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 is to eliminate the issue of a single point of failure, which affects other platforms such as GitHub and GitLab. The problems that users have identified with this approach are 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 your only means of collaboration, they are the judge, jury, and executioner. Even if they don't change the state of your repositories, they can decide to block your commits, make terrible commits in your name, or delete your repositories (ideally, you would still have an instance on your machine).

To address all of these issues, we aim to minimize the power of the hosting platform as much as possible. Instead of hosting your repository through GitHub or GitLab, you and everyone who contributes to it will host the repository and share it with others via a P2P protocol. When sharing the repository, your computer will seed it with a .torrent file. By sharing this .torrent file with someone else, they will be able to leech the repository, downloading it directly from you.

This strategy clearly resolves issue one; even if your computer is off, others can seed the repository, allowing anyone with the .torrent file to leech it. The solutions to 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 you are the one hosting the repository (along with everyone else who has access to it), you are the judge, jury, and executioner. You control the state of the repository, and it is up to the other contributors to decide if they wish to leech your changes and work off of them.

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

web application that allows 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 will be available, allowing alternatives to be easily created and switched to in minutes.

Another concern is 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 want to limit its power as much as possible. To ensure that only authorized individuals can commit to a repository, a file in the repository will store the public GPG keys of those allowed to commit, which can be 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 the file. When a new GitTor repository is initialized, it will initially contain 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 will also enable 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 intend to be using:

- 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 20 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 involves numerous systems that none of our team members have 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 will provide all our team members with the opportunity to delve deeper into the Git protocol, a system we have all used but can always benefit from understanding further.

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

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 it has been 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 can take anywhere from hours to days. On many professional projects, this amount of downtime without collaboration can cost hundreds of 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 aims to resolve all of these issues. Instead of hosting your project through GitHub or GitLab, you and everyone on your team will host it and share 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, will 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. In this new approach, if everyone else in the team unanimously decides not to use the contributions of one team member, then the result would be identical to what it was before. However, this seems like more of an issue with the team itself rather than the system. Since team members are the ones directly sharing the project, there is no external way to prevent someone within the team from contributing.

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 will be 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 will be 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 will 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 will 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 will 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 will 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 will 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.
  Since our project is entirely software-based. Nearly everything can be tested using this standard as a guideline. We'll use this standard to make comprehensive tests.

3. Project Plan

3.1. Project Management/Tracking Procedures

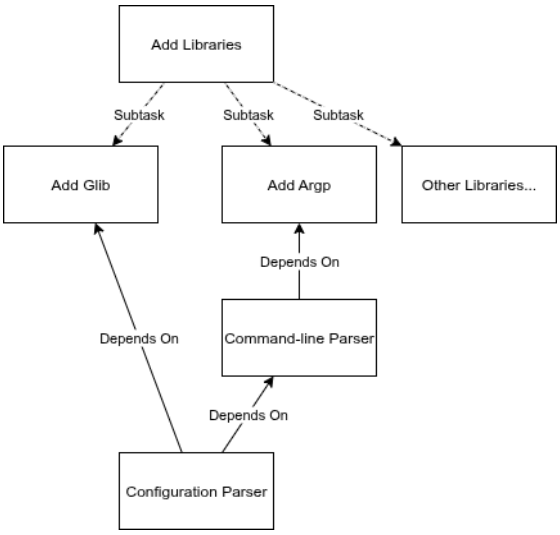
For our project, we are employing an agile management style to deliver working software early and often, enabling us to refine features and respond quickly to issues as they arise. This style also

ensures that both our CLI and web application evolve together as new requirements and challenges emerge.

