Project Management Style Adopted by the team

Our project is developed in an Agile style of management. However, unlike most Agile implementations, we do not use story pointing, retrospective, or refinements.

## 11.5. Initial Project Management Roles

- **Team Lead** - Isaac Denning
- **Web Application Lead** - Phu Nguyen
- **CLI Application Lead** - Cameron Gilbertson
- **Record Keeper** - Tyler Gorton
- **Quality Assurance Lead** - Jayson Acosta
- **Technical Research Lead** - Seth Clover

## 11.6. Team Contract

### 11.6.1. Team Procedure

#### 11.6.1.1. Day, time, and location (face-to-face or virtual) for regular team meetings:

We will meet every Thursday in the library or virtually, depending on the needs of the meeting.

#### 11.6.1.2. Preferred method of communication updates, reminders, issues, and scheduling (e.g., e-mail, phone, app, face-to-face):

Our primary form of communication will be through a Discord server.

#### 11.6.1.3. Decision-making policy (e.g., consensus, majority vote):

Decisions will be made by majority vote.

#### 11.6.1.4. Procedures for record keeping (i.e., who will keep meeting minutes, how will minutes be shared/archived):

Audio recordings will be taken for each meeting and shared on some cloud storage like CyBox.

### 11.6.2. Participation Expectations

#### 11.6.2.1. Expected individual attendance, punctuality, and participation at all team meetings:

Individuals will notify the rest of the team if they know they are not going to be available for a meeting.

#### 11.6.2.2. Expected level of responsibility for fulfilling team assignments, timelines, and deadlines:

Individuals will notify the rest of the team if they suspect that they won't be able to complete their tasks by the original expected date.

#### 11.6.2.3. Expected level of communication with other team members:

Individuals are expected to communicate with other team members when available. Communication will be done through the GitTor Discord server, as well as during class.

#### 11.6.2.4. Expected level of commitment to team decisions and tasks:

Individuals are expected to contribute to decisions and tasks when available. Each individual will take responsibility to assign themselves a ticket on the kanban board and ask for help or assist other team members where appropriate.

### 11.6.3. Leadership

#### 11.6.3.1. Leadership roles for each team member:

- **Team Lead** - Isaac Denning
- **Web Application Lead** - Phu Nguyen
- **CLI Application Lead** - Cameron Gilbertson
- **Record Keeper** - Tyler Gorton
- **Quality Assurance Lead** - Jayson Acosta
- **Technical Research Lead** - Seth Clover

#### 11.6.3.2. Strategies for supporting and guiding the work of all team members:

There is a questions channel on the Discord server that team members can post to.

#### 11.6.3.3. Strategies for recognizing the contributions of all team members:

All tasks will be put onto our GitHub project here, picked up by team members, and tagged as "Done" when the task is completed.

### 11.6.4. Collaboration and Inclusion

#### 11.6.4.1. Describe the skills, expertise, and unique perspectives each team member brings to the team:

**Jayson Acosta** - I have worked on many full-stack applications, and have used C in many of my classes. I will be bouncing around with the CLI and the Web Application, but my primary focus will be on the web application

**Seth Clover** - I bring project experience working with full-stack web apps using TypeScript, backend development using Java with Springboot, and low-level systems programming with C++ and C. I will be spending likely an equal amount of time between the CLI and Web App, with a focus on researching and prototyping.

**Isaac Denning** - I have worked on a few full-stack applications, as well as have a lot of C experience. As the person who formulated the original idea for this project, I give the perspective of envisioning the project in its entirety.

**Cameron Gilbertson** - I have experience using C, C#, C++, and Java through my classes and internships. Based on my experience I will be working mostly on the CLI portion of this project.

**Tyler Gorton** - My experiences have ranged from full-stack web development to low-level graphics programming, and I am comfortable with JavaScript/TypeScript, Java, C, C++, and Rust. I expect to work on both the CLI and the web app, with a slight focus on the web app.

**Phu Nguyen** - Full-stack experience through internships as well as experience with C through classes and personal projects. I will focus more on the web-app part, as that is where I have the most experience.

#### 11.6.4.2. Strategies for encouraging and support contributions and ideas from all team members:

Team members can make suggestions in Discord, create their own tasks, make comments on tasks, and request changes on Pull Requests.

#### 11.6.4.3. Procedures for identifying and resolving collaboration or inclusion issues:

If an individual has any conflicts within the team, they may either: bring the issue up to the team and see if there is a solution that can be agreed upon, or bring the issue up to TAs or professors of the class to see if they can be of assistance. If no solution is found when bringing the issue up to the team, TAs or professors will be brought in to help.

### 11.6.5. Goal-Setting, Planning, and Execution

#### 11.6.5.1. Team goals for this semester:

Get a functional system that accomplishes the core functionality. It does not have to do everything, it does not have to be pretty, and there may be bugs.

#### 11.6.5.2. Strategies for planning and assigning individual and team work:

Tasks will be created on our GitHub project here, and team members will be able to self-assign which tasks they wish to work on.

#### 11.6.5.3. Strategies for keeping on task:

Our weekly meetings will go over goals for tasks to be completed within the upcoming week.

### 11.6.6. Consequences for Not Adhering to Team Contract

#### 11.6.6.1. How will you handle infractions of any of the obligations of this team contract?

Infractions will first attempt to be handled within the team; however, if the issue continues, TAs or professors will be brought in to help.

#### 11.6.6.2. What will your team do if the infractions continue?

Contact TAs or professors to see if they can help or have any advice.

**********

*a) I participated in formulating the standards, roles, and procedures as stated in this contract.*
*b) I understand that I am obligated to abide by these terms and conditions.*
*c) I understand that if I do not abide by these terms and conditions, I will suffer the consequences as stated in this contract.*

1) Jayson Acosta_____ DATE 09/14/2025

2) Isaac Denning_____ DATE 09/15/2025

3) Seth Clover_____ DATE 09/15/2025

4) Phu Nguyen_____ DATE 09/16/2025

5) Cameron Gilbertson_____ DATE 09/16/2025

6) Tyler Gorton_____ DATE 09/16/2025