University of Plymouth

School of Engineering, Computing, and Mathematics

COMP3000
Computing Project
2024/2025

Janus Version Control

Benjamin Sanders-Wyatt

10808929

BSc (Hons) Computer Science

Acknowledgements

I want to thank my initial supervisor, Thomas Wennekers, for his guidance during the early stages of this project and my current supervisor, Shaymaa Al-Juboori, for her support throughout the remainder of the project. I am also grateful to my peers for their encouragement and the participants who volunteered to test Janus and provided valuable feedback.

Abstract

The Janus Distributed Version Control System (DVCS) was developed to meet the growing demand for secure, on-premise version control solutions in enterprise environments. Traditional cloud-hosted DVCS platforms raise concerns over data sovereignty and security; Janus provides organisations with a self-hosted alternative that ensures codebases remain under their control. The project objectives include the development of a cross-platform command line interface (CLI) with an extensible plugin and an internal Dockerised secure web-based frontend, compliant with security standards. An Agile Scrum development methodology was used, allowing interactive refinement through continuous integration and user feedback.

Key achievements include successfully implementing the core CLI, web application and flexible plugin framework. The system was tested on multiple operating systems to ensure cross-platform functionality. Janus is compliant with relevant data protection regulations, GDPR, and the Data Protection Act, as well as security best practices, utilising robust authentication, encryption, and logging mechanisms to protect user data. Professional practices were upheld throughout the project, including thorough project management, systematic testing and comprehensive documentation. In summary, Janus fulfils its objectives by providing a secure, enterprise-level DVCS solution that organisations can host on-premise to maintain complete control over their code and development workflows.

Contents

WORD COUNT: **10412**

GitHub:

https://github.com/benjaminsanderswyatt/COMP3000-JanusVersionControl

1 Introduction

1.1 Background

Version control systems (VCS) are foundational to modern software development. Early centralised VCS relied on a single central repository as a source of truth, creating bottlenecks, single points of failure, and limited offline work (Chacon & Straub, 2020). In contrast, DVCS solutions allow each developer to have a full local copy of the project history, enabling offline commits, fast branching, and inherent redundancy (Loeliger & McCullough, 2023).

Recent industry trends show growing interest in on-premise DVCS solutions. Concerns over data sovereignty regulations (UK Government, 2018) and industry-specific rules mean that many companies cannot allow code or data to be stored on external systems. Surveys report that most IT leaders cite regulatory and compliance pressures as key reasons for preferring privately hosted code repositories (Ponemon Institute, 2023). High-profile incidents such as SolarWinds in 2020 and Log4Shell in 2021 have shown that reliance on external services can introduce critical security risks. By hosting systems entirely on-premise, enterprises gain greater assurance of security and auditability.

Security concerns around public cloud services amplify these pressures, with cloud environments increasingly becoming targets for malicious attacks. IBM's Cost of a Data Breach Report 2023 states that cloud misconfigurations and third-party vulnerabilities were major contributors to breaches, with the average breach costing organisations around £3.6 million in the UK (IBM Security, 2023). In addition, reports from Verizon (Verizon, 2024) and Gertner (Gartner, 2023) indicate that the most recent breaches involved cloud-hosted data or misconfigured cloud services. The rising frequency and cost of cloud-targeted breaches drive organisations to minimise exposure by keeping their code on-premise.

Together, these factors explain the shift in trends and the need for a self-hosted version control system designed to meet modern requirements while maintaining the collaborative benefits of distributed systems.

1.2 Problem Statement

Despite the widespread adoption of DVCS platforms, several limitations make many current systems suboptimal for large-scale enterprise deployment. The reliance on external cloud services exposes organisations to significant security vulnerabilities; their centralised nature makes them attractive targets for cyberattacks. Additionally, many available solutions offer only basic access control functionalities, which falls short of the stringent internal policies and granular permission management required by modern enterprises (Stöcklin, 2022).

Integrating public DVCS platforms into existing corporate infrastructures can be complex and resource-intensive. Modern enterprises increasingly require systems that support robust local operations through an intuitive Command Line Interface (CLI), containerised remote management, and the capability to incorporate customisable features through extensible plugins.

There is also a growing need for extensible systems which can be easily adapted as requirements evolve. These challenges demonstrate the necessity of Janus, a DVCS solution specifically designed for the enterprise environment.

1.3 Project Vision & Objectives

1.3.1 Project Vision

Janus is a secure and flexible Distributed Version Control System that allows enterprises to manage their codebases internally. It is named after the Roman god Janus, who is depicted with two faces, one looking into the past and the other into the future, symbolising the responsibilities a version control system holds. Janus aims to eliminate dependence on external cloud services and give organisations comprehensive control over their intellectual property and development processes (Ilag, et al., 2024).

1.3.2 Primary Objectives

The core objectives of Janus are:

- **Intuitive, Cross-Platform CLI:**
  Develop a user-friendly CLI for local repository management that operates seamlessly on Windows, macOS, and Linux (Majrashi, et al., 2020).
- **Dockerised Web Application:**

  Create a secure, containerised web interface for remote repository management. This ensures that sensitive data remains within a controlled on-premise environment (Bojović, 2024).
- **Plugin Architecture:**

  Implement a flexible plugin framework that supports future expansion and the integration of custom functionalities to meet evolving enterprise needs (Bhattacharya, 2018).
- **LSEP Compliance:**

  Ensure the system meets Legal, Social, Ethical, and Professional (LSEP) standards, incorporating best practices in data protection, regulatory compliance, and internal governance requirements (Akinsola, 2025).

1.4 Structure of the Report

The report is structured to provide a systematic and comprehensive analysis of the Janus project, from its conceptual foundation to its technical implementation and evaluation. Each section builds upon the previous to ensure clarity and coherence:

- **Section 1 (Introduction)** establishes the project's context, problem statement, and objectives. It outlines the necessity of Janus in addressing enterprise security and compliance gaps in existing DVCS solutions.
- **Section 2 (Context & Literature Review)** critically evaluates the evolution of version control systems, identifies gaps in current tools, and aligns Janus with industry requirements. This section grounds the project in academic and industry research.
- **Section 3 (Project Scope & Deliverable)** defines the functional and non-functional requirements of Janus, clarifying boundaries between in-scope deliverables and future enhancements.
- **Section 4 (LSEP Issues)** addresses legal, social, ethical, and professional considerations, demonstrating compliance with regulations and ethical standards.
- **Section 5 (Methodology)** explains the Agile Scrum framework adopted for development, emphasizing iterative delivery and risk management.
- **Section 6 (Design)** details system architecture, UML diagrams, and prototypes to visualize the technical implementation.
- **Section 7 (Development)** documents sprint outcomes, tools, and technologies used, illustrating progress against the project plan.
- **Section 8 (Testing Strategy)** validates the system against requirements through unit, integration, system, user and performance testing.
- **Sections 9–11** (Evaluation, Reflections, Conclusions) assess the project's success, reflect on lessons learned, and propose future work.

2 Context & Literature Review

This section follows the evolution of version control systems, compares centralised and distributed approaches, and evaluates leading Distributed Version Control Systems (DVCS) features, identifying the gaps that Janus will address.

2.1 Evolution of Version Control Systems

2.1.1 Early File-Based Systems: SCCS vs RCS

The Source Code Control System (SCCS), developed at Bell Labs in 1972, was the first widespread VCS. It stores an interleaved delta for each file, which stores each revision directly into each file so that any version can be reconstructed in one pass.

While this offers fast checkouts for any version, it comes at the cost of increased storage and write operations.

By 1982, Walter Tichy's Revision Control System (RCS) introduced reverse deltas, which store the full text of the latest revision and the changes needed to roll back to earlier versions. This dramatically increases the speed at which the most recent changes are reverted at the expense of slower access to older versions. This trade-off between the speed of its most common use case and storage efficiency influenced almost all later VCS design decisions (Koç & Tansel, 2011).

2.1.2 Centralized, Project-Level Control: CVS & Subversion

In the late 1980s, the Concurrent Versions System (CVS) moved from per-file to project-wide tracking, using a client-server model to group multiple files under a single repository. However, CVS still applied commits on a file-by-file basis, meaning a multi-file commit could leave the repository in an inconsistent state if only partially completed (Marjanovic, 2006).

Apache Subversion (SVN), made in 2000, solved these issues by introducing atomic commits, versioned directories, and built-in rename tracking, making branching and merging more reliable. The continued reliance on a central server creates a single point of failure, forces constant network connectivity, and can introduce bottlenecks for large teams (Nikander, 2024).

2.1.3 Distributed Systems: BitKeeper's Legacy & the Git/Mercurial Split

The proprietary BitKeeper system was the first to demonstrate the benefits of full history replication, enabling offline commits and fast local branching (Högblom & Green, 2013). Following licensing disputes, two open-source DVCS projects emerged in 2005:

Git, created by Linus Torvalds, stores commit, trees, and blobs in a content-addressable Merkle DAG, named and identified by a SHA-1 hash. This guarantees cryptographic integrity, high-speed local operations and lightweight branching (Kuhn, 2010).

Mercurial, developed by Olivia Mackall, prioritised a consistent, user-friendly command set, Python-based extensibility, and a smoother learning curve. It sacrificed some of Git's low-level optimisations for better maintainability and enterprise-friendly customisation (Hibbs, et al., 2009).

Both tools cemented the distributed paradigm, their differing priorities, gits performance focus and mercurial usability focus, illustrating how human-friendly workflows, not just technical abilities, have driven the advances in version control systems.

2.2 Comparison Between Centralised & Distributed Systems

Centralised and distributed VCS each have strengths and weaknesses. A comparison of key characteristics is provided in Table 2.1 (Marjanovic, 2006).

Table 2.1: Architectural Comparison of Centralised and Distributed VCS

Characteristic	Centralised VCS	Distributed VCS
Repository Location	Single server	Full local copy per user
Offline Work	Requires network	Fully functional offline operations
Branching & Merging	Heavyweight, server-dependent	Lightweight, local and fast
Fault Tolerance	Single point of failure	Redundancy via multiple replicas
Performance	Network latency, server load	Fast local operations
Access Control	Centralised policies, often coarse-grained	Can implement fine-grained, per-push controls
Scalability	Server bottlenecks at scale	Scales naturally with replicas
Extensibility	Varies; some support hooks/plugins	Rich plugin ecosystems (e.g., Git hooks)

2.3 Feature Comparison of Leading DVCS Tools

A visual comparison of popular DVCS platforms illustrates their strengths and weaknesses in meeting enterprise requirements. Table 2.2 below compares key functional features of Git, Mercurial and Bazaar (Knittl-Frank, 2010).

Table 2.2: Feature Comparison of Git, Mercurial, and Bazaar

Feature	Git	Mercurial	Bazaar
Offline Capabilities	✓	✓	✓
Branching & Merging	✓	✓	✓
Staging Area	✓	✗	✗
Empty Directory Tracking	✗	✗	✓
History Rewriting	✓	✗	✗
Atomic Directory Commits	✓	✓	✗
Ease of Use	✗	✓	✓

Plugin Extensibility	✓	✓	✓
Large File Handling	Plugins needed	Plugins needed	Plugins needed
Access Control	Plugins needed	Plugins needed	Plugins needed
CI/CD Pipeline Integration	✓	✓	✓

2.4 Gaps in Existing Solutions

While modern DVCS tools like Git and Mercurial have transformed software development, they fall short in addressing critical enterprise needs:

- **External Hosting Risks:**
  Public platforms, such as GitHub and GitLab, require data to be stored on third-party servers, exposing organisations to data sovereignty violations and enterprises cannot risk external data exposure (European Union, 2016).
- **Coarse-Grained Access Control:**
  Existing systems rely on plugins for permission management, which lack native support for role-based access control (RBAC). This forces enterprises to develop custom middleware, increasing complexity and maintenance costs (Microsoft, 2024).
- **Integration Overhead:**
  Adapting public DVCS tools to corporate CI/CD pipelines often requires manual configuration of hooks and APIs. For example, integrating Git with internal audit systems demands significant scripting effort, delaying deployment (Ali, 2022).
- **Scalability & Logging Limitations:**
  Centralized logging in cloud-hosted solutions creates bottlenecks during high-volume operations. Enterprises report challenges synchronizing audit trails across distributed teams, risking compliance failures during audits (ISMS, 2020).
- **Limited Extensibility:**
  While plugins exist for features like large file storage (e.g., Git LFS), they are often platform-specific and lack standardization. This complicates cross-tool interoperability and limits customisation (Alnafessah, et al., 2021).

Janus addresses these gaps by prioritizing on-premise deployment, native RBAC, and a modular plugin architecture, reducing reliance on external services and manual integrations.

2.5 Industry Requirements & Relevance

When selecting a distributed version control system (DVCS), modern enterprises face a complex landscape of regulatory, security, performance, and adaptability demands. Janus's architecture is designed to address these requirements, aligning with the critical needs outlined below.

2.5.1 Regulatory Compliance & Data Sovereignty

Enterprises must adhere to stringent data protection regulations such as GDPR, the UK Data Protection Act, and industry-specific frameworks, such as MiFID II for finance. These laws instruct that source code, audit logs, and metadata remain within approved network boundaries to be compliant. Immutable audit trails are equally important, as organizations must provide tamper-evident records of all repository operations, including commits, permission changes, and access attempts, to demonstrate accountability during audits (Sarioguz, et al., 2024).

Janus responds to these requirements by prioritizing on-premise deployment, ensuring data never leaves the internal infrastructure. Its built-in audit logging captures every action, enabling enterprises to satisfy regulations and avoid issues regarding third-party data exposure.

2.5.2 Security & Granular Access Control

Modern enterprises require precise control over repository access to mitigate insider threats and enforce zero-trust security models. Role-based access control (RBAC) is essential, allowing permissions to be tailored to specific teams or workflows.

Janus addresses these needs through native RBAC, enabling administrators to define per-repository permissions without relying on third-party plugins. Revokable Personal Access Tokens (PATs) with configurable expiration dates enforce least-privilege principles, while real-time policy enforcement prevents unauthorized data access.

2.5.3 Performance & Scalability

Modern enterprises require rapid version control operations to sustain developer productivity, particularly in environments with large codebases or distributed teams. CLI actions, such as merging branches, must execute quickly, even for large repositories. Simultaneously, backend infrastructure must scale efficiently to support high concurrency, ensuring that workflows remain uninterrupted during peak usage (Kansal & Balasubramaniam, 2024).

Janus addresses these demands through a lightweight CLI designed for speed, prioritizing fast local operations even with large repositories. Its containerized backend, deployed via Docker, provides a foundation for dynamic scaling. While the current implementation focuses on Docker-based orchestration, the architecture is designed to support future integration with tools like Kubernetes, enabling enterprises to handle horizontally scaling the backend services as needed. This approach ensures consistent performance under load while maintaining compatibility with evolving infrastructure requirements.

2.5.4 Extensibility & Integration

Enterprises rely on many tools, including CI/CD pipelines and compliance checkers. A modern DVCS must offer standardized plugins to embed these tools directly into the version control workflow, reducing manual configuration.

Janus meets this demand through a plugin architecture, enabling organizations to extend Janus functionality. This extensibility ensures that the system can evolve alongside emerging tools and standards.

2.5.5 Alignment with Enterprise Priorities

Janus directly resolves the limitations of public DVCS platforms by combining four core pillars: on-premise deployment for data sovereignty, native RBAC for precise security, optimized performance for scalability, and Plugins for extensibility. By addressing regulatory mandates, mitigating security risks, and supporting many different workflows, Janus allows enterprises to future-proof their version control infrastructure while balancing compliance and innovation.

3 Project Scope & Deliverable

This section outlines the Janus project's deliverables, detailing the components that comprise the final version while clearly stating what is out of scope for the project's development.

3.1 Main Components

The Janus project consists of several core components:

Dockerised Frontend, Backends & Database:

The project will deploy a containerised environment using Docker, which is to be hosted on-premise, including:

- Frontend: A responsive, user-friendly web interface providing secure access to repository data and management tools (De la Torre, 2016).
- Backend: Endpoints designed to support CLI and web interactions, managing authentication, audit logging and data processing (Gowda & Gowda, 2024).
- Database: A solution for storing user and remote repository data (U.S. Cybersecurity and Infrastructure Security Agency (CISA), 2023).