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

3.2. Task Decomposition

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



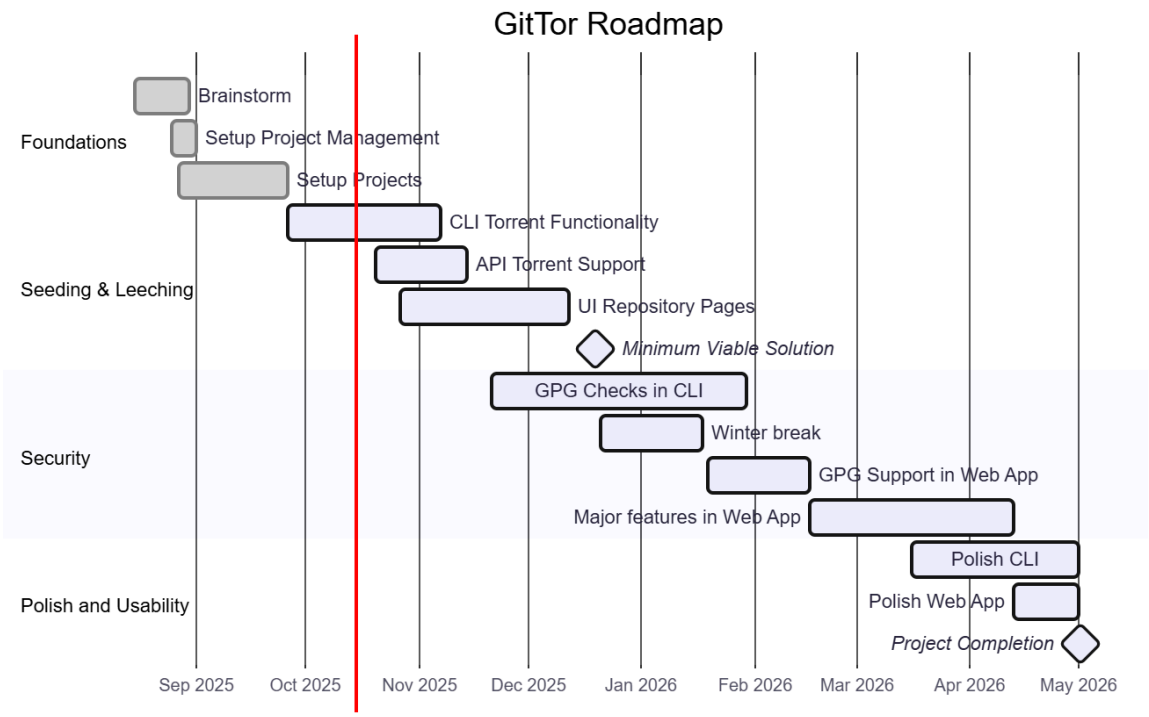
An example of this task decomposition can be seen in the implementation of our configuration file parser in the CLI. Since the parser will utilize the Glib library, this task depends on adding Glib, which is a subtask of adding necessary libraries. There also needs to be a way to call the configuration file parser, which requires a command-line parser that uses the argp library. Once again, this library is a subtask of adding necessary libraries.

3.3. Project Proposed Milestones, Metrics, and Evaluation Criteria

Although we do not have a complete task decomposition due to our agile structure, we have laid out the necessary milestones for our project.

- **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



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. 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

3.6. Personnel Effort Requirements

Major Task	**High-Level Description**	**Related Sub Tasks***	**Total Time Required**
Brainstorm	Beginning Stages of the GitTor project. Laid out our project design	- Come up with ideas - Pitch our ideas to faculty	20 hours
Set up Project Management	Setting up our project environments	- Set up a GitHub Repository - Set up our Kanban and Discord	15 hours
Setup Projects	Setting up the framework for GitTor	-Setting up the framework for the CLI -	30 hrs
CLI Torrent Functionality	Develop a working CLI that can functionally work as a git service	-Research Torrent Integrations - Torrent Demo	100 hrs
API Torrent Support	Develop a working API to be able to call our CLI	- Create a design for the API with methods - Implement the design to work with our CLI	60 hrs
UI Repository Pages	Develop a working UI for our repository for the WebApp	- Choosing a UI - Implementing basic functionality for users to use the UI Repository Page (such as login)	90 hrs
Minimum Functioning Implementation	System Integration between the UI and the CLI.	- Have a functioning CLI - Have a functioning WebApp - Integrate the CLI and WebApp together	
GPG Checks in the CLI	Enable GPG Checks for the CLI	*TBD	70 hrs

GPG Support in the WebApp	Enable GPG Support for the WebApp	*TBD	60 hrs
Major Features in Web App	Complete major features to fully complete our WebApp	*TBD	120 hrs
Polish CLI	Finish the CLI while creating readable and maintainable code	- Complete the CLI - Add READMEs *TBD	80 hrs
Polish Web App	Finish the WebApp while creating readable and maintainable code	- Complete the WebApp - Add READMEs *TBD	70 hrs
Documentation	Create final documentation pieces	- Create a documentation explaining GitTor in either an extra document or as a Wiki	30 hrs

**Some of the sub-related tasks have not been concretely laid out. Especially for the second-semester major tasks.*

3.7. Other Resource Requirements

Our project is entirely software-based, and we don't expect to require any physical resources.

