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: 11/3/2025

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

full-stack 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 full-stack 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 full-stack 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, etc.
    - Unity C testing library with Gcovr coverage reports and Valgrind memory analysis
    - CppLint, Clang-Tidy, and Clang-Format rules
    - Pipeline for rule checks, tests, reports, and builds
- Full-stack 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

List all standard circuit, hardware, and software practices used in this project.
List all the Engineering standards that apply to this project that were considered.

Summary of Requirements

List all requirements as bullet points in brief.

Applicable Courses from Iowa State University Curriculum

List all Iowa State University courses whose contents were applicable to your project.

New Skills/Knowledge acquired that was not taught in courses

List all new skills/knowledge that your team acquired which was not part of your Iowa State curriculum in order to complete this project.

Table of Contents

List of figures/tables/symbols/definitions (This should be the similar to the project plan)

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 seeders exist (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 4880 - OpenPGP Message Format
  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

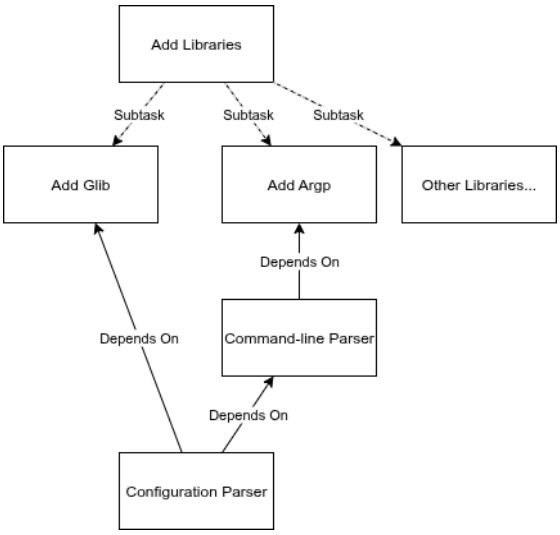
For our project, we are employing an agile management style to deliver working software early and often, enabling us to refine features and respond quickly to issues as they arise. This style also

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.



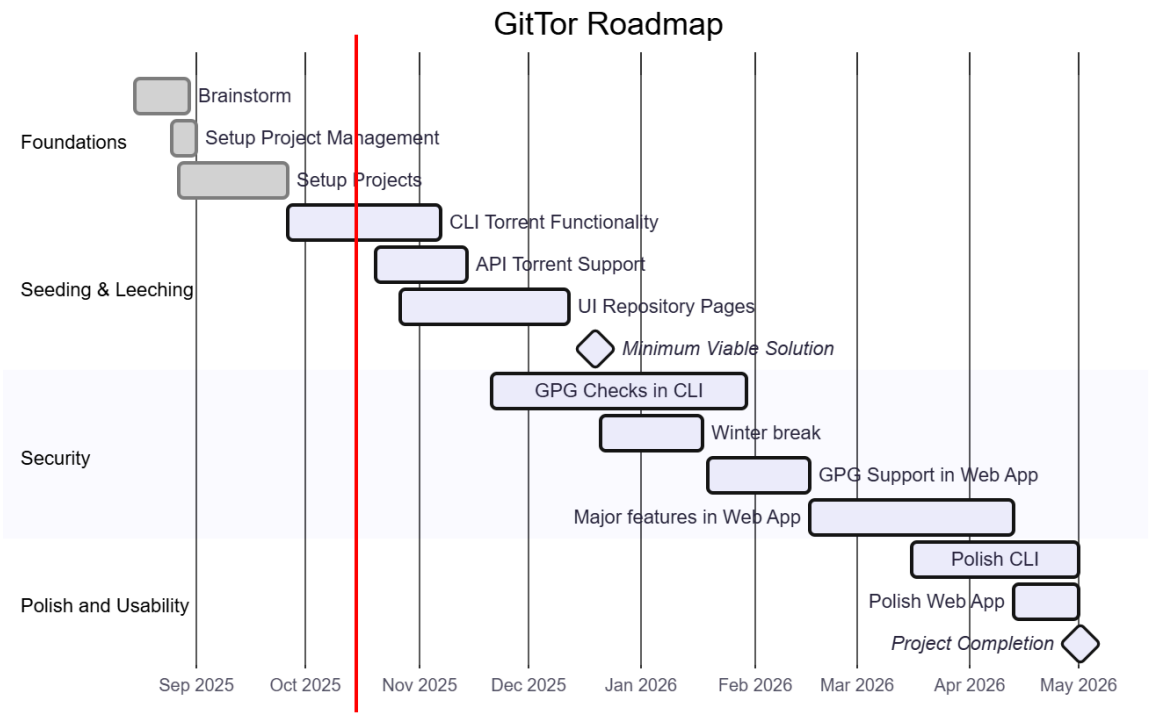
An example of this task decomposition can be seen in the implementation of our configuration file parser in the CLI. Since the parser will utilize the Glib library, this task depends on adding Glib, which is a subtask of adding necessary libraries. There also needs to be a way to call the configuration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