Local CLI:

The Command Line Interface (CLI) is the core of Janus's local operations and will support essential functionalities such as:

- Repository initialisations (Creating a hidden directory containing the local repository for version control)
- File staging and commit history loggings
- Branch management, merging and conflict detection
- Pushing changes to a secure, internal remote repository (Singh, et al., 2023).

Plugin Framework:

The CLI will be developed to allow users to extend Janus's core functionality through plugins. This plugin system enables custom features to be integrated easily into the Janus system to adapt to varying enterprise needs. (Rellermeyer, 2011).

Documentation:

Comprehensive guides covering installation and usage of the CLI and Dockerised system will be provided to ensure that both technical and non-technical users can effectively use and maintain the system (Khalid, et al., 2025) along with documentation on how plugins can be created and used with the CLI.

3.2 Requirements

Functional requirements detail what the Janus system should do. They encompass the DVCS operations, user interactions and systems behaviours necessary to meet the projects goals.

3.2.1 Functional Requirements

Table 3.1: Janus Functional Requirements

ID	Requirement	Priority	Acceptance Criteria
FR-01	Repository Initialisation: CLI shall initialise a new local repository by creating a hidden .janus/ directory.	Must	janus init creates a .janus/ folder containing metadata files (objects, commits, branches).
FR-02	Commit Changes: CLI shall stage files and commit snapshots with author, timestamp, and message.	Must	janus add <file> marks files as staged; janus commit "msg" creates commit object and updates the branch HEAD.
FR-03	Branch Management: CLI shall create, list and switch branches.	Must	janus branch create <name> adds branch; janus list_branch shows all branches; janus switch_branch <name> switches current branch.

FR-04	Merge & Conflict Detection: CLI shall merge two branches and mark conflicts for user resolution.	Must	janus merge <branch> applies non-conflicting changes and displays conflict areas to users.
FR-05	Push to Remote: CLI shall push commits to Docker-hosted remote using HTTPS.	Must	janus push returns HTTP 200 and remote HEAD advances to the pushed commit hash.
FR-06	Fetch from Remote: CLI shall fetch new objects from remote without modifying working tree.	Must	janus fetch populates objects and commits from the remote and updates remote refs.
FR-07	Pull from Local Repository: CLI shall pull new commits from local repository to update working directory.	Must	janus pull merges or fast-forwards local repository to the working directory.
FR-08	Clone Repository: CLI shall clone a remote repository, creating a working copy of the remote.	Must	janus clone <link> creates working copy from remotes and sets up origin remote pointing to the link.
FR-09	Remote Status: CLI shall report ahead/behind counts relative to the remote.	Should	janus status displays “ahead/behind” when local and remote differ.
FR-10	Revert Command: CLI shall revert the working directory to a previous commit and record a new “revert” commit.	Should	janus revert <hash> resets working tree to <hash> state and updates head to a new commit whose tree is the same.
FR-11	User Authentication Web: Web application shall require login via email and password.	Must	Invalid credentials return 401; valid credentials establish a session and redirect to dashboard.
FR-12	PAT Generation/Revoke: Web application shall let users create PATs with expiry and revoke them at will.	Must	UI form issues a token with the chosen expiry; revoked tokens fail on any API request.
FR-13	PAT Authentication: CLI shall use email and PAT for all authenticated API calls.	Must	CLI commands requiring auth (push, pull, fetch) succeed with valid PAT; fail with expired/revoked PAT.
FR-14	Repository Management: Web UI shall allow create, view, rename, and delete remote repositories.	Must	Web UI actions on dashboard and repository settings update database states.
FR-15	Access Control: System shall enforce per-repository permissions (read/write/admin/owner).	Must	Users without write permission get invalid response on push; without read get invalid response on clone/fetch; owners & admins can grant/revoke roles.
FR-16	Commit History: Web application shall display a paginated list of	Must	Commit list loads in under 1s.

	commits, with author, date, and message.		
FR-17	File Tree Browsing: Web application shall display repository file tree for all branches.	Must	Users can expand folders; clicking file shows its contents.
FR-18	Repo Visibility & Description: Web application shall allow marking repos as public/private and set a description.	Should	Toggling visibility and editing description save database state.
FR-19	README Rendering: Web application shall detect and render README.md in markdown on the repo home page.	Could	If README.md exists in root, its rendered on the repo page.

Each functional requirement has an associated priority, following the MoSCoW method (Hudaib, et al., 2018) and acceptance criteria that clarify what it takes to meet the requirement.

3.2.2 Non-Functional Requirements

Non-functional requirements specify the criteria for the system rather than specific behaviours; these include performance benchmarks and security standards.

Table 3.2: Janus Non-Functional Requirements

ID	Category	Requirement	Metric / Target	Verification
NFR-01	Performance	CLI operations (commit, branch, clone, fetch ≤200ms for responsories with many files).	≤200ms per operation	CLI performance testing
NFR-02	-	Web UI main page loads ≤1s under normal load.	≤1s	Frontend performance testing
NFR-03	Scalability	Server supports ≥100 concurrent users without >1s response times.	≤1s under 100 users	Load testing
NFR-04	Security	All CLI server traffic uses HTTPS (TLS 1.2+).	100% encrypted; no insecure endpoints	System review
NFR-05	-	Passwords stored with PBDF2 SHA-256 (≥ 600 000 iterations) and salted with ≥128-bits.	OWASP password storage compliant	System review

NFR-06	Reliability	Database operations are ACID-compliant.	ACID-compliant	System review & database testing
NFR-07	Usability	CLI provides concise help (janus help <cmd>) and clear error messages.	Command descriptions & usage	CLI testing
NFR-08		Web UI meets WCAG 2.1 AA accessibility guidelines.	WCAG 2.1 AA conformant	Accessibility testing
NFR-09	Maintainability	Code follows DRY & SOLID principles.	Testing covers most of the system	System testing
NFR-10	Portability	CLI runs on Windows, Linux & macOS.	Verified on each OS	Cross-Platform testing
NFR-11	Compliance	System complies with GDPR & Data Protection Act (Data remains on-premise).	Full on-premise hosting	System review
NFR-12	Documentation	User and developer guides complete and stored on GitHub repository	Full feature coverage	Documentation review
NFR-13	Extensibility	Plugin setup allows integration of new custom commands.	New plugins load dynamically	Plugin testing

3.3 Out-of-Scope

For the delivered release, Janus will focus on core enterprise-grade DVCS functionalities and secure internal deployment. Potential future advancements such as advanced customisable user interfaces beyond the standard web dashboard, enhanced encryption protocols for data at rest and more advanced security features, development of a community platform to facilitate sharing and collaboration on custom plugins, advanced history modification and tamper-evident auditing are considered out of scope for the project.

4 Legal, Social, Ethical, & Professional (LSEP) Issues

Significant attention was given to ensuring that legal, social, ethical, and professional issues were addressed both in and before the development of Janus. This section evaluates the measures to manage these concerns and discusses their implications for system design and accountability.

4.1 Legal Considerations

4.1.1 Data Protection & Privacy Compliance

Janus is designed to operate entirely on-premise, ensuring that sensitive code and personal data remains within the organisation's control. By leveraging Docker to deploy the system within a controlled subnet, the system minimises risks associated with external data exposure; this approach supports compliance with data protection regulations such as GDPR (European Union, 2016) and the Data Protection Act (UK Government, 2018).

Additionally, enforcing HTTPS for all data transfers provides an essential layer of encryption that safeguards data in transit and mitigates the risk of interception (OWASP, 2025).

4.1.2 Secure Authentication & Account Management

Robust security is implemented using JSON Web Tokens (JWT) for API authentication, enabling secure communications between system components; this method is widely recognised for its efficiency in distributed environments (Jones, et al., 2015). The system utilises Personal Access Tokens (PAT) that are revocable and have configurable expiry times, reducing the reliance on static passwords and enhancing session security (National Institute of Standards and Technology, 2017).

Furthermore, passwords are salted and hashed using industry-standard cryptographic practices (600,000 iterations of PBKDF2 with SHA256), ensuring resilience against brute force and dictionary attacks (OWASP, 2025). The use of 128-bit salt is balanced between collision risk and performance (NIST, 2017). These measures ensure that stored credentials are robust and that data integrity is maintained through transactional data interactions (Oracle, 2025).

4.1.3 Licensing & Intellectual Property

Janus integrates various third-party libraries and frameworks, all of which have been reviewed for licensing compatibility; this minimises legal risks related to open-source or proprietary components (OSI, 2024).

In addition, Janus has plugin functionality that allows users to develop custom commands designed for user customisation of the system. As a result, the user or organisation will hold the intellectual property of the custom-developed plugins (Svitla, 2024).

4.1.4 Audit & Accountability

An essential component of Janus is its comprehensive audit logging, which records all database interactions to create a solid audit trail. This supports internal audits and serves as legal evidence in cases of data breaches or non-compliance; transparency in system operations is maintained because both the old and new states of data are logged, ensuring accountability and regulatory compliance (Souppaya & Kent, 2006).

4.2 Social Considerations

4.2.1 User Trust & Data Sovereignty

A key social advantage of Janus is the enhanced control organisations have over their codebases; by eliminating the need for external cloud services, user trust is improved as all information is managed internally. This approach reinforces data sovereignty and ensures users know who handles their information (Scherenberg, et al., 2024).

4.2.2 Accessibility & Transparency

Janus has been designed with the user experience in mind. The React-based web interface adheres to most modern accessibility standards, such as the WCAG guidelines (WCAG, 2024), ensuring that users from any background can effectively navigate and utilise the system.

Clear documentation is provided, including detailed usage instructions for the CLI. While features like light/dark themes support usability by accommodating user preferences and reducing eye strain (Kristallovich & Eisfeld, 2020).

4.3 Ethical Considerations

4.3.1 Responsible Data Handling

Ethically, Janus prioritises the responsible management of user data. Users must accept the Terms of Use and Privacy Policy before creating an account, ensuring informed consent regarding data handling.

Sensitive data remains confined within the organisation, minimising the risk of unauthorised exposure; this approach protects individual privacy rights and upholds ethical standards in data management (Chang, et al., 2016).

4.3.2 Automated systems

Janus deliberately avoids automated resolutions, such as automated merge conflict handling, ensuring that users fully hold control over critical actions: this places accountability with the users and reduces the risk of compromising data integrity.

4.3.3 Transparency in Operations

The detailed audit logging mechanism, combined with explicit Terms of Use and Privacy Policy, ensures that users are well informed about data collection and processing practices; transparency is essential for ethical accountability and enabling users to make informed decisions about their data (Information Commissioner's Office, 2025).

4.4 Professional Considerations

4.4.1 Adherence to Industry Standards

From a professional standpoint, Janus adheres to established industry best practices in software development. The implementation of design principles such as DRY (Don't Repeat Yourself) (Thomas & Hunt, 2000) and the use of modular, reusable code components contribute to maintainability and scalability (Parnas, 1972). The selection of industry-standard frameworks, such as .NET Core and React, ensures that the system is robust and that professional standards are upheld (Anjum & Alam, 2019).

4.4.2 Quality Assurance & Continuous Improvement

Professional responsibility is demonstrated through rigorous testing and continuous integration/continuous deployment (CI/CD) pipelines. Regular unit, integration, system and usability tests ensure that Janus maintains high standards of quality and reliability (Bhanushali, 2023). Moreover, the agile development methodology and sprint planning ensure that professional standards are maintained throughout the development lifecycle (Dybå & Dingsøyr, 2008).

4.4.3 Documentation

Comprehensive documentation is essential for maintaining professionalism. Janus provides detailed documentation for the CLI, along with user guides and technical documents (see Appendix B – CLI Documentation and Appendix C – Plugin Developer Guide), ensuring that users and developers can understand and effectively utilise the system (Gao, et al., 2003). This clear documentation demonstrates the project's commitment to professional clarity and accountability.

4.4.4 Risk Management & Incident Response

Finally, Janus incorporates a robust risk mitigation strategy that addresses potential issues ranging from feature creep to security vulnerabilities. Revocable PATs, audit logging and continuous testing reflect the approach to managing professional and ethical risks; this approach safeguards the system and aligns with the professional duty to anticipate and mitigate potential threats (Canedo, et al., 2025).

5 Methodology

The Janus project followed the Agile Scrum framework for its iterative and incremental development process. This approach was selected over traditional models like Waterfall due to the project's evolving requirements and the need for continuous feedback.

5.1 Rational for Agile Scrum

Agile Scrum was selected for four primary reasons:

- **Adaptability to Change:**
  As prototype testing progressed, requirements around sections of the project evolved. Scrum's short, iterative sprints allowed for dynamic changes to the backlog, re-prioritising and incorporating new insights without derailing the schedule.
- **Incremental Delivery:**
  Organising work into two-week sprints, integration issues and usability concerns could be detected early on.
- **Continuous Feedback:**
  Regular sprint reviews with supervisors and stakeholders ensured that every step aligned with the project vision. Actionable feedback from these sessions guided subsequent sprint planning, helping to refine design and implementation details on an ongoing basis.
- **Progress Tracking:**
  Utilising sprint backlogs and Kanban boards provided real-time visibility into task status and potential blockers. This transparency made it straightforward to identify and mitigate risks before they could impact the project.

5.2 Sprint Structure & Workflow

Each two-week sprint followed a four-step cycle:

1. **Sprint Planning:**
   Define the sprint goal, select backlog items, and assess any associated risks to adjust priorities accordingly.
2. **Development & Testing:**
   Implement features and perform automated/manual tests.
3. **Sprint Review:**
   Demonstrate functionality to supervisors, modifying the project backlog based on feedback.
4. **Sprint Retrospective:**
   Reflect on what worked well and what could be improved and refine the processes, tools and documentation before the next sprint.

5.3 Backlog & Planning Tools

Monday.com was used to manage scope, risks and tasks by utilising multiple views.

- **Kanban Boards:**
  Visualise task statuses (To Do, In Progress, Done) and quickly identify bottlenecks.

- **Gantt Charts:**
  View the project's high-level progress.
- **Risk Register:**
  Assess each task's likelihood and impact, documenting mitigation strategies.
- **User Stories and Acceptance Criteria:**
  Identify requirements and clear success criteria to guide development and testing.

5.4 Git & GitHub Version Control for Development

Janus development follows a trunk-based Git workflow on GitHub, with all changes committed directly to the main branch to eliminate branch divergence and merge conflicts. Every push triggers the GitHub Actions pipeline, which restores, builds, and tests the CLI on Ubuntu, Windows, and macOS, guaranteeing consistent cross-platform functionality. This streamlined setup minimises administrative overhead, delivers instant CI/CD feedback, and maintains a linear traceable commit history that mirrors the sprint plan tasks.

6 Design

6.1 System Diagrams

Several design diagrams were created to visualise Janus's architecture and key components to develop a clear implementation plan. Each diagram validated design decisions and outlined the system's structure, helping to align the implementation with the project requirements.

6.1.1 System Architecture Diagram

Figure 6.1 illustrates Janus's high-level structure, showing all major components and how they interact. It highlights how these components are separated into client-side and server-side.

This separation of concerns into client, server and database supports scalability, as each part can be scaled or replaced independently. Extensibility is also supported, as new components or services can be integrated without affecting unrelated parts of the system. Alternative architectures were considered, such as using a single monolithic application; however, the chosen client-server model was more suitable for an enterprise DVCS regarding flexibility and maintainability. Deploying the server in isolated Docker containers was decided to simplify maintenance and improve reliability.

In summary, the system architecture diagram guided Janus's modular design and provided a clear understanding of each component.