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 other major repository hosting platforms such as GitHub and GitLab. But we aim to solve the need for a centralized server owned by a large corporation like Alphabet Inc. 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 needs to provide a secure way for users to store and share their code repositories. This will provide peace of mind that their code is safe and secure.	• HTTPS communication between the server and users • P2P secure communications between users • 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 • Decentralized Network
Environmental	N/A	N/A
Economic	Our project is an open-source software product. This will make it available to hobbists as well as professionals.	• Our product will be free to use

4.1.2. Prior Work/Solutions

Our project is an alternative to common repository hosting platforms such as GitHub [1] 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." [2]. Using this protocol will prevent 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 [3]. 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 enables continuous repository seed availability even when users aren't actively running commands but adds 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. Proposed 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.

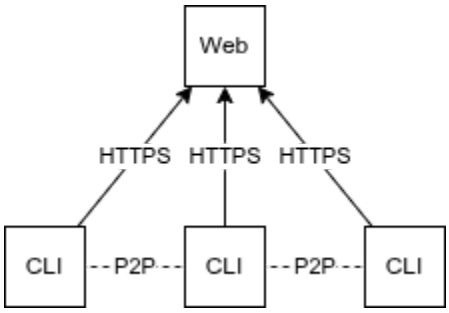
The command-line tool depends on a parser to read user input and trigger specific operations. Each operation manages different aspects of repository sharing, configuration, and authentication. The design focuses on keeping the tool lightweight and direct. It supports sharing repositories over a Peer-To-Peer 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. For larger or unstructured data, such as repository previews, the system uses Minio Simple Storage Service.

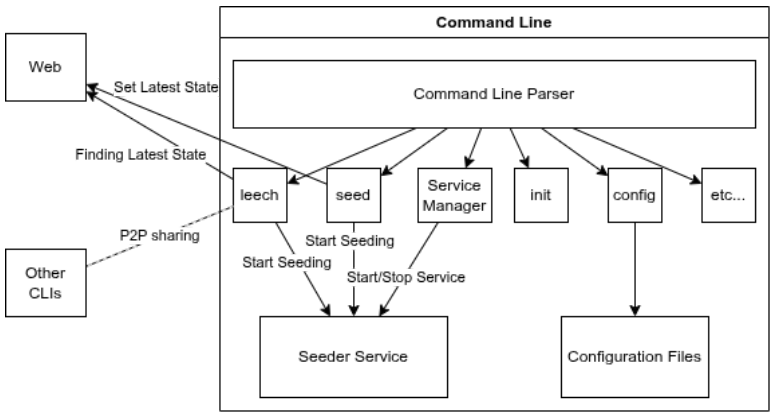
All components run inside Docker containers managed with Docker Compose. This setup isolates services, simplifies deployment, and improves scalability. The design creates a clear separation between the user interface, API, and storage, ensuring the system remains organized and efficient.

4.3.2. Detailed Design and Visual(s)

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.



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.

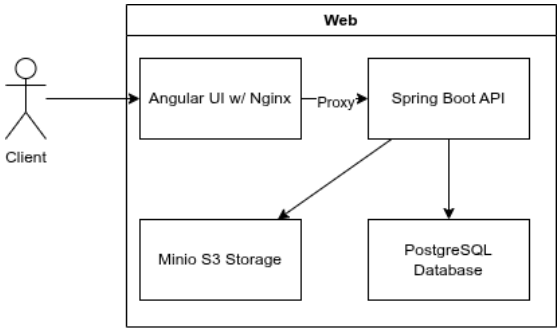
To understand why we need this separate process, we must first realize that the basic command-line program must start and stop regularly as users call it. However, to properly seed repositories on a Peer-To-Peer network, there must be a process actively seeding at all times.

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

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.



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 Concern and Development

The current design addresses all core function requirements. The P2P BitTorrent protocol ensures decentralized repository sharing without third-party dependencies, and the GPG signature verification provides cryptographic authentication for commits.

The first concern for product delivery is the seeder service reliability. The seeder service must remain stable across system reboots and handle issues such as network interruptions, a lack of disk space, and concurrent seeding of repositories. Failures with this service undermine the value of using GitTor. To address this concern, we will implement comprehensive error handling and automatic recovery, along with integrations for automatic restart on failure.

The second concern is with GPG verification. Validating commit signatures across large repository histories could introduce latency. We need to ensure that the verification process scales efficiently and does not create a bottleneck during leeching. To address this concern, we plan to implement incremental verification that caches previous results and only validates newer commits.