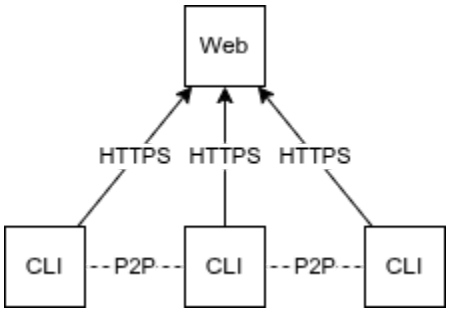
The command-line tool depends on a parser to read user input and trigger specific operations. Each operation manages different aspects of repository sharing, configuration, and authentication. The design focuses on keeping the tool lightweight and direct. It supports sharing repositories over a Peer-To-Peer 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.

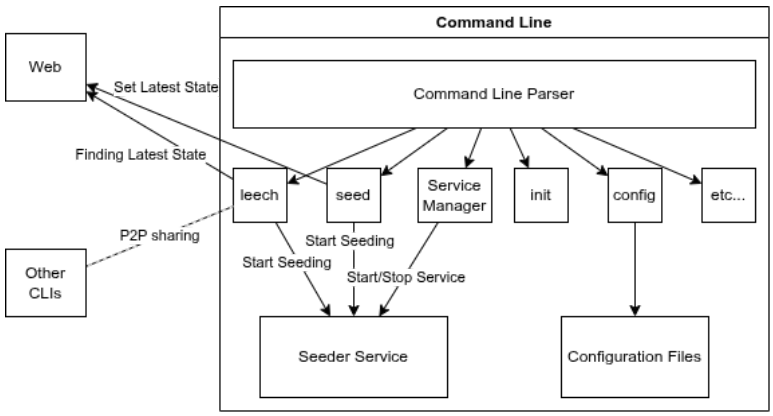
All components run inside Docker containers managed with Docker Compose. This setup isolates services, simplifies deployment, and improves scalability. The design creates a clear separation between the user interface, API, and storage, ensuring the system remains organized and efficient.

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.

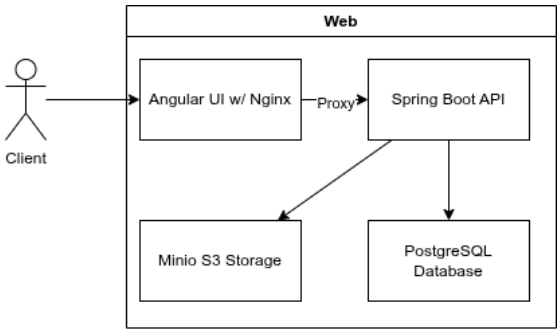


- The leech component determines the latest state of the repository using the web API, retrieves it from other contributors, and then instructs the seeder service to begin seeding.
- The seed component updates the state of the local repository according to the current changes, tells the seeder service 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 for helping manage custom configurations, repository initialization, and developer authorizations. This design provides only the necessary functionality for sharing repositories over a Peer-To-Peer BitTorrent protocol via a command-line tool, as the extra layers of complexity are not entirely needed; They're more of a nice-to-have.

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

Describe any (preliminary) implementations of your design thus far. Support any general, descriptive text with relevant images. If your project has inseparable activities between design and implementation, you can list them either in the Design section or this section.

7. Ethics and Professional Responsibility

Use this section to describe your considerations of engineering ethics and professional responsibility. Most importantly how are you defining engineering ethics and professional responsibility in the context of your project and what steps are you taken to ensure ethical and responsible conduct. Each section references one type of ethical/professional responsibility considerations. You may also use this introductory section to note any overarching ethical philosophy among your team.