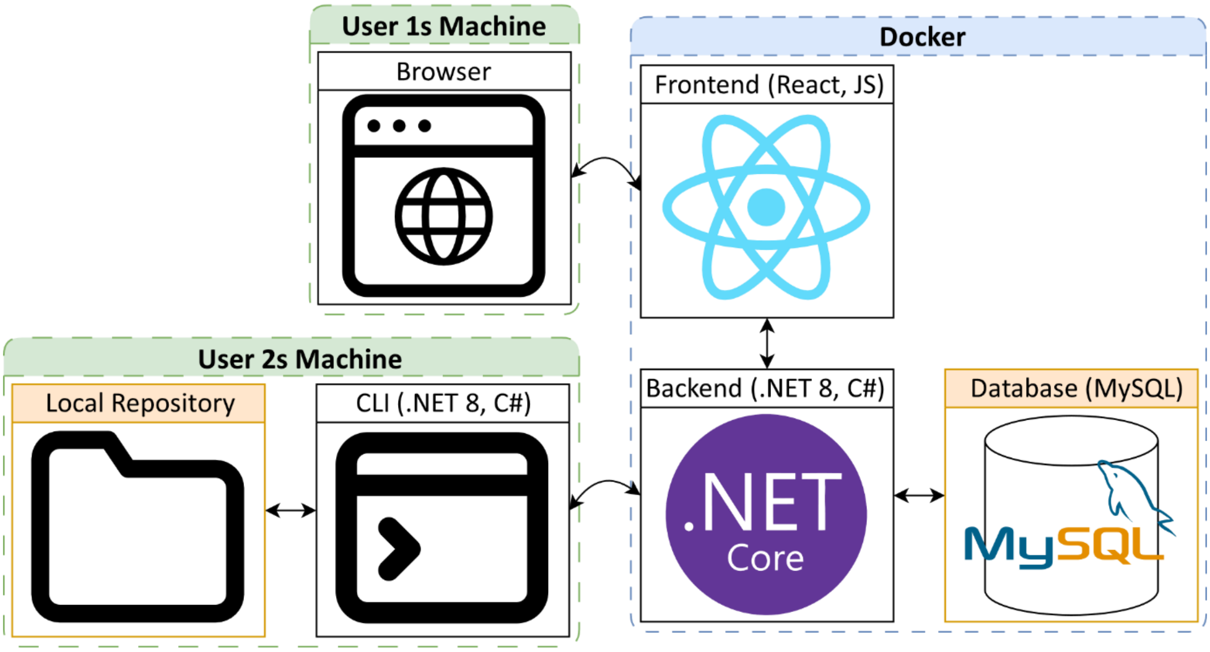


Figure 6.1: System Architecture Diagram

6.1.2 Use Case Diagram

A UML use case diagram was created to display the permissions associated with each role in the Role-Based Access Control system. Figure 6.2 shows the actions available to users based on their assigned role: Read, Write, Admin or Owner.

The roles build upon each other; for example, the Write role inherits all Read permissions. This hierarchical structure is shown using dashed arrows to represent the inheritance between roles. This approach ensures that all permissions are clearly defined and logical.

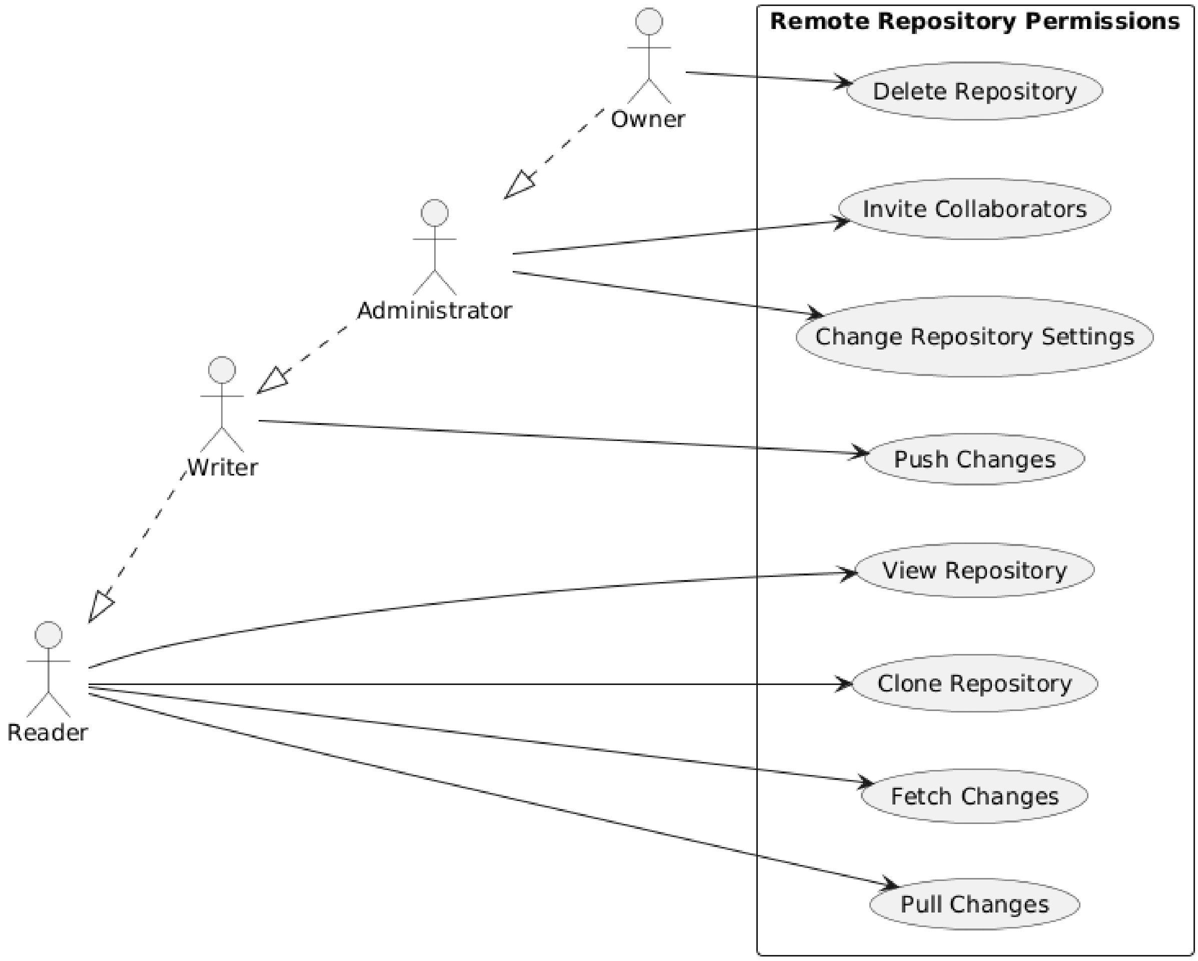


Figure 6.2: Use Case Diagram for Janus Role-Based Access Control

6.1.3 Class Diagram

The UML class diagram Figure 6.3, illustrates the structure of the Janus plugin system, which supports the dynamic loading of external plugin commands. The diagram highlights the interaction between interfaces and the abstract BaseCommand class, showing how they work together to execute commands. Dynamic behaviour is achieved through reflection within the PluginLoader class, allowing the Program to discover load and execute commands at runtime.

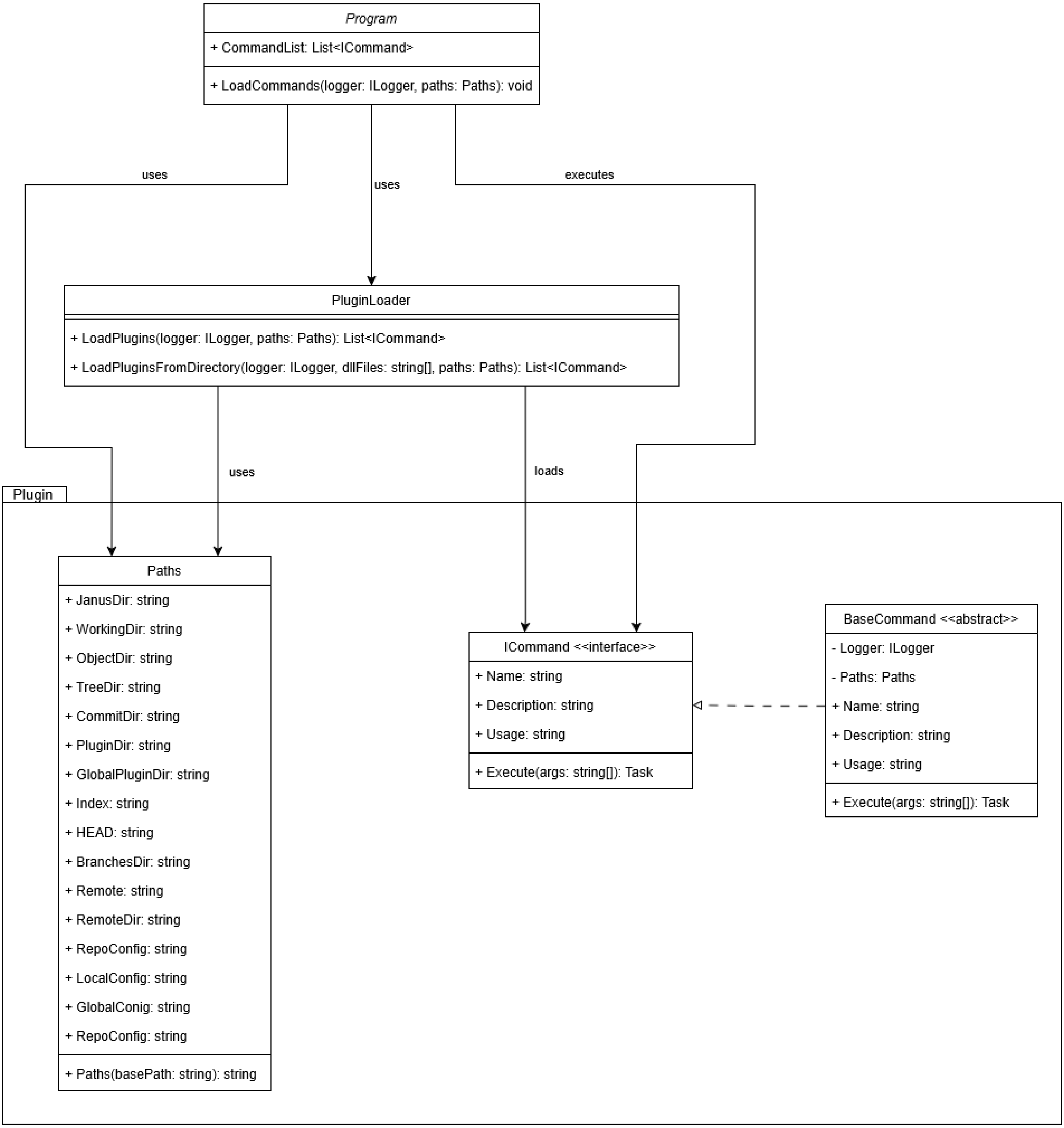
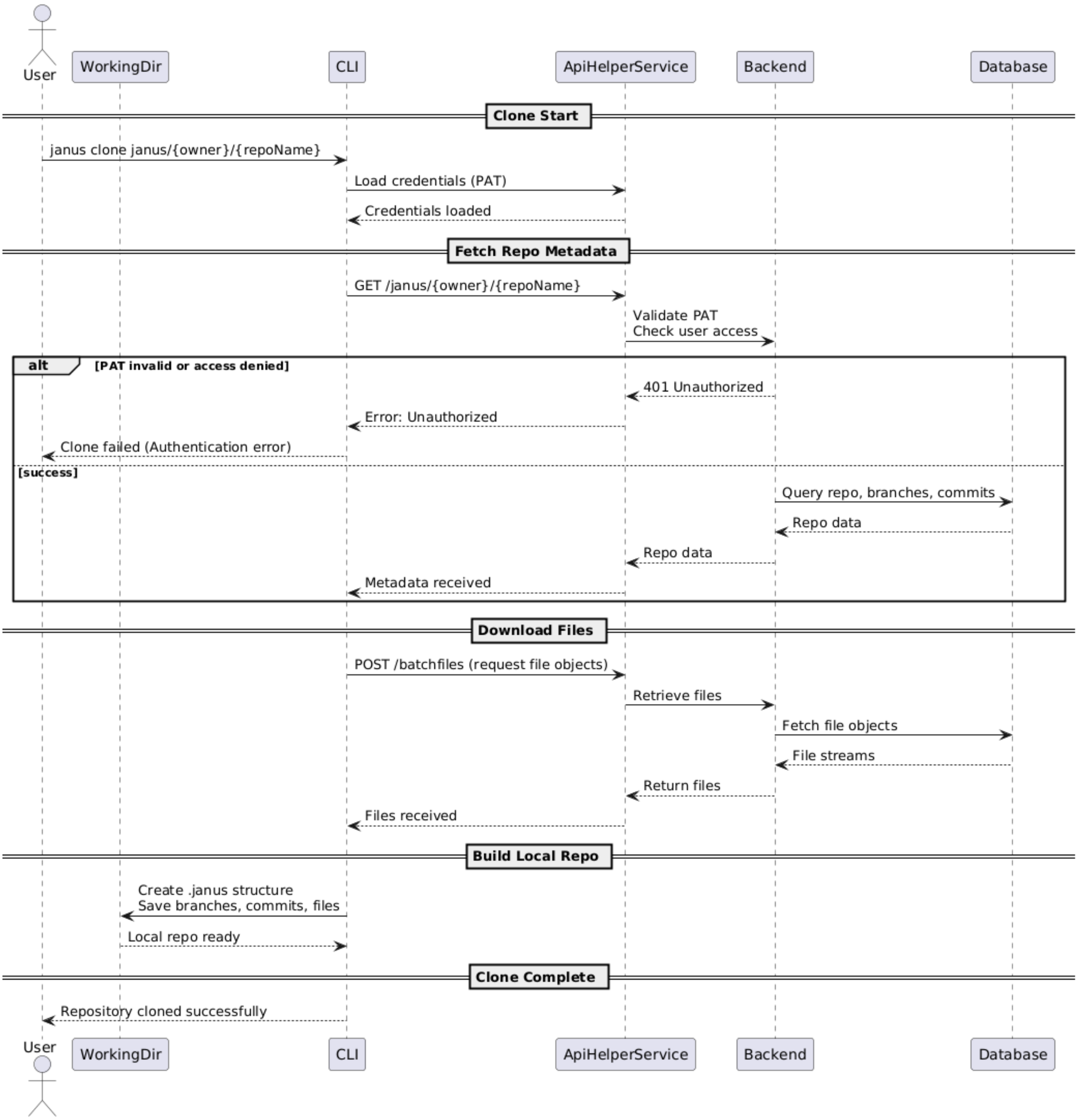


Figure 6.3: UML Class Diagram of the Janus Plugin System

6.1.4 Sequence Diagram

Figure 6.4 is a sequence diagram of the janus clone CLI command, showing how it operates and interacts with key parts of the system. This provides an easy way to visualise how data is handled.

Separating the retrieval of metadata from the downloading of file objects improves reliability, supports scaling and optimises performance for large repositories. Additional sequence diagrams for the Push and Fetch operations can be found in Appendix D.



6.1.5 Entity Relationship diagram

Figure 6.4: Clone Sequence Diagram

A database schema diagram was produced using MySQL Workbench to generate Figure 6.5 automatically. It displays the database tables and the relationships between them. The ERD helped visualise foreign key relationships and identify the best places of indexes to improve speed and balance normalisation with performance.