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.

4.5. Design Analysis

So far, the groundwork for GitTor has been laid by creating the CLI and web-app repos with foundational implementations.

We have built the basic API, database, testing, general layout, theming, and authentication for the web-app portion. What we have created for the web-app design is working as intended, and we are still on track to continue following the original design. We haven't built important pages(including the homepage) and API to CLI compatibility. Our plans for the web-app are to implement these features, including figuring out the intricacies of getting leeching working.

For the CLI, we've built the command-line parser, testing, and have begun the process of torrenting repos. What we have created for the CLI design is working as intended, and we are still on track to continue following the original design structure. We haven't added seeding and leeching repositories yet, and the plan is to get that done soon to unblock other work areas. Other than that, the CLI plans to implement GPG key validation for each commit, which at this point looks feasible.

Overall, the two projects are steadily progressing according to the design set, and we plan to continue with the aid of our tools we have set up and the designs we have laid out.

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 will function differently, their overall goal will be the same: to ensure that the requirements for the developed feature are met. The sections below will outline the various forms of testing we use and explain how they are best suited for each part of our application.

For each new task on our Kanban board, we will mark the form of testing that needs to be implemented and guarantee that those tests have been created and passed before merging. This will help us keep track of our project and ensure old features aren't unintentionally broken in the implementation of a new feature.

5.1. Unit Testing

In general, our project avoids unit testing where possible, as the more accurate the tests are in simulating the whole environment, the better they are. This isn't to say we don't have unit testing, as it's often the best option available; however, when integration testing is a viable alternative, it will be used on our project.

Our CLI is an excellent case where unit testing is 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 is built. For this reason, the CLI's tests are 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 are primarily focused on preventing memory leaks and infinite loops more than anything else. To handle this, our tests are run on top of Valgrind, which analyzes the memory usage and detects any errors.

5.2. Interface Testing

As our project evolves, we will continue to develop additional interfaces and tests for these interfaces. However, as of now, our best example of interface testing 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 an interface that allows the program to upload, download, or delete these objects.

The only complication is that there are actually three different S3 implementations, depending on the runtime: in-memory, in files, or in MinIO. In the future, this could even expand to include Amazon S3 as well.

Since we don't want the program to run differently between each of the runtimes, we need a test suite to ensure that none of the implementations function differently from the others, at an interface level. This test suite doesn't even guarantee that they function "properly" it just checks that they all function the same when provided 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 is the API. It is necessary to ensure that all controllers function as expected, which can be extremely tedious to do manually especially as the system grows and changes. For this reason, almost all of our API tests are examples of integration tests.

Developed using JUnit, the tests send REST requests to the controller layer, which calls the service layer, repository layer, and finally the database/S3. For these tests to run correctly, all pieces of the API must be integrated and running together. However, most of the time, we don't want to set up an entire PostgreSQL database or MinIO instance to test the application. In these cases, our system simply replaces 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 is most achievable with the web user interface and its interaction with the API.

To create these tests, we will be using Cypress, which runs the entire web user interface and interacts with it as an automated user ensuring the proper information appears on screen. This approach provides confidence that both the frontend and backend are functioning together correctly under real-world conditions.

5.5. Regression Testing

To ensure that new changes do not break existing features, the entire test suite must be run, which can be pretty tedious. To manage this, we have set up Github workflows which run all our tests for each new commit to the repository, as well as other checks. For a pull request to get merged into the main branch, all checks must pass. Along with ensuring new changes do not break existing features, it also verifies that the latest tests created with this feature also pass.

5.6. Acceptance Testing

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

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

5.7. Security Testing

For every system, security testing appears in a different form. For our system, and given our resources, the best form of security testing we can realistically have is primarily static code analysis. This process searches our codebase for common security vulnerabilities as well as other issues, which we have implemented in some form for each environment: CLI, API, and UI. As mentioned before we are using 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 always use more security testing, but at some point, we have to call it and focus our efforts elsewhere.

5.8. User Testing

Once our system reaches a minimum viable product we will move towards user testing to learn what needs to be improved and refined. Since our project's intended users are limited to software developers, so too will our user testing. We will gather up friends and other software developers to use our application and learn what gives them struggle, confusion, etc.

Once we have collected this data, we can work towards creating a plan to solve these problems to the best of our ability and creating future tasks. Improvement is a never ending process, so we may need to go through multiple rounds of user testing before finally reaching a design we are happy with.