7.1. Areas of Professional Responsibility/Codes of Ethics

This discussion is with respect to the paper by J. McCormack and colleagues titled “Contextualizing Professionalism in Capstone Projects Using the IDEALS Professional Responsibility Assessment”, International Journal of Engineering Education Vol. 28, No. 2, pp. 416–424, 2012

Pick one of IEEE, ACM, or SE code of ethics (all linked in class slides). Create a table, like Table 1 in the McCormack et al. (2012, pg. 418)) paper, with the following columns representing: Area of Responsibility (from the paper), Definition (in your own words), Relevant Item from Code of Ethics (from the Code of Ethics you selected, and description of how your team has interacted with that area of professional responsibility or adhered to that code during your project thus far.

In text below the table, describe one area in which your team is performing well. Describe what your team is doing and how that signifies strong performance. Also describe one area in which your

team needs to improve. Describe what your team is currently doing and what it should do in the future to improve.

7.2. Four Principles

Create a table with rows for each broader context area (see Section 4.1.1) and columns for each of the four principles (beneficence, nonmalificence, respect for autonomy, and justice; see Beauchamp, 2007). Within the table, identify at least one way each of the four principles applies to each of the broader context areas. Some principle-broader context connections might be more prominent than others, but you should be able to identify something for each table cell. Note: Your design may end up negative or neutral in some cell. For example, your product might perform poorly in environment-nonmaleficence because it utilizes natural resources without a positive/mitigating effect.

Below the table, note one broader context-principle pair that is important to your project. Briefly describe the benefit in that area you are working towards and how you will ensure it. Also note one broader context-principle pair in which your project/end design is or will be lacking. Describe either (a) how this negative is overcome by other positives in other areas of the project/design or (b) what your team must do to improve in this area.

7.3. Virtues

List and define at least three virtues that are important to your team. Describe what you will do or have done as a team to support these virtues among all team members.

Each team member should also answer the following:

- Identify one virtue you have demonstrated in your senior design work thus far? (Individual)
    - Why is it important to you?
    - How have you demonstrated it?
- Identify one virtue that is important to you that you have not demonstrated in your senior design work thus far? (Individual)
    - Why is it important to you?
    - What might you do to demonstrate that virtue?

8. Closing Material

8.1. Conclusion

Summarize the work you have done so far. Briefly re-iterate your goals. Then, re-iterate the best plan of action (or solution) to achieving your goals. What constrained you from achieving these goals (if something did)? What could be done differently in a future design/implementation iteration to achieve these goals?

8.2. References

List technical references and related work / market survey references. Do professional citation style (ex. IEEE). See link:
https://ieee-dataport.org/sites/default/files/analysis/27/IEEE%20Citation%20Guidelines.pdf

[1] GitHub, "About GitHub and Git," *GitHub Docs*, 2024.
https://docs.github.com/en/get-started/start-your-journey/about-github-and-git
[2] C. Staff, "What Is a Peer-to-Peer Network?," *Coursera*, Oct. 24, 2024.
https://www.coursera.org/articles/peer-to-peer
[3] "Radical," *Radicle.xyz*, 2025. https://radicle.xyz/ (accessed Oct. 28, 2025).

8.3. Appendices

Any additional information that would be helpful to the evaluation of your design document.

If you have any large graphs, tables, or similar data that does not directly pertain to the problem but helps support it, include it here. This would also be a good area to include hardware/software manuals used. May include CAD files, circuit schematics, layout etc,. PCB testing issues etc., Software bugs etc.

9. Team

Complete each section as completely and concisely as possible. We strongly recommend using tables or bulleted lists when applicable.

9.1. Team Members

9.2. Required Skill Sets for Your Project

(if feasible – tie them to the requirements)

9.3. Skill Sets covered by the Team

(for each skill, state which team member(s) cover it)

9.4. Project Management Style Adopted by the team

Typically, Waterfall or Agile for project management.

9.5. Initial Project Management Roles

(Enumerate which team member plays what role)

9.6. Team Contract

Team Members:

1. Jayson Acosta

2. Seth Clover
3. Isaac Denning
4. Cameron Gilbertson
5. Tyler Gorton
6. Phu Nguyen

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 (e.g., team organization, client interaction, individual component design, testing, etc.):

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 (e.g., how will a team member inform the team that the team environment is obstructing their opportunity or ability to contribute?):

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