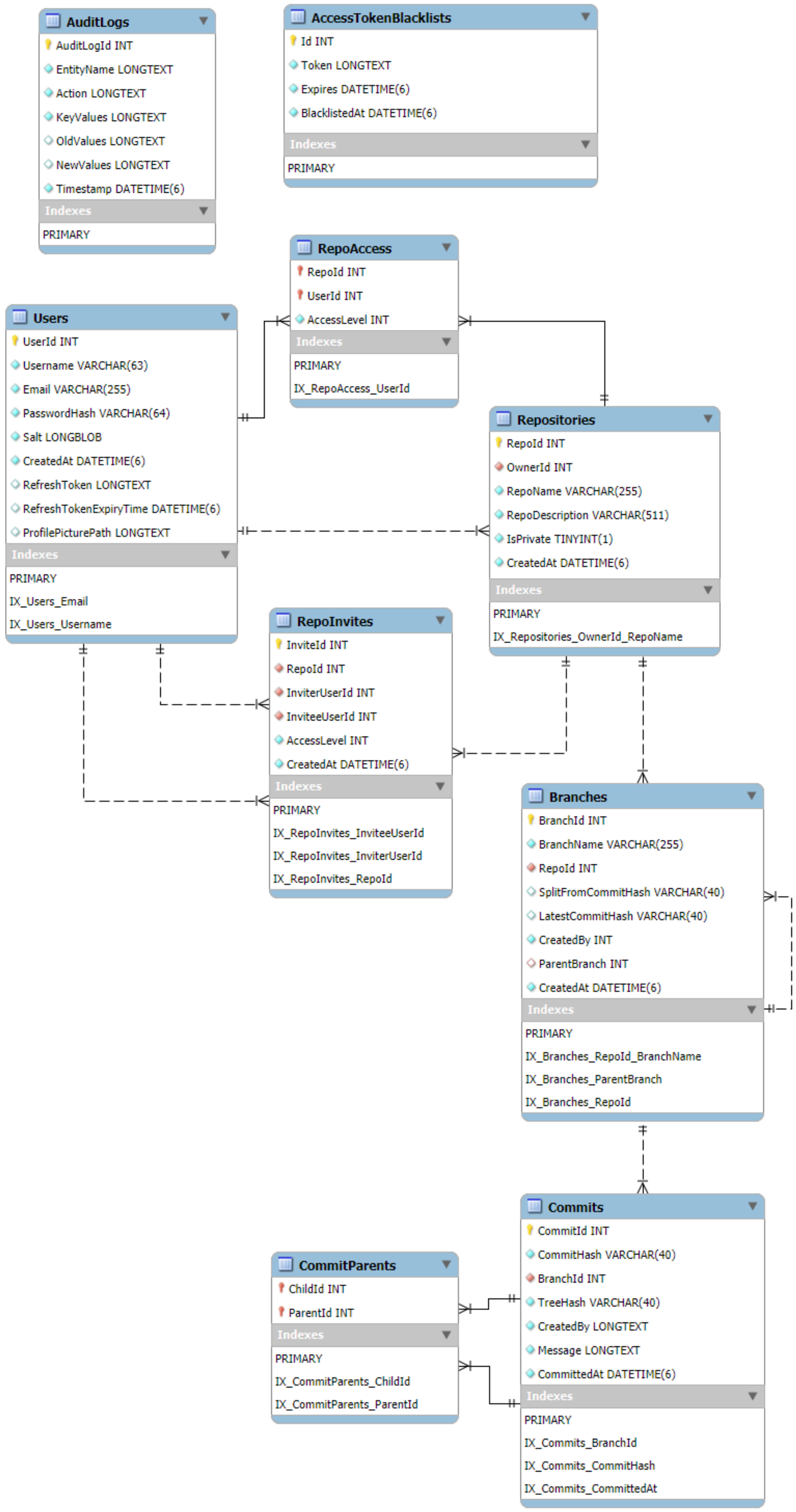


Figure 6.5: Entity Relationship Diagram

6.2 HCI Principles

Throughout the design phase, Human-Computer Interaction (HCI) principles were carefully considered to ensure that Janus is accessible to a wide range of users. The design followed key HCI heuristics outlined by Nielsen (Nielsen, 1995) , such as Consistency and Standards, ensuring similar actions are presented uniformly across different parts of the system; and Recognition rather than Recall, making icons and navigation intuitive and clear labels were used to minimise the information the user has to remember. User Control and Freedom were supported by ensuring users could easily navigate into and out of pages.

Accessibility was a core focus with adherence to WCAG 2.1 AA guidelines (World Wide Web Consortium (W3C), 2018) for visual design, colour contrast and navigability. Designs were tested using colour blindness simulations to ensure inclusivity and accessibility.

Usability testing of low-fidelity and high-fidelity prototypes was conducted to gather user feedback and validate design decisions, following a User-Centred Design methodology (Norman, 2013). By applying these HCI principles to the prototypes and designs, Janus delivers an intuitive and inclusive user experience.

6.3 Colour Scheme

The visual design of Janus’s web interface was carefully considered to ensure a professional look and feel and adherence to accessibility standards. Multiple colour schemes were prototyped and tested during the design phase to gather user feedback. Consistency and clarity were key goals in the decision, as the chosen colours needed to make the interface intuitive, with clear distinctions between elements to result in a good user experience. Several initial colour palettes were created, each with different primary and secondary colours and varying contrast levels, as shown in Figure 6.6. These designs were used during user testing to gather feedback and make adjustments where necessary.

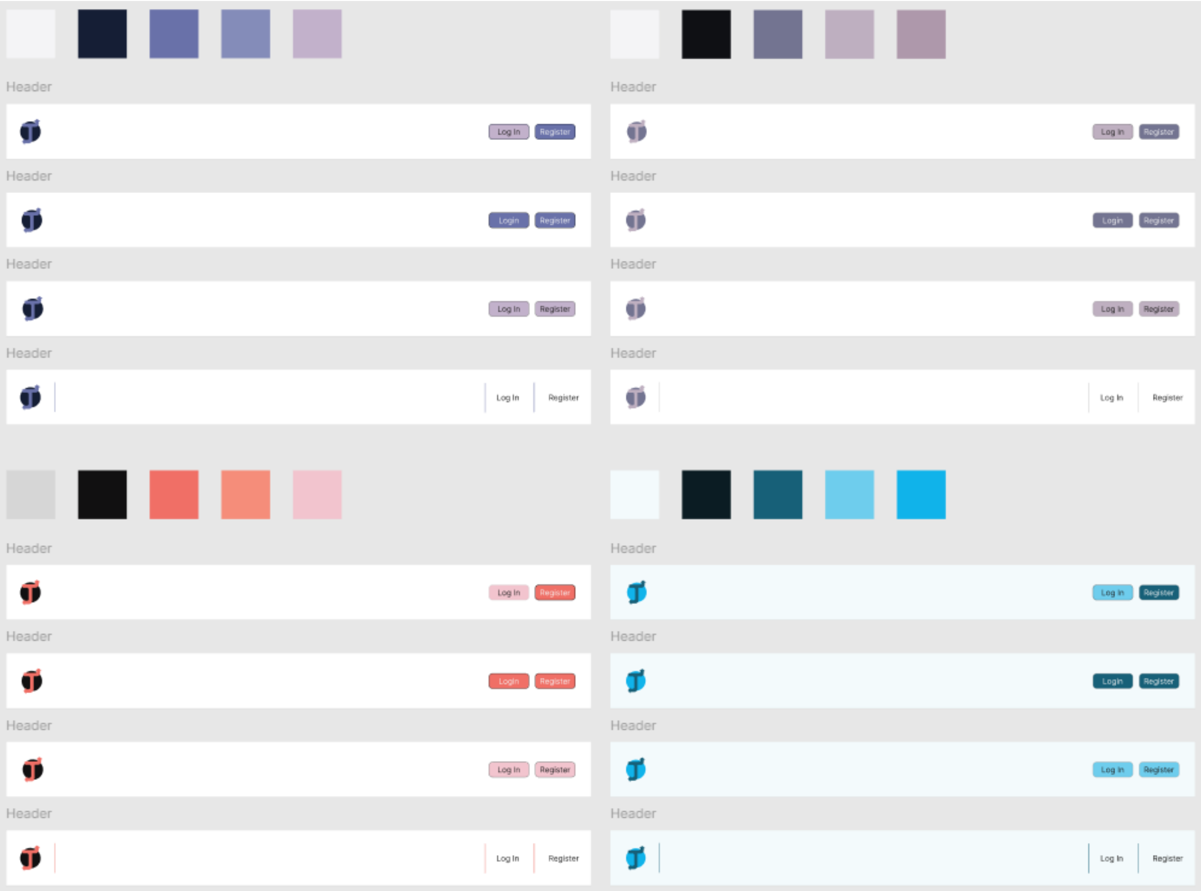


Figure 6.6: Initial Colour Scheme Options for User Feedback

Through this iterative process, balanced light and dark colour themes were selected, shown in Figure 6.7, as feedback showed that many developers preferred darker themes. The final choice was made after incorporating feedback that favoured subtle modern aesthetics over bright, saturated colours.

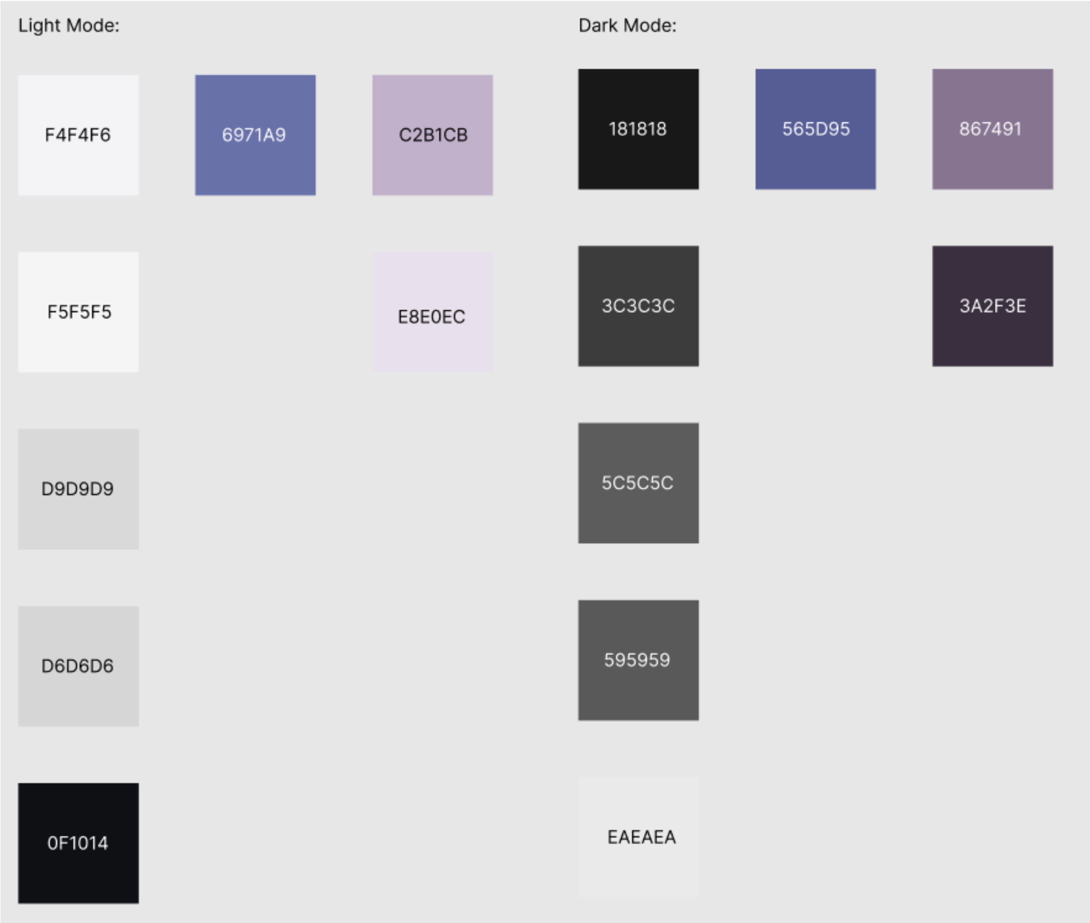


Figure 6.7: Final Chosen Colour Scheme for Light & Dark Themes

The chosen colour scheme was then tested for accessibility against WCAG guidelines (WCAG, 2024), and to ensure usability, the colour palette was examined under colour blindness filters. Figure 6.8 and Figure 6.9 the colour scheme as seen by users with deuteranopia (green deficiency) and protanopia (red deficiency), respectively, and the colours remained clear and distinguishable in both simulations.

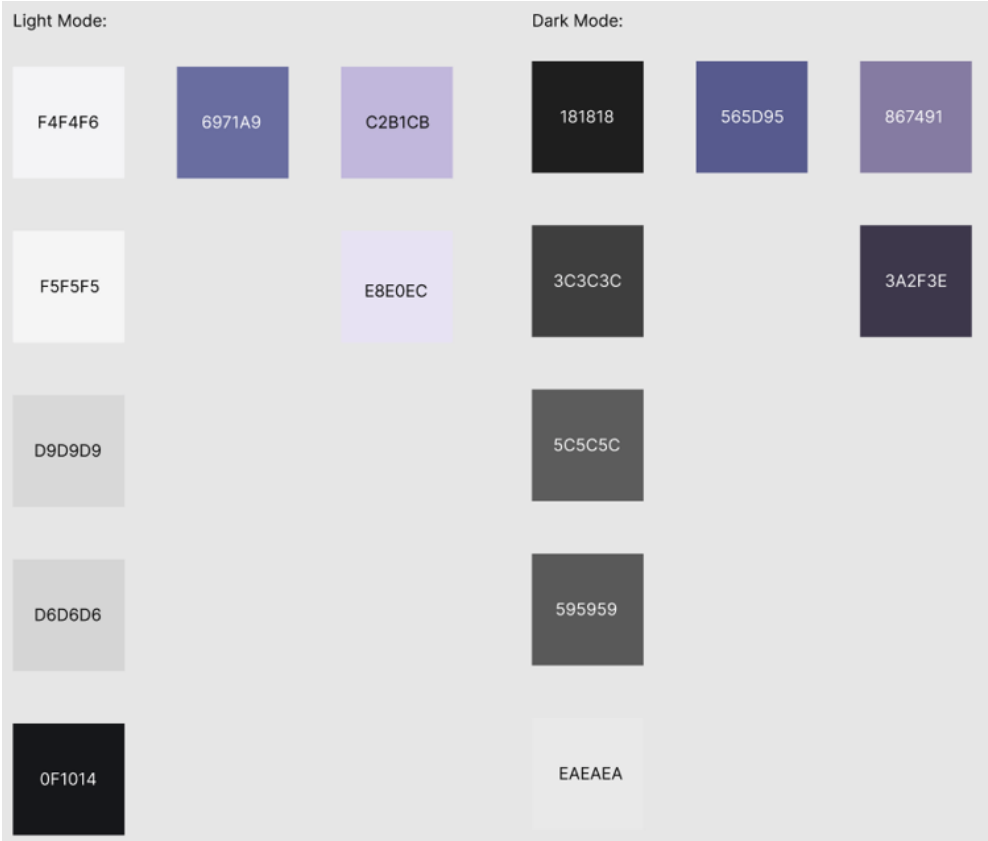


Figure 6.8: Colour Scheme under Deuteranopia Simulation

Figure 6.9: Colour Scheme under Protanopia Simulation

6.4 Logo

Branding is an important design aspect, so a distinctive logo was developed for Janus. The design process for the logo went through multiple interactions, incorporating feedback at each stage. The original concept for the logo was to represent Janus, the Roman deity, with two faces, symbolising looking into the past and future. However, user feedback indicated that a simpler modern design would be preferable. The logo evolved into the form seen in Figure 6.10.

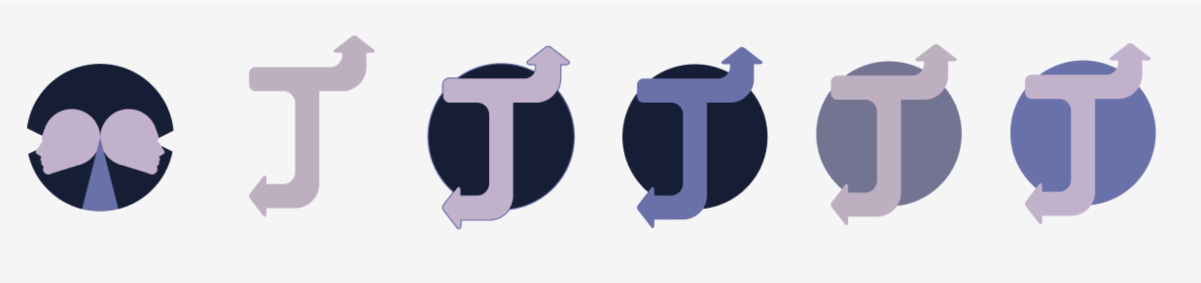


Figure 6.10: Logo Design Iterations

6.5 Prototypes

The design phase included creating both low-fidelity and high-fidelity prototypes of the Janus web application to refine the user interface and experience before full implementation. This prototyping process was crucial for testing assumptions about layout and usability and allowed iterative Improvement of the UI in alignment with user expectations. All prototyping was made using Figma, which enabled quick adjustments and iterations.