5.9. Results

Currently, we have one Cypress test in the UI, 22 Unity tests in the CLI, and 304 JUnit tests in the API. The discrepancy between the API and the UI is because the API was created from a template by one of our team members, which we were able to build upon immediately unlike the UI which needed to be developed from the ground up. However, we expect that number to grow to a respectable size soon.

Our coverage reports indicate 100% coverage in the CLI and 94% in the API. The last 6% of the API consists mainly of unimplemented utility functions and almost unreachable error handlers.

Due to our extensive testing, we have identified only one bug to make it to main, which was caused by an inconsistency between mobile Safari's rendering and that of all other browsers. The remaining issues marked as bugs are 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 largely complete within the first month, in reality, this has been an ongoing process of improvement, and it has only recently reached a point where we are fully satisfied with our development environment.

6.2. Command Line Interface

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 is

not a requirement for our project, it would still be 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.

```
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.
```

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.

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

done, shutting down
```

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.

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 last major feature implemented in the command-line interface is the creation of a separate service process to be used for continuous seeding. This process must support inter-process communication from the CLI with multiple connections at a time. The only way to solve this is with multithreading. However, the main complexity of this feature is that it should be functional on both Linux and Windows, if possible. Because of this, the only proper way to manage inter-process communication is through ports and sockets, which introduces a whole new layer of complexity.

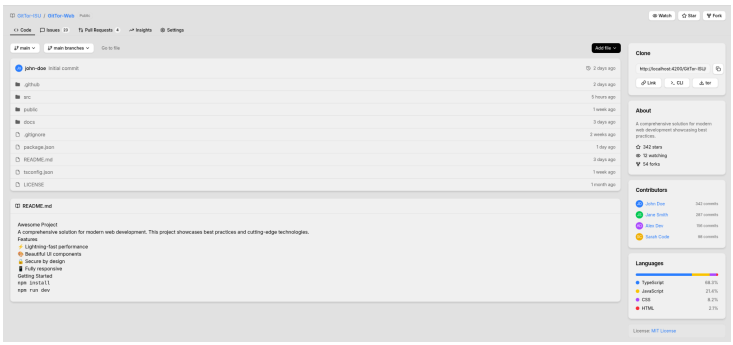
6.3. Web Application

Unfortunately, unlike the CLI, the setup for the web application was not limited to the items discussed in the section above. We knew that 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 and simple storage service, in addition to the API and UI containers.

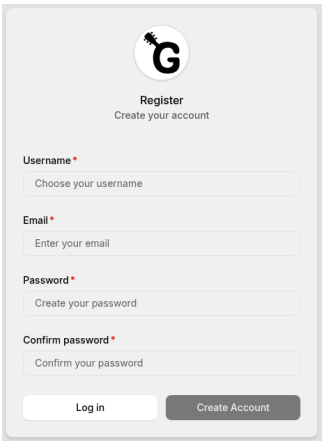
Another piece of setup we wanted was the ability to generate API connection code in the UI. We used the OpenAPI specification generated by Spring Boot, along with the OpenAPI command-line interface, to automatically generate TypeScript code in our frontend for our data transfer objects and API endpoints.

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. 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 being approved and will likely undergo further changes in the future once we can interact with them directly.



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.



To date, many pages are currently under construction; however, the login and registration pages have been fully completed, supporting both desktop and mobile device aspect ratios. These pages are also fully functional and use the backend's refresh token and JWT token for authentication.

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.
Financial Responsibility	Create it at a reasonable price for its value.	2. CLIENT & EMPLOYER: Act in the client's best	Our project is entirely open source ensuring that our users never

		interest.	have to pay a dime for it.
Communication Honesty	Be honest to your users.	4. JUDGMENT: Integrity in professional judgment. 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. 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.
Sustainability	Minimize environmental risks of the product.	1. PUBLIC: Act consistently with public interest. 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. 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.

Of these responsibilities, our team has been performing 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 we will need to evaluate in the future.

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, Cultural, Social	Supports global collaboration across cultures and communities by enabling decentralized participation without central censorship.	Network reliance and complexity could disadvantage users in regions with limited bandwidth or technical expertise.	Contributors can choose whom to collaborate with and which repositories to access, supporting self-directed participation.	Benefits and 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.

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 will 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 is partially mitigated by allowing users to control when and how much they seed, as well as by distributing the workload across multiple peers. We will still need to investigate further to see the actual environmental impact of our design.