6.5.1 Low-Fidelity Prototypes

Low-fidelity prototypes included wireframes illustrating the application’s basic screens without any detailed styling. The emphasis at this stage was on structure and functionality rather than appearance. By feedbacking wireframes, thoughts on navigation and logically grouped elements came up. For example, the repository page was deemed to need some search or filtering to find repositories. This step laid a solid base for the web app structure that supports the user experience.

Figure 6.11: Low-Fidelity Login Wireframe

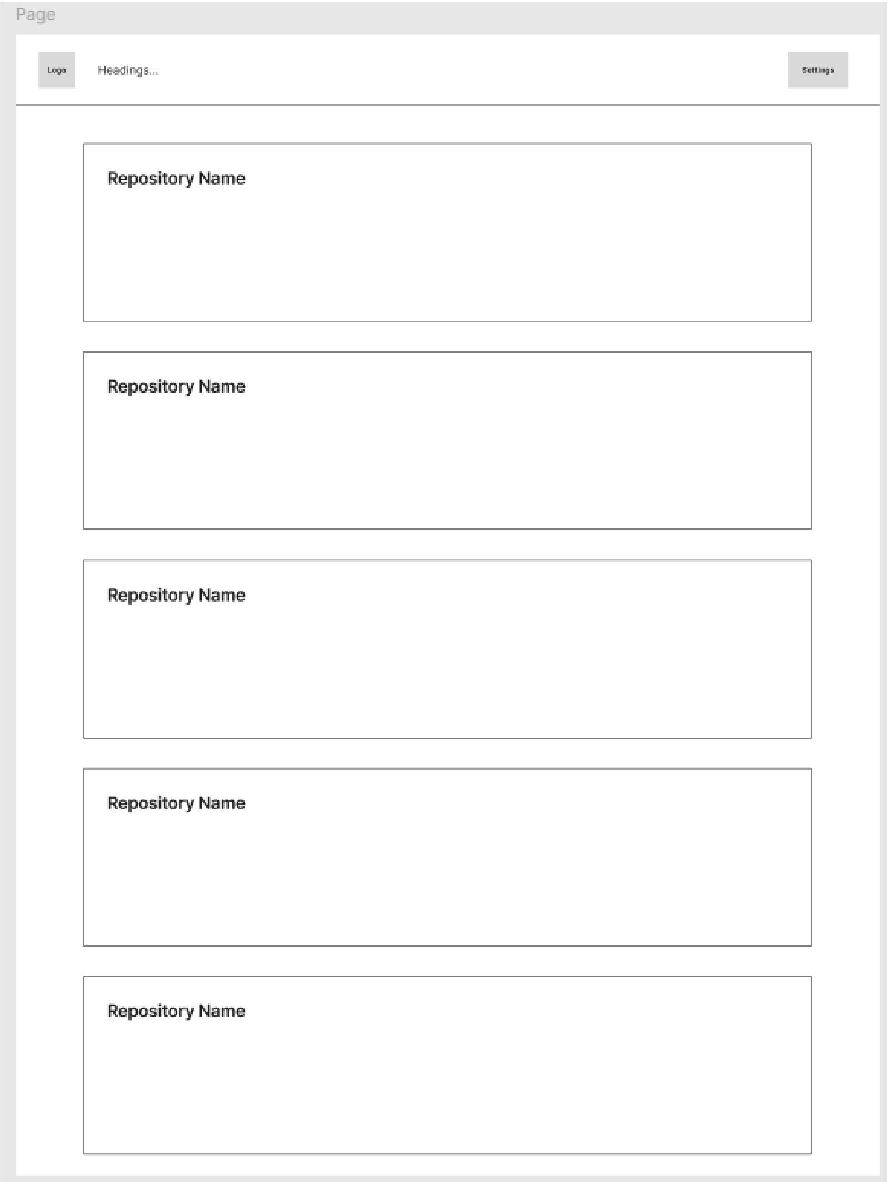


Figure 6.12: Low-Fidelity Repositories Page Wireframe

6.5.2 High-Fidelity Prototypes

High-fidelity prototypes were created to represent the near-final look and feel of the Janus web interface. These more detailed prototypes incorporated the logo, colour scheme, and placeholder data. Multiple designs were created and compared before a final design was chosen. These final prototypes served as instructions on implementing the web application.

While this report cannot include every design, the final designs of key pages and elements are given below. Additional screenshots of the designs and a link to the Figma project are provided in Appendix F.

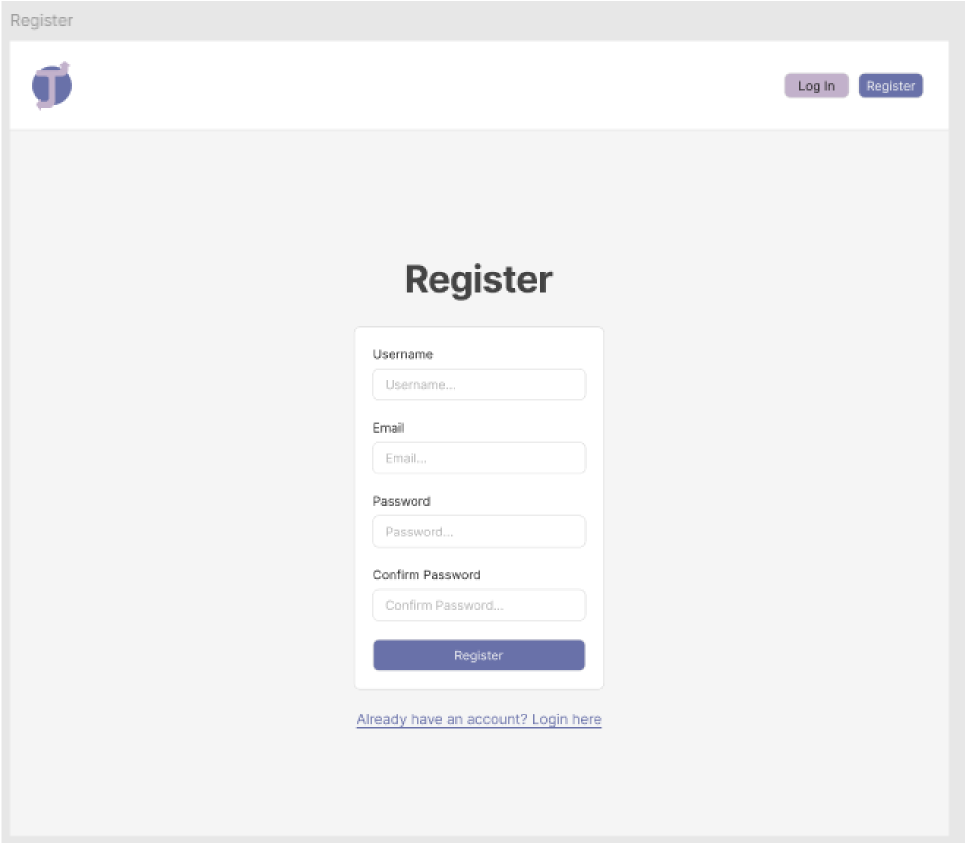


Figure 6.13: High-Fidelity Register Page Prototype

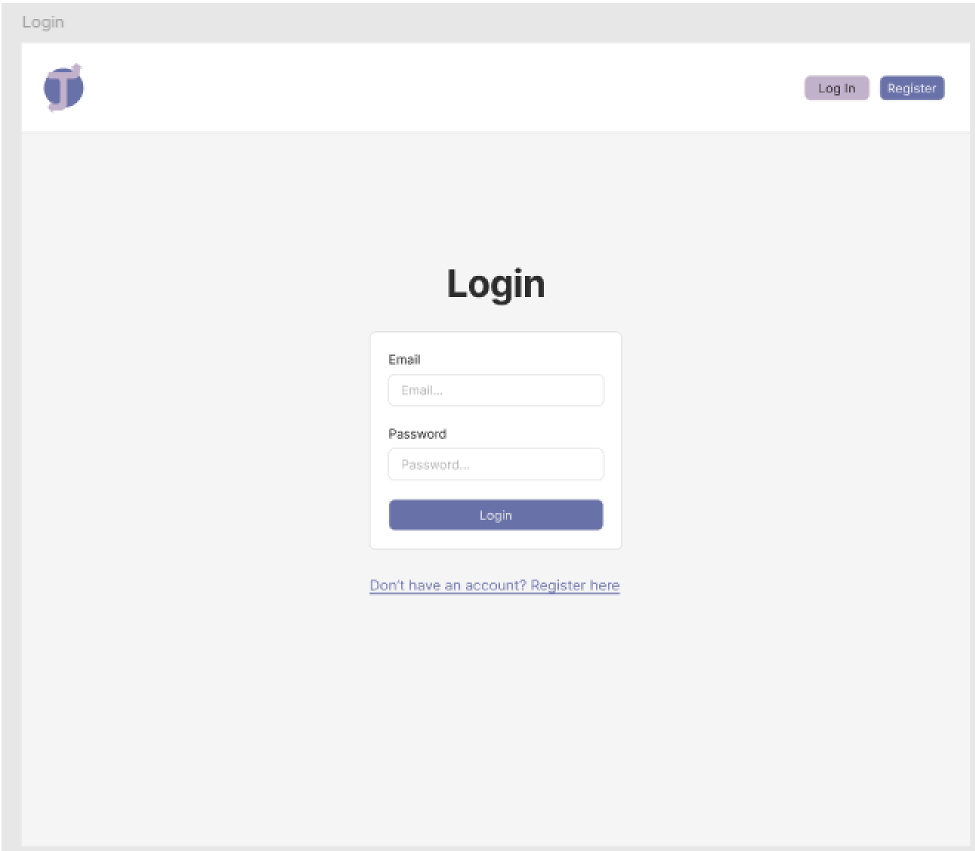


Figure 6.14: High-Fidelity Login Page Prototype

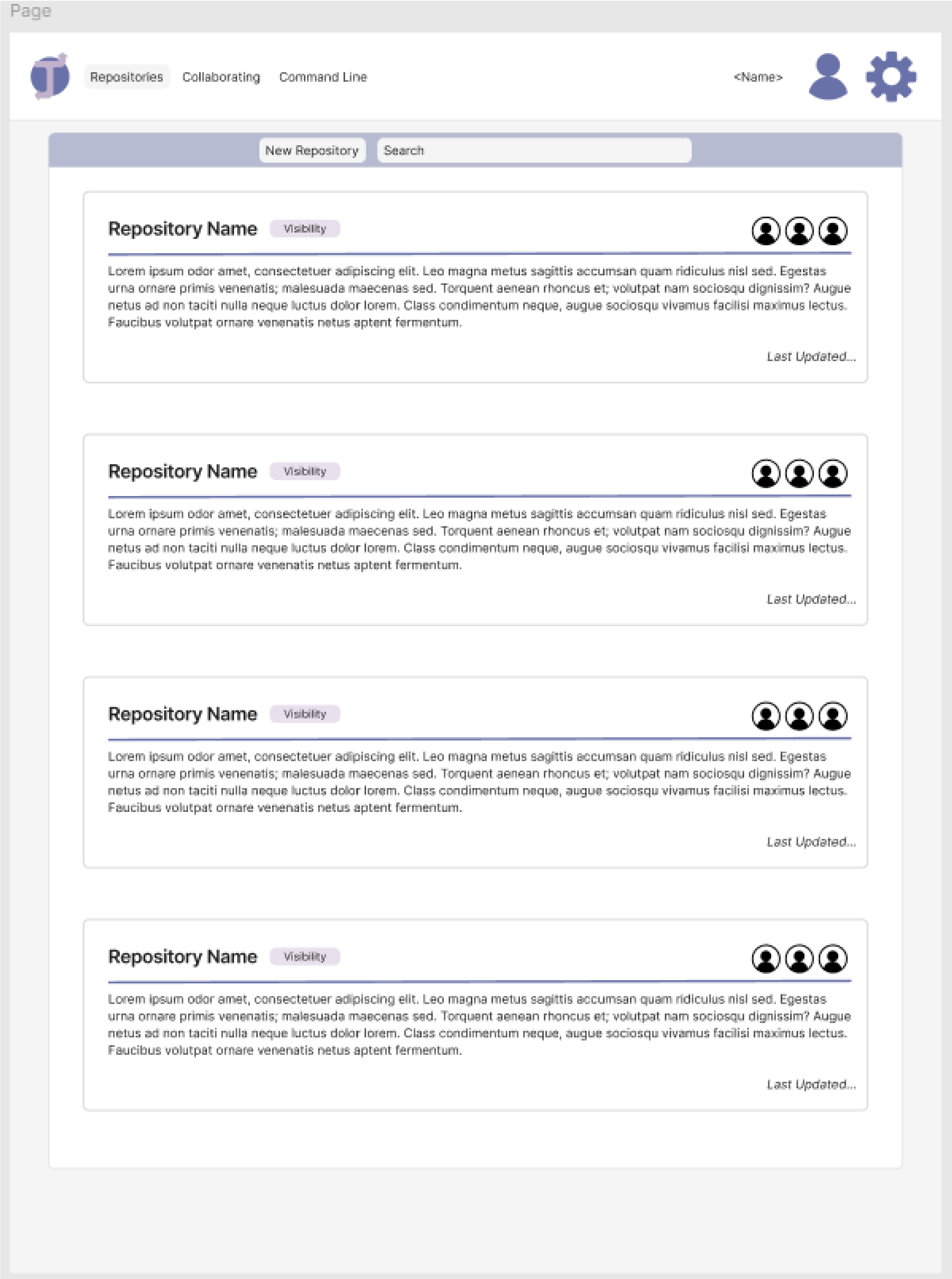


Figure 6.15: High-Fidelity Repositories Page Prototype

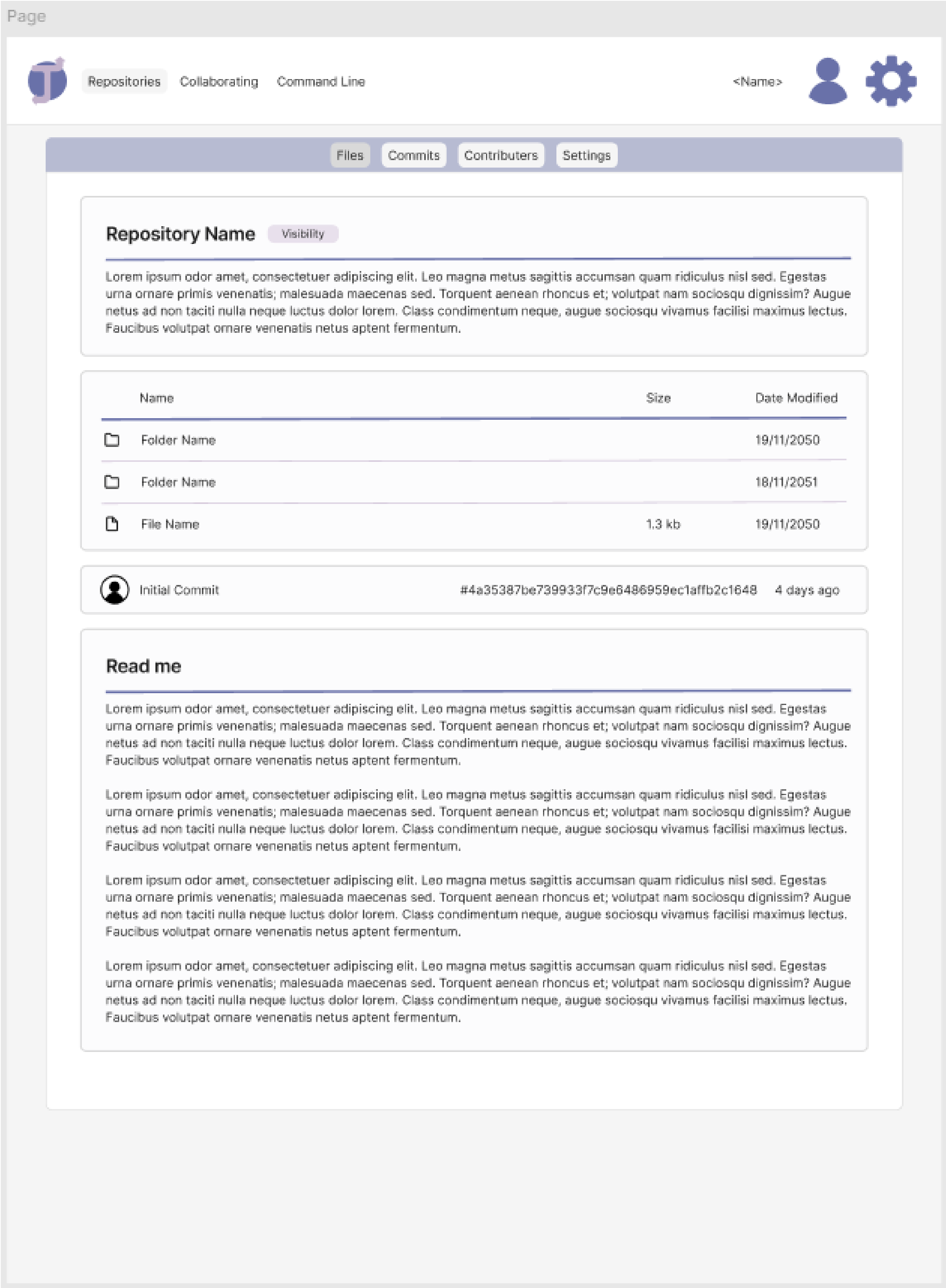


Figure 6.16: High-Fidelity Repository Detail Page Prototype

With the iteratively created designs, the implementation could move forward, confident in the interface and overall user experience.

7 Development

This section details how Janus was developed, covering the technologies and tools used, key implementation decisions and a sprint-by-sprint overview of development progress following Agile methodologies.

7.1 Technology & Tools Used

Janus was implemented with a modern, enterprise grade technology stack chosen for its performance, security and cross platform compatibility. The primary technologies and tools include:

- **Backend (ASP.NET 8, C#):**
  The backend was built with ASP.NET Core 8 for its high performance, robust security framework, and native support for dependency injection and middleware features. This offered built-in solutions for rate limiting, CORS and password hashing (PBKDF2-SHA256), reducing the need for third-party libraries. One drawback of ASP.NET is its larger memory footprint compared to options like Node.js; however, using C# for the backend allowed code to be shared with the CLI, ensuring consistent logic and simpler development. This, along with the mature ecosystem supporting ASP.NET, justifies its selection.

- **Frontend (React 18, JavaScript):**
  The web frontend was implemented using React; its component-based architecture enabling reusability and organised structures. It also has a good ecosystem of libraries and integrates well with JSON Web Token (JWT) based authentication for secure communication with the backend. Other frontend frameworks, such as Angular or Vue.js, were considered but Angular was too heavyweight for the projects scope and while Vue was simpler it lacked the enterprise grade support of React.

- **Database (MySQL, Pomelo Entity Framework Core):**
  MySQL was used for the database for its reliability and ACID-compliant transactions, which are critical for version control systems, ensuring commits and rollbacks are handled safely. The Entity Framework Core (EFC) enabled fast development through object orientated structures and the high performance of MySQL's read/write operations suited the needs of Janus. A limitation of MySQL, however is its horizontal scaling as it requires clustering or sharding of the database and doesn't offer advanced JSON querying available in alternatives like PostgreSQL. These trade-offs were acceptable given the project requirements and data access optimisations to minimise write contention.

- **Command Line Interface (CLI) (.NET 8, C#)**

The Janus CLI was implemented in C# on .NET 8 for its native cross-platform functionality and code sharing with the backend. The CLI architecture was designed to be extensible by using reflection to dynamically load plugins from external assemblies. While the complied CLI binary is larger than comparable tools written in Go or C, sticking with C# ensures consistency across the project.

- **Containerisation (Docker):**
  Docker was used to deploy the microservices (frontend, backend and database). Docker Compose was used to configure the containers, enabling the system to be easily deployed in a consistent environment. Using Docker adds complexity in managing the container network and ensuring CORS/TLS settings are configured correctly. However, the containerised approach aligns with Janus's on-premise deployment goal. At the same time, Kubernetes could have been used for more advanced features; given the scale, it was considered beyond the scope of the project at this time.

- **Continuous Integration (CI):**
  GitHub Actions was utilised for continuous integration. Every push to the repository triggered an automatic workflow, which built the CLI and ensured tests passed on Windows, macOS, and Linux. This ensured that the CLI operated consistently on all platforms.

- **Libraries:**
  A number of libraries and tools were used to develop Janus. For authentication, jwt-decode was used on the frontend to parse token data, and react-markdown was used to display markdown content. The DiffPlex library, implementing the Myers diff algorithm, was utilised in the CLI to compute file differences for the diff and merge commands. The file pattern matching for the ".janusignore" rules was implemented using Glob.

- **Development Utilities:**
  For the development of the project, Visual Studio 2022 was used for the backend and CLI, while Visual Studio Code was used to develop the frontend. The testing frameworks NUnit and Moq were used to test the CLI, enabling simulations of many scenarios and ensuring functions and commands worked as expected on all platforms. Manual testing of the web app was aided by browser tools like Lighthouse, and API endpoints were tested using Postman. During development, the database was inspected using MySQL Workbench to verify data. This combination of technologies and tools ensured a reliable, maintainable system and an efficient development workflow.

7.2 Implementation Overview

Command Line Interface (CLI):

The Janus CLI manages local version control by utilising a hidden directory (.janus/) in each repository to store file blobs and commit metadata, tree objects, and branch information. Core CLI commands were implemented for initialising repositories, staging files, committing snapshots, viewing commit history, branching and merging changes. The merge functionality uses a DiffPlex to identify the differences between file versions; any merge conflicts are flagged for the user to resolve manually. The CLI was designed to work offline for all local operations and only interacts with the remote server when performing network actions such as push, pull, fetch and clone. All commands were developed with cross-platform compatibility in mind and were tested on Windows, macOS and Linux to ensure consistent behaviour. See Appendix B for CLI Command Documentation.

CLI Plugin System:

To support extensibility, Janus's CLI was built with a plugin-based architecture in mind. The CLI scans designated plugin folders for any assemblies implementing the command interface and dynamically loads them using reflection; this design allows users to add custom functionality to Janus. A demo plugin was made to serve as an example for users, with the plugin's description and usage being included in the CLI's help command output. This plugin system adds flexibility to Janus, ensuring it can integrate with other tools and evolve for future needs. See Appendix C for Plugin Development Guide.

Backend Services:

The backend of Janus is a RESTful web API that handles all remote operations. It provides endpoints for user management, repository management and version control actions. All client requests are authenticated using JWT tokens, with the CLI using Personal Access Tokens generated through the front end, ensuring that only authorised actions can occur. The backend enforces role-based access control on repository actions; for example, only a repository owner can delete the repository. Security was a key focus, with all network traffic to and from the server being encrypted with HTTPS and audit logging recording every change to the database.

Web Frontend:

Janus includes a React-based web application that provides a user-friendly interface for repository management. Through the web UI, users can register accounts, log in and create new repositories or manage existing ones. The web app's key features are its repository explorer, which has repository structure and file viewing, a commit history viewer for each branch, and a contributor management interface. The contributor page allows repository owners and admins to invite collaborators with roles for access control. The interface was designed to be responsive and accessible, offering light and dark themes and a collapsible navigation menu for use on smaller screens.

7.3 Sprint Reviews

The sprint reviews, including screenshots and a link to the Monday board, are provided in Appendix E.

7.3.1 Sprint 0 – Project Initiation

Objective:

Establish the project foundations and plan for the development phase.

Deliverables:

- Researched existing DVCS tools and different algorithms to inform design decisions.
- Selected a technology stack that aligned with the project’s needs.
- Set up a GitHub repository and project management tools.
- Wrote the project initiation document.

Outcome:

The project initiation was approved, and the necessary project management tools were set up to guide the development process. This provided a clear direction and a well-defined foundation to ensure the project proceeded smoothly through development.

7.3.2 Sprint 1 Core CLI Implementation

Objective:

Develop a minimum viable set of version control commands in the CLI and containerise all system components.

Deliverables:

- Implemented basic CLI commands: init, add, commit, providing essential functionality for local version control.
- Introduced a command interface to allow easy addition of commands and support the plugin architecture.
- Dockerised the backend API, database and frontend using Docker Compose.
- Prototyped basic endpoints on the backend.
- Set up automated testing for the CLI.

Outcome:

The CLI application became operational with simple functionality and plugin support. Docker containers for the base React app, ASP.NET backend, and MySQL database were set up, demonstrating the feasibility of the chosen stack. This early foundation validated the architecture and set up a stable environment for future features.

7.3.3 Sprint 2 – Authentication & UI Prototyping

Objective:

Implement secure authentication across the system and design a web interface through iterative user feedback.

Deliverables:

- Developed user registration and login functionality using JSON Web Tokens.
- Enforced security with hashed and salted password storage.
- Created initial UI components for login, register, and repository pages and conducted usability testing of the designs.

Outcome:

By the end of Sprint 2, the system supported secure account creation and login for the web application. Early UI prototypes and feedback ensured that the frontend design was user-friendly and accessible before full implementation, establishing a strong foundation for the next steps in user interaction.

7.3.4 Sprint 3 – Enhanced Security

Objective:

Strengthen the security of the system with token management.

Deliverables:

- Implemented Personal Access Tokens for CLI authentication, including token revocation and configurable expiry of token lifetimes.
- Introduce role-based access control (RBAC) into the backend endpoints middleware.
- Improved transport security using a demonstrative self-signed SSL certificate and enabled HTTPS protocols.

Outcome:

This sprint strengthened the system’s security, ensuring it is robust and ready for enterprise use. PATs provided enhanced management of authentication tokens, while RBAC enabled fine-grained control over user access. The use of HTTPS reinforced secure communication, preparing Janus for deployment in sensitive environments.

Sprint 4 – Cross Platform Optimisation

Objective:

Improve the CLI's robustness across operating systems and ensure a consistent yet responsive web interface.

Deliverables:

- Enhanced the CLI's existing commands by improving error and edge-case handling.
- Integrated automated testing into a continuous integration pipeline to run tests across multiple operating systems. This testing caught OS-specific issues early on.
- Improved the React frontend interface by introducing theming and responsive design, ensuring compatibility across different screen sizes.

Outcomes:

Janus became more robust and user-friendly. The CLI was thoroughly tested across major operating systems, ensuring consistency in different environments. The web interface also became more polished and accessible, setting the stage for advanced features without concerns about cross-platform issues.

7.3.5 Sprint 5 – Advanced CLI Features

Objectives:

Expand the CLI features by adding more functionality and improving existing commands.

Deliverables:

- Added advanced repository commands, including log to display the commit history and status to show working directory changes.
- Implemented base branching functionality, enabling users to create, list, and commit to different branches.
- Created unit and integration tests for all new commands and improved test coverage of existing functionality.

Outcomes:

The CLI's functionality grew significantly with these additions, and rigorous testing improved stability and ensured reliability. This sprint demonstrated Agile's iterative design philosophy, where the project progressively matured through each sprint, with added functionality and reliability.

7.3.6 Sprint 6 – Merge & UI Enhancements

Objective:

Introduce branch merging capability and polish both backend data handling and frontend usability.

Deliverables:

- Implemented the merge command in the CLI, including conflict detection using DiffPlex to identify overlapping changes.
- Updated the database schema and backend logic to support merge commits , ensuring data integrity with multiple parent commits.
- Refined the frontend pages for better component reusability and used mocked endpoint responses to display content.

Outcomes:

Janus supported branch merging and conflict resolution, critical features for version control systems. The frontend's mocked content allowed for further interface testing and provided valuable insights into user interaction.

7.3.7 Sprint 7 – Remote Repository Operations

Objective:

Enable distributed version control capabilities by supporting repository cloning and synchronisation with the server.

Deliverables:

- Implemented clone, fetch, pull and push commands in the CLI, allowing users to manage their local and remote repositories.
- Set up backend endpoints to authenticate and transfer repository data efficiently to clients using the CLI and web application.
- Implemented the frontend repository file explorer with breadcrumb navigation and file viewing.

Outcome:

This sprint transformed Janus into a fully distributed version control system. Integrating CLI commands with backend services achieved core functionality for remote operations, enabling synchronisation between local and remote repositories. This was a key milestone in Janus's development, enabling its use in development environments.

7.3.8 Sprint 8 – Compliance and Final Feature Completion

Objective:

Finalise the remaining system features and ensure compliance with regulatory standards.

Deliverables:

- Implemented functionality for safe user account deletion, ensuring cascading deletion across to fulfil the user's "right to be forgotten" (Information Commissioner's Office (ICO), 2018).
- Added contributors' management page to the web interface, utilising RBAC to roles for repository access.
- Created and implemented Terms of Use and Privacy Policy for registration page.

Outcomes:

By the end of Sprint 8, Janus met enterprise compliance and security standards. The system's feature set was finalised, covering local repository management, distributed features, and account management. The introduction of compliance measures, such as the "right to be forgotten" and user role management, completed the system's readiness for deployment.

7.3.9 Sprint 9 – Testing & Documentation

Objective:

Rigorously test the system and produce thorough documentation for users and enterprises.

Deliverables:

- Conduct extensive system testing, identifying and fixing any bugs or security vulnerabilities discovered.
- Performed final performance and stress tests on critical components.
- Clean up the repository codebase in preparation for release.
- Created comprehensive documentation, including a user manual for the CLI and a developer guide for creating plugins (see Appendix B – CLI Documentation and Appendix C – Plugin Developer Guide).

Outcomes:

Sprint 9 resulted in a highly polished product ready for enterprise use. The validation phase ensured the final system was robust, secure, and usable. The testing confirmed system reliability under load, while the documentation ensured clarity in deployment and use.

7.3.10 Sprint 10 – Submission

Objective:

Perform final checks and complete all project deliverables for submission.

Deliverables:

- Submitted all documents and artefacts.
- Updated the GitHub README and added the release version of the CLI to the repository.

Outcomes:

The project concluded with all essential features implemented, creating an enterprise-grade DVCS solution complete with documentation and deployment guides. This final sprint concluded the Agile workflow, delivering a fully tested, complete product in line with the project's original objectives.

8 Testing Strategy

A comprehensive testing strategy was implemented to ensure the reliability, security, and performance of the Janus system. The testing process included automated unit and integration tests for the Command Line Interface (CLI), manual system testing, performance and accessibility evaluations, and user testing. Testing was integrated throughout development to detect issues early and ensure that all functional and non-functional requirements were met.

8.1 Automated Unit & Integration Testing

The Janus CLI was validated through extensive automated tests, using the NUnit framework. Unit tests were focused on fundamental operations such as the tree data structure and diff algorithm, while integration tests verified that the commands functioned as intended. Each CLI command had dedicated tests to ensure correct error handling and edge case scenarios. For commands that required communication with the remote server, the Moq library was used to simulate the backend API responses, making the tests isolated and repeatable.

All automated tests were integrated into a continuous integration (CI) pipeline using GitHub Actions, which utilised a matrix of runners for Windows, macOS and Ubuntu Linux so that it builds and tests each platform. This cross-platform CI process ensured the Janus CLI functioned correctly and consistently across operating systems. Any regressions of platform-specific issues, such as differences in file path handling, could be detected and addressed early during development.

8.2 Manual Testing

In addition to automated testing, manual testing was conducted to evaluate the system under realistic conditions from a user perspective. During development, the frontend was initially tested using simulated API responses and sample data to verify that content was being displayed correctly before the backend system communication was enabled. Once the backend API was available, end-to-end manual tests of the web application were performed by following typical user workflows and identifying edge cases. This approach allowed issues to be systematically isolated to their relevant parts and resolved.

The backend RESTful API was manually verified using a collection of Postman requests to cover all endpoint scenarios. Each endpoint was tested with both valid and invalid inputs to ensure correct responses and robust error handling. This manual API testing was repeated throughout development to ensure that they behaved as intended.

Manual testing provided an opportunity to gain a user perspective on Janus, enabling issues on usability to be detected and improved. These manual tests were necessary to ensure the automated tests were configured correctly and handled all edge cases.