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 semester, 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 semester. I make 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 semester 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 semester 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 semester 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. Closing Material

8.1. Conclusion

So far, we have implemented several core features across both the CLI and the web application. On the CLI side, we have a command line parser with sub-commands, implemented global and local configuration handling, prototyped cross-network torrent-based repository sharing, and developed a multithreaded background seeding service using ports and sockets for cross-platform inter-process communication. On the web side, we implemented full authentication with users, roles, authorities, and JWT/refresh tokens with login and registration pages on the UI. We created endpoints for uploading, retrieving, and updating repository torrent files.

Our overall goal is to develop a fully decentralized Git hosting platform where the repository state is distributed peer-to-peer through torrents, contributors are authenticated through GPG-verified signatures, and the web application functions solely as a discovery layer, rather than a central source of truth.

To complete the system, the CLI must implement seeding and leeching, which it's close to having, integrate the torrent file API requests, and perform GPG signature validation against each repository's contributor keyfile. On the API side, the backend needs the ability to leech repositories itself so it can validate and display their contents on the frontend. The frontend must then implement the repository-viewing pages and load the necessary data from the API.

8.2. References

[1] Callas, J., Donnerhacke, L., Finney, H., Shaw, D., & Thayer, R. (2007). OpenPGP Message Format. *Www.rfc-Editor.org*. https://doi.org/10.17487/RFC4880

[2] GitTor-ISU. (2025a). *GitHub - GitTor-ISU/GitTor-Cli: GitTor-Cli Implementation*. GitHub. https://github.com/GitTor-ISU/GitTor-Cli

[3] GitTor-ISU. (2025b). *GitHub - GitTor-ISU/GitTor-Web: GitTor-Web implementation*. GitHub. https://github.com/GitTor-ISU/GitTor-Web

[4] GitTor-ISU. (2025c). *GitTor*. GitHub. https://github.com/orgs/GitTor-ISU/projects/1

[5] ISO/IEC. (2023). *Information technology — Database languages SQL*. https://www.iso.org/standard/76583.html

[6] ISO/IEC/IEEE. (2022). *Software and systems engineering — Software testing*. https://doi.org/10.15199/48.2020.10.35

[7] Rescorla, E. (2018). The Transport Layer Security (TLS) Protocol Version 1.3. *Internet Engineering Task Force (IETF), RFC 8446*. https://doi.org/10.17487/rfc8446

[8] The Open Container Initiative. (2019a). *Distribution Specification*. Open Container Initiative. https://specs.opencontainers.org/distribution-spec/

[9] The Open Container Initiative. (2019b). *Image Format Specification*. Open Containers Initiative. https://specs.opencontainers.org/image-spec/

[10] The Open Container Initiative. (2019c). *Runtime Specification*. Open Containers Initiative. https://specs.opencontainers.org/runtime-spec/

[11] Thomson, M., & Benfield, C. (2022). HTTP/2. *Internet Engineering Task Force (IETF), RFC 9113*. https://doi.org/10.17487/rfc9113

[12] Wouters, P., Huigens, D., Winter, J., & Yutaka, N. (2024). OpenPGP. *Internet Engineering Task Force (IETF), RFC 9580*. https://doi.org/10.17487/rfc9580

8.3. Appendices

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

9. Team

9.1. Team Members

- Jayson Acosta
- Seth Clover
- Isaac Denning
- Cameron Gilbertson
- Tyler Gorton

- Phu Nguyen

9.2. 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.

9.3. 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

9.4. 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.

9.5. 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

9.6. Team Contract

9.6.1. Team Procedure

9.6.1.1. 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.

9.6.1.2. 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.

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

Decisions will be made by majority vote.

9.6.1.4. 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.

9.6.2. Participation Expectations

9.6.2.1. 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.

9.6.2.2. 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.

9.6.2.3. 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.

9.6.2.4. 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.

9.6.3. Leadership

9.6.3.1. 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

9.6.3.2. 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.

9.6.3.3. 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.

9.6.4. Collaboration and Inclusion

9.6.4.1. 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.

9.6.4.2. 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.

9.6.4.3. 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.

9.6.5. Goal-Setting, Planning, and Execution

9.6.5.1. 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.

9.6.5.2. 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.

9.6.5.3. Strategies for keeping on task:

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

9.6.6. Consequences for Not Adhering to Team Contract

9.6.6.1. 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.

9.6.6.2. 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