8.3 Performance & Accessibility Testing

Performance and accessibility checks were critical to the testing strategy. The web interface testing used Google Lighthouse to check page load performance, accessibility compliance and adherence to best practices. Lighthouse enabled data to be collected on First Contentful Paint, Time to Interactive and an overall Accessibility score. This was run on key pages to ensure that performance remained high, accessibility issues were addressed, and a compliant user experience was created.

CLI performance was also measured by stress testing with large repositories in terms of file and content size. Examples of these operations included repository cloning, committing changes, and pushing to the remote repository. These tests confirmed that all operations were completed within an acceptable time frame, ensuring that the performance requirements were met.

8.4 User Testing

Finally, user testing was carried out to assess the usability and overall user experience of both the CLI and frontend. A small group of participants was asked to perform realistic tasks using Janus, including registering a new account, creating a new repository, cloning a repository, making a commit and pushing changes. Each testing session was observed in person to note any confusion, hesitation or problems the participants might not notice or report.

After completing the tasks, participants filled out a structured Google Forms survey to rate aspects of the system, such as its ease of navigation, interface clarity, and intuitiveness of CLI commands. Based on anything noted during the testing process, further questions were also asked to gather more in-depth feedback about areas in need of improvement. This user testing ensured Janus met its requirements to provide a good user experience.

9 End-Project Report

The Janus project set out to deliver an on-premise, secure distributed version control system (DVCS) for enterprise use. Its core functionality was implemented; supporting essential version control operations such as initialising a new repository, staging files, committing changes, branching, merging and synchronisation with remote repositories using push, pull fetch and clone. The Janus solution aligns with the original vision of the project by operating entirely within an internal network, with authenticated user access, encrypted communications and detailed audit logging to satisfy enterprise data sovereignty and compliance. Providing a functional DVCS that achieves the projects goals.

- **Core DVCS functionality:** Achieved. All required commands were implemented and verified through testing and the CLI operates across multiple platforms. The web interface allows users to manage their remote repository. These satisfy the fundamental requirements of a DVCS.
- **Security and compliance:** Achieved. The system uses user authentication, role-based access control, personal access token and HTTPS/TLS encryption. These satisfy the specified security criteria (FR-11, FR-12, FR-13, and FR-15) and the on-premise deployment with audit logs ensures compliance with enterprise regulations.
- **Plugin architecture:** Achieved. A plugin framework was implemented for the CLI, which users can develop and load custom plugins; a demo plugin was provided to show this functionality. This extensibility satisfies the requirement (FR-19).
- **Automated backup:** Not achieved. The plan feature for automatic repository backups was dropped from the scope due to time constraints. Backing up the repositories manually using the Docker volume is possible for users however automatic this process would be more reliable and reduce risk.
- **Commit diff viewing**: Partially achieved. While the CLI has a command to compare commits the web interface lacks the ability to view the commit diffs. This feature would greatly aid collaboration in enterprise environments.
- **Web based commits**: Not achieved. The ability to edit files and commit the changes directly through the web interface was removed from the scope to focus on core features.

Overall, Janus meets all essential objectives, as verified by comprehensive testing against the functional and non-functional requirements. The projects scope was reduced to priorities core features, which was necessary as time constraints and ambitious initial plans meant that advanced features had to be left out from the delivered solution. In particular addressing cross platform intricacies, such as differing file paths and handling edge cases in commands, took significant time during development; however, they did not prevent the Janus from achieving its core aims.

10 Reflections

The project's agile approach, using two-week sprints and an iterative structure, provided valuable feedback loops and adaptability. Regular planning and reviewing ensured the development was kept on track, and project management tools allowed priorities to be dynamically adjusted. This flexibility worked well, enabling feedback and scope changes without ruining the plan. However, initial estimates for tasks were too optimistic, suggesting future iterations should be allocated more time to account for unexpected issues.

The chosen technology stack proved effective for the project. Using .NET C# for both the CLI and backend API proved an efficient solution for quick, consistent development. Reacts' frontend offered response and reusable components, saving time and effort, while Docker containerisation ensured a consistent deployment environment. These decisions sped up development and helped maintain code quality. At the same time, some learning curves were encountered early on, involving Docker networking and handling cross-platforms consistently.

Testing and automation were critical to the development process. The continuous integration pipeline for the CLI ensured that new changes did not negatively affect the system, quickly catching regressions and compatibility issues. In the future, expanding the coverage of automated testing across the whole system would have saved time during development and improved overall quality.

Several technical challenges arose during development. One of these was ensuring consistent file path handling and other differences across operating systems, which led to frequent refactoring early on. To guarantee data integrity, database interactions had to use transactions with robust error handling, which increased the complexity of backend systems. Had these issues been identified earlier, quicker steps could have been taken to mitigate these problems.

On a personal level, the author learned the importance of scope management and adaptability. The initial project scope was too ambitious, meaning that as the project progressed, it had to be repeatedly redone to focus on critical features. In future projects, it would be effective to start with a narrower scope, which can be expanded later on if possible. The author's experience in time management, effective project management, and professional practices has improved throughout the project's lifecycle.

In summary, utilising structured methodologies and thorough testing were key factors in creating a functional system. For future work, the author would implement more rigorous initial planning, a more focused scope, and improved automated test coverage early on. These lessons will inform the future approach taken to development and project management, providing a stronger foundation for future endeavours.

11 Conclusion

The Janus project delivered a working on-premise DVCS solution that meets its project vision and core objectives. It enables enterprises to maintain complete internal control over their codebases without relying on external systems. All essential requirements were fulfilled, and key features such as the plugin framework, on-premise hosting, and cross-platform compatibility demonstrate the system's extensibility and flexibility.

Janus provided many lessons throughout its development, highlighting the necessity of a clear, focused scope and risk mitigation for complex projects. Future work on Janus should focus on extending its functionality and robustness:

- **Implement dropped features:** To improve reliability and user experience, all originally planned functionality, such as automated repository backups and a web diff viewer, should be implemented.
- **Advanced version control operations:** The Janus system would benefit from more advanced repository management commands, such as history rewriting and automated conflict resolution.
- **Performance optimisation:** Comprehensive benchmarking and profiling using real-world simulations should be conducted to identify and optimise system bottlenecks.
- **Security enhancements:** Formal security audits and penetration testing of Janus ensure that the solution is robust and data is protected, resulting in a solid enterprise-ready solution.
- **Enterprise integration testing:** Testing Janus in real-world enterprise environments would identify issues with deployment, scalability, and user experience that would have been difficult to observe during the testing phase.
- **Scalability improvements:** Container orchestration, like Kubernetes with clustering and load balancing, should be explored to allow Janus to scale more effectively for larger teams.

In conclusion, Janus achieves its defined goals as a secure internal DVCS, but additional development will be necessary to match the feature set and maturity of leading version control systems.

12 References

Akinsola, O. K., 2025. *Ensuring Ethical Conduct and Legal Compliance within Corporate Boards: Legal Standards, Best Practices, and Accountability,* s.l.: s.n.

Ali, H., 2022. *Implementing CI/CD Pipelines with Jenkins: Common Challenges and Solutions.* [Online]
Available at:
https://www.researchgate.net/publication/374955515_Performance_Analysis_of_Devops_Practice_Implementation_Of_CICD_Using_Jenkins
[Accessed 26 4 2025].

Alnafessah, A. et al., 2021. Quality-aware DevOps Research: Where Do We Stand?. *IEEE Software,* 38(6), pp. 44-53.

Anjum, N. & Alam, S., 2019. A Comparative Analysis on Widely used Web Frameworks to Choose the Requirement based Development Technology. *International Advanced Research Journal in Science, Engineering and Technology (IARJSET),* 6(1), pp. 16-24.

Bhanushali, A., 2023. Ensuring Software Quality Through Effective Quality Assurance Testing: Best Practices and Case Studies. *International Journal of Advances in Scientific Research and Engineering,* 9(2), pp. 45-52.

Bhattacharya, P., 2018. Aligning enterprise systems capabilities with business strategy: An extension of the strategic alignment model (SAM) using enterprise architecture. *Procedia Computer Science,* Volume 138, pp. 664-671.

Bojović, Ž., 2024. *Application of Network Function Virtualization in Modern Computer Environments.* Boston: Now Publishers.

Canedo, E. D., Nunes, R. R. & Serrano, A. L. M., 2025. *Cybersecurity Risk Assessment Through Analytic Hierarchy Process: Integrating Multicriteria and Sensitivity Analysis.* s.l., SCITEPRESS – Science and Technology Publications, p. 117–128.

Chacon, S. & Straub, B., 2020. *Pro Git.* 2nd ed. ed. New York: Apress.

Chang, H., Zankl, W. & Lee, W. W., 2016. *An Ethical Approach to Data Privacy Protection.* [Online]
Available at: https://www.isaca.org/resources/isaca-journal/issues/2016/volume-6/an-ethical-approach-to-data-privacy-protection
[Accessed 7 April 2025].

De la Torre, C., 2016. *Containerized Docker Application Lifecycle with Microsoft Platform and Tools,* s.l.: Microsoft Corporation.

Dybå, T. & Dingsøyr, T., 2008. Empirical Studies of Agile Software Development: A Systematic Review. *Information and Software Technology,* 50(9-10), pp. 833-859.

European Union, 2016. *GDPR.* [Online]
Available at: https://www.legislation.gov.uk/eur/2016/679/contents
[Accessed 7 April 2025].

European Union, 2016. *General Data Protection Regulation (GDPR) Article 45: Transfers on the Basis of an Adequacy Decision.* [Online]
Available at: https://gdpr-info.eu/art-45-gdpr/
[Accessed 26 4 2025].

Gao, J., Tsao, J. & Wu, Y., 2003. *Testing and Quality Assurance for Component-Based Software.* Hoboken, NJ: Wiley.

Gartner, 2023. *Hype Cycle for Cloud Security,* Stamford: Gartner Research.

Gowda, P. & Gowda, A. N., 2024. Best Practices in REST API Design for Enhanced Scalability and Security. *Journal of Artificial Intelligence, Machine Learning and Data Science,* 2(1), p. 827–830.

Hibbs, C., Jewett, S. & Sullivan, M., 2009. *The Art of Lean Software Development: A Practical and Incremental Approach.* 1 ed. Sebastopol: O'Reilly Media.

Högblom, M. & Green, V., 2013. *Version Control Systems in Corporations: Centralized and Distributed,* Gothenburg: s.n.

Hudaib, A., Masadeh, R. & Qasem, M. H., 2018. Requirements Prioritization Techniques Comparison. *Modern Applied Science,* 12(2), pp. 126-138.

IBM Security, 2023. *Cost of a Data Breach Report 2023,* s.l.: IBM.

Ilag, B. N., Baljoshi, A. K. P., Sangale, G. J. & Athave, Y., 2024. *Mastering GitHub Enterprise Management and Administration: A Guide for Seamless Management and Collaboration.* Berkeley, CA: Apress.

Information Commissioner’s Office (ICO), 2018. *Right to erasure under the UK GDPR (Article 17).* [Online]
Available at: https://ico.org.uk/for-organisations/uk-gdpr-guidance-and-resources/individual-rights/individual-rights/right-to-erasure/
[Accessed 26 4 2025].

Information Commissioner's Office, 2025. *Transparency – Accountability Framework.* [Online]
Available at: https://ico.org.uk/for-organisations/uk-gdpr-guidance-and-resources/accountability-and-governance/accountability-framework/transparency
[Accessed 8 April 2025].

ISMS, 2020. *What’s Involved in an ISO 27001 Audit?.* [Online]
Available at: https://www.isms.online/iso-27001/what-is-the-iso-27001-audit-process/
[Accessed 26 4 2025].

Jones, M. B., Bradley, J. & Sakimura, N., 2015. *RFC 7519: JSON Web Token (JWT).* [Online]
Available at: https://datatracker.ietf.org/doc/html/rfc7519
[Accessed 7 April 2025].

Kansal, S. & Balasubramaniam, V. S., 2024. Microservices Architecture in Large-Scale Distributed Systems: Performance and Efficiency Gains. *Journal of Quantum Science and Technology,* 1(4), p. 633–648.

Khalid, A., Haneef, F. & Waseem, F., 2025. Enhancing Open-Source Projects: The Synergy Between Code Readability Metrics and User Experience. *International Journal of Innovations in Science & Technology,* 7(1), pp. 129-145.

Knittl-Frank, D., 2010. *Analysis and Comparison of Distributed Version Control Systems: Git, Mercurial, and Bazaar,* s.l.: s.n.

Koç, A. & Tansel, A. U., 2011. *A Survey of Version Control Systems.* s.l., International Institute of Informatics and Systemics (IIIS).

Kristallovich, F. & Eisfeld, H., 2020. *Dark Mode Interfaces: Perceptions and User Preferences.* [Online]
Available at: https://urn.kb.se/resolve?urn=urn:nbn:se:hj:diva-50563
[Accessed 7 April 2025].

Kuhn, D., 2010. *Distributed Version Control Systems,* s.l.: s.n.

Loeliger, J. & McCullough, M., 2023. *Version Control with Git.* 3rd ed. ed. Sebastopol: O'Reilly Media.

Majrashi, K., Hamilton, M., Uitdenbogerd, A. L. & Al-Megren, S., 2020. Cross-Platform Usability Model Evaluation. *Multimodal Technologies and Interaction,* 4(4), p. 80.

Marjanovic, D., 2006. *Developing a Meta Model for Release History Systems,* Zurich: s.n.

Microsoft, 2024. *Prepare for Azure Classic Administrator Roles Retirement and Transition to Azure Role-Based Access Control (RBAC).* [Online]
Available at: https://learn.microsoft.com/en-us/azure/cost-management-billing/manage/classic-administrator-retire
[Accessed 26 4 2025].

Mukherjee, P., 2010. *A Fully Decentralized, Peer-to-Peer Based Version Control System,* Darmstadt: s.n.

National Institute of Standards and Technology, 2017. *NIST Special Publication 800-63B.* [Online]
Available at: https://pages.nist.gov/800-63-3/sp800-63b.html
[Accessed 7 April 2025].

Nielsen, J., 1995. *10 Usability Heuristics for User Interface Design,* s.l.: s.n.

Nikander, J., 2024. *Version Control of PLC Blocks and Code,* Vaasa: s.n.

NIST, 2017. *Digital Identity Guidelines.* [Online]
Available at: https://nvlpubs.nist.gov/nistpubs/SpecialPublications/NIST.SP.800-63b.pdf
[Accessed 7 April 2025].

Norman, D. A., 2013. *The Design of Everyday Things.* Cambridge, MA: MIT Press.

Oracle, 2025. *Transactions.* [Online]
Available at: https://docs.oracle.com/en/database/oracle/oracle-database/19/cncpt/transactions.html
[Accessed 7 April 2025].

OSI, 2024. *The License Review Process.* [Online]
Available at: https://opensource.org/licenses/review-process
[Accessed 7 April 2025].

OWASP, 2025. *Password Storage Cheat Sheet.* [Online]
Available at:
https://cheatsheetseries.owasp.org/cheatsheets/Password_Storage_Cheat_Sheet.html
[Accessed 7 April 2025].

OWASP, 2025. *Transport Layer Security Cheat Sheet.* [Online]
Available at:
https://cheatsheetseries.owasp.org/cheatsheets/Transport_Layer_Security_Cheat_Sheet.html
[Accessed 7 April 2025].

Parnas, D. L., 1972. *On the criteria to be used in decomposing systems into modules,* New York: Association for Computing Machinery.

Ponemon Institute, 2023. *The State of Cloud Security Survey,* s.l.: Ponemon Institute Research.

Rellermeyer, J. S., 2011. *Modularity as a Systems Design Principle,* s.l.: s.n.

Sarioguz, O., Miser, E. & Teslim, B., 2024. Integrating AI in financial risk management: Evaluating the effects of machine learning algorithms on predictive accuracy and regulatory compliance. *International Journal of Science and Research Archive,* 13(2), p. 789–811.

Scherenberg, F. v., Hellmeier, M. & Otto, B., 2024. *Data Sovereignty in Information Systems.* [Online]
Available at: https://doi.org/10.1007/s12525-024-00693-4
[Accessed 8 April 2025].

Singh, R., 2021. A Survey on Version Control Systems: Past to Present. *Journal of Software Engineering and Applications,* 14(4), pp. 147-162.

Singh, R., Goyal, H., Kumar, K. & Arora, G., 2023. *A Command-Line Interface Tool for Efficient Git Workflow: “Commandeer”.* s.l., IEEE, pp. 1-6.

Souppaya, M. & Kent, K., 2006. [Online]
Available at: https://nvlpubs.nist.gov/nistpubs/Legacy/SP/nistspecialpublication800-92.pdf
[Accessed 7 April 2025].

Stöcklin, T. T., 2022. *Evaluating SSH for Modern Deployments,* s.l.: 2022.

Svitla, 2024. *Intellectual Property in Software Development.* [Online]
Available at: https://svitla.com/blog/intellectual-property-software/
[Accessed 8 April 2025].

Thomas, D. & Hunt, A., 2000. In: *The pragmatic programmer: from journeyman to master.* Boston: Addison-Wesley Longman Publishing Co., Inc..

U.S. Cybersecurity and Infrastructure Security Agency (CISA), 2023. *Cloud Security Technical Reference Architecture,* s.l.: Cybersecurity and Infrastructure Security Agency.

UK Government, 2018. *Data Protection Act.* [Online]
Available at: https://www.legislation.gov.uk/ukpga/2018/12/contents
[Accessed 7 April 2025].

Verizon, 2024. *2024 Data Breach Investigations Report,* s.l.: Verizon.

WCAG, 2024. *Web Content Accessibility Guidelines.* [Online]
Available at: https://www.w3.org/TR/WCAG21/
[Accessed 7 April 2025].

World Wide Web Consortium (W3C), 2018. *Web Content Accessibility Guidelines (WCAG) 2.1.* [Online]
Available at: https://www.w3.org/TR/WCAG21/
[Accessed 26 4 2025].

13 Appendices

Appendix A – Installation Instructions

Build Docker system

1. Prerequisites
   Install:

   Docker Engine 20.10+

   Docker Compose 2.15+

2. Verify with:

   bash

   ```
   docker –version && docker-compose --version
   ```

3. Clone the Janus Repository

   bash

   ```
   git clone https://github.com/benjaminsanderswyatt/COMP3000-JanusVersionControl.git
   cd COMP3000-JanusVersionControl/Fullstack
   ```

4. Install frontend dependencies

   bash

   ```
   cd frontend
   npm install
   cd ..
   ```

5. Copy & configure env

   bash

   ```
   cp .env.example .env
   ```

   Configure .env file as desired

6. Build
   Run the up-internal.sh script to build the Janus containers.

   bash

   ```
   ./up-internal.sh
   ```

Run CLI

Head to the GitHub repository and follow the instructions given for the release version.

Here is a copy of the instructions included in the release:

Janus CLI

A cross-platform, command-line interface for Janus in a single folder that works on Linux (x86_64 & ARM64), macOS (x86_64 & ARM64), and Windows.

Contents

```
janus-cli/
├── README.md          # This file
├── janus              # Unix shell launcher
├── janus.cmd          # Windows batch launcher
├── janus.ps1          # PowerShell launcher
├── linux-x64/         # Linux x86_64 single-file binary + runtime
│   └── Janus          # Executable
├── linux-arm64/       # Linux ARM64 single-file binary + runtime
│   └── Janus          # Executable
├── osx-x64/           # macOS x86_64 single-file binary + runtime
│   └── Janus          # Executable
├── osx-arm64/         # macOS ARM64 single-file binary + runtime
│   └── Janus          # Executable
└── win-x64/           # Windows x64 single-file binary + runtime
    └── janus.exe      # Executable
```

Installation

Download the janus-cli.zip from the releases page.

1. Unpack the archive to a directory of your choice:

```bash
unzip janus-cli.zip -d /path/to/install
```

2. Ensure executability (Unix only):

```bash
cd /path/to/install/janus-cli
chmod +x janus
chmod +x linux-*/Janus osx-*/Janus
```

3. Add to your PATH:

- Linux / macOS (~/.bashrc, ~/.zshrc…):

```bash
export PATH="$HOME/path/to/install/janus-cli:$PATH"
```

- Windows (via System Properties -> Environment Variables -> Path):

```bash
C:\path\to\install\janus-cli
```

Reload your shell or open a new terminal for the changes to take effect.

Usage

Once installed run:

janus help

The janus launcher will detect your OS and architecture, and use the appropriate self-contained binary.

Examples

```bash
# Show help
janus help
# Initalise a repository
janus init
```

Troubleshooting

- Permission denied on Unix: ensure janus and the executables have execute permissions.
- Command not found: verify that the install path is correctly added to your $PATH.

- Unsupported OS/ARCH: make sure you have the matching binary folder (linux-x64, linux-arm64, osx-x64, osx-arm64, or win-x64).

System Requirements

These are the minimum requirements in order for the Docker system and CLI to run:

Table 13.1: Minimum System Requirements for Docker Containers

Component	Minimum Requirements
OS	64-bit Linux
CPU	2 cores, 1.6GHz
RAM	4GB (8GB recommended)
Storage	10GB free (images & volumes)
Software	Docker Engine 20.10+, Docker Compose 2.15+

Table 13.2: Minimum System Requirements for CLI

Component	Minimum Requirements
OS	64-bit Windows 10+, macOS 13+, Linux
CPU	1 core, 1.8GHz
RAM	2GB
Storage	1GB free (for the zip)
Software	none

Appendix B - CLI Documentation

The full Markdown file is also available in the repository under /Documentation/CLI_DOCUMENTATION.md:

Janus CLI Documentation

Janus is a version control command-line tool for managing codebases. It provides commands for repository configuration, managing branches, making commits, and synchronizing changes with remote servers. This documentation describes each command's purpose, usage, and examples.

Table of Contents

Overview

Janus is a version control CLI that:

- **Manages repository configuration:** Adjust repository properties with config.
- **Supports local version control:** Stage changes using add, commit with commit, and view commit logs and differences.
- **Manages branches:** Create, list, switch, merge branches, and revert to previous commits.
- **Handles authentication:** Use login to store your credentials for remote repositories.
- **Remote repository interactions:** Clone, fetch, pull, and push changes with remote repositories.

Getting Started

Before using most commands, ensure you have done the following:

- Installed Janus on your system.
- Log in with your credentials using the janus login command.
- Initialised a repository with janus init if not already done.

Commands Reference

Each command supports a **usage string** that explains arguments, options, and examples. The following sections detail each command.

help

Description:
Displays a list of all available commands (with short descriptions) or detailed usage of a specific command if provided.

Usage:

janus help [command]

Examples:
- Show a list of all commands: janus help - Get detailed help about the clone command: janus help clone

---

config

Description:
Manages local and global configuration settings. Includes subcommands for IP configuration and repository properties.

Usage:

janus config <subcommand> [arguments]

Subcommands: - ip get [--global]: Gets the configured IP (local or global) - ip set <value> [--global]: Sets the IP configuration - ip reset [--global]: Removes the IP configuration file - repo get <property>: Gets a repository property (e.g., is-private, description) - repo set <property> <value>: Sets a repository property

Examples:

janus config ip get
janus config ip set 192.168.1.100 --global
janus config repo get is-private
janus config repo set description "My project"

---

login

Description:
Prompts you for your user credentials username, email, and personal access token (which is only required for commands interacting with a remote repository) and saves them for later repository interactions.

Usage:

janus login

Example:

janus login

Follow on-screen prompts to enter your credentials.

---

remote

Description:
Manages remote repository settings. This command supports adding, removing and listing remote repositories.

Usage:

janus remote <subcommand> [arguments]

Subcommands: - add <name> <endpoint>: Adds a new remote repository. - remove <name>: Removes an existing remote repository. - list: Lists all configured remote repositories.

Example:

janus remote add origin janus/repo/user

---

clone

Description:
Clones a repository from a remote server to your local machine, initialising repository folders and setting branch configurations.

Usage:

janus clone <endpoint> [branch]

Parameters:
- <endpoint>: Repository endpoint in the format janus/{owner}/{repoName}. - [branch]: (Optional) Branch to check out. Defaults to main if not specified.

Example:

janus clone janus/owner/repo main

---

fetch

Description:
Fetches the latest commits and repository information from a remote repository. Updates local commit history and repository configuration.

Usage:

janus fetch [remote]

Examples:

janus fetch
janus fetch upstream

---

pull

Description:
Synchronises local branches with the fetched remote commits. It analyzes each remote branch and fast-forwards if possible.

Usage:

janus pull

Example:

janus pull

---

push

Description:
Pushes local commits to a remote repository, uploading new commits along with file objects if changes are found.

Usage:

janus push [remote] [branch]

Examples:

janus push
janus push origin main

---

init

Description:
Initialises the Janus repository by creating the necessary directory structure (.janus folder) and configuration files.

Usage:

janus init

Example:

janus init

---

add

Description:
Stages the specified files or directories for the next commit.

Usage:

janus add <file(s) or directory>

Examples:

janus add file.txt
janus add folder
janus add --all

---

commit

Description:
Commits the staged changes to the repository with a mandatory commit message.

Usage:

janus commit <commit message>

Example:

janus commit "Fixed bug in file upload"

---

revert

Description:
Reverts your repository state to the state of a previous commit.

Usage:

janus revert <commit-hash> [--force]

Examples:

janus revert abcdef
janus revert 123456 --force

---

log

Description:
Displays the commit history. You can filter the results by branch, author, and date times, or limit the number of results displayed.

Usage:

janus log [options]

Example:

janus log --branch main --author Alice --since 2023-01-01 --limit 10

---

diff

Description:
Displays differences between commits.

Usage:

janus diff [options] [<commit> [<commit>]] [--path <file>]

Examples:

janus diff
janus diff --staged
janus diff abc123 def456

janus diff abc123 def456 --path file.txt
janus diff abc123 --parent

---

merge

Description:
Merges changes from another branch into the current branch.

Usage:

janus merge <branch>

Example:

janus merge featureBranch

---

create_branch

Description:
Creates a new branch from the current HEAD commit.

Usage:

janus create_branch <branch name>

Example:

janus create_branch featureBranch

---

list_branch

Description:
Lists all branches in the repository.

Usage:

janus list_branch

Example:

janus list_branch

---

switch_branch

Description:
Switches the working directory to a different branch.

Usage:

janus switch_branch <branch name> [--force]

Examples:

janus switch_branch develop
janus switch_branch featureBranch --force

---

status

Description:
Displays the current repository status.

Usage:

janus status

Example:

janus status

Workflow and Usage Tips

- **Initialization:**
  Always start with janus init in a new project directory.
- **Making Changes:**
  Use janus add to stage new or modified files, then commit using janus commit with a descriptive message.
- **Branch Management:**
  Use janus create_branch to start new features and janus switch_branch to safely change to the new branch. Display existing branches with janus list_branch.
- **Remote Operations:**
  Clone repositories from a remote source with janus clone. Keep your repository up to date by using janus fetch, janus pull, and janus push.
- **Version History:**
  Review commit history with janus log and compare commit differences using janus diff.
- **Handling Conflicts and Reverts:**
  Merge with janus merge or revert using janus revert.

Plugin System

Janus supports an extensible plugin architecture that allows developers to add custom commands by placing compiled .dll files into the designated plugin folders.

How It Works

At startup, Janus loads all built in commands and scans for external plugin assemblies located in:

- **Local Plugins:**
  <project-root>/.janus/plugins
- **Global Plugins:**
  <user-home>/Janus/plugins

Each plugin must implement the ICommand interface defined in Janus.Plugins.

ICommand Interface

```
public interface ICommand
{
   string Name { get; }
   string Description { get; }
   string Usage { get; }
   Task Execute(string[] args);
}
```

BaseCommand (Convenience Class)

For easier plugin creation, extend the BaseCommand class which provides access to logging and paths:

```
public abstract class BaseCommand : ICommand
{
   protected ILogger Logger { get; }
   protected Paths Paths { get; }

   public BaseCommand(ILogger logger, Paths paths)
   {
      Logger = logger ?? throw new ArgumentNullException(nameof(logger));
      Paths = paths ?? throw new ArgumentNullException(nameof(paths));
   }

   public abstract string Name { get; }
   public abstract string Description { get; }
   public abstract string Usage { get; }
   public abstract Task Execute(string[] args);
}
```

Example Plugin

Below is an example of a simple plugin that outputs “hello world”:

```
using Janus.Plugins;

namespace PluginDemo
{
   public class PluginDemo : BaseCommand
   {
      public PluginDemo(ILogger logger, Paths paths) : base(logger, paths) { }

      public override string Name => "plugin";
      public override string Description => "A demo command for plugins";
      public override string Usage =>
@"janus plugin
This command shows:
   - Says hello world
Example:
   janus plugin";

      public override async Task Execute(string[] args)
      {
         Logger.Log("Plugin Demo - hello world");
      }
   }
}
```

Using Plugin Commands

Once the plugin is compiled into a .dll and placed in the plugin directory, the command is picked up by Janus to be used.

- Run janus help to see the added plugin command listed.
- Execute the plugin command as you would with any built in command:

Additional Notes

- **User Prompts:**
  Some commands require confirmation (e.g., switch_branch, pull, revert).
- **Configuration:**
  Use config for managing IP settings and repo metadata.
- **Error Logs:**
  Errors and conflicts are logged to assist in troubleshooting.

- **Plugin Developer Guide:**
  For more detailed information on developing plugins, refer to the Plugin Developer Guide.

Appendix C - Plugin Developer Guide

The full Markdown file is also available in the repository under /Documentation/PLUGIN_DEVELOPMENT_GUIDE.md:

Plugin Developer Guide

Janus supports an extensible plugin architecture that allows you to add custom commands by creating a plugin assembly (a .dll file). This guide explains the plugin system and how you develop plugins for Janus CLI.

Table of Contents

Overview

Janus can automatically load custom commands from external plugin assemblies. By following this guide, you will create a plugin that integrates with Janus and extends its functionality.

Plugins are compiled as .dll files and must implement the ICommand interface defined in the Janus.Plugins namespace. Janus searches designated plugin directories and loads any compatible commands at startup.

Architecture

Plugin Loading

When Janus starts, it scans two key directories for plugin assemblies:

- **Local Plugins:** Located in <project-root>/.janus/plugins

- **Global Plugins:** Located in <user-home>/.janus/plugins

For each plugin assembly (.dll file), Janus uses reflection to discover classes implementing the ICommand interface. It instantiates these classes, supplying dependencies such as ILogger and Paths objects via the class constructor.

Developing a Plugin

Implementing the ICommand Interface

A base plugin must implement the ICommand interface:

```
namespace Janus.Plugins
{
   public interface ICommand
   {
      string Name { get; }         // The command name used to invoke the plugin
      string Description { get; }  // A short description of the command
      string Usage { get; }        // Detailed usage instructions for the command
      Task Execute(string[] args); // The logic that executes when the command is run
   }
}
```

Using BaseCommand for Convenience

Janus provides a BaseCommand abstract class that implements ICommand and helps by providing:

- A logging mechanism via an ILogger instance.
- Access to paths and other utility functions via a Paths object.

To create your plugin, simply extend the BaseCommand class:

```
using Janus.Plugins;

public abstract class BaseCommand : ICommand
{
   protected ILogger Logger { get; }
   protected Paths Paths { get; }

   protected BaseCommand(ILogger logger, Paths paths)
   {
      Logger = logger ?? throw new ArgumentNullException(nameof(logger));
      Paths = paths ?? throw new ArgumentNullException(nameof(paths));
   }
```

```
    public abstract string Name { get; }
    public abstract string Description { get; }
    public abstract string Usage { get; }
    public abstract Task Execute(string[] args);
}
```

This simplifies development, by providing access to commonly used services.

Example Plugin

Below is a simple example “Hello World” plugin:

```
using Janus.Plugins;

namespace PluginDemo
{
    public class PluginDemo : BaseCommand
    {
        public PluginDemo(ILogger logger, Paths paths) : base(logger, paths) { }

        public override string Name => "plugin";
        public override string Description => "A demo command for plugins";
        public override string Usage =>
@"janus plugin
This command displays:
    - A greeting message
Example:
    janus plugin";

        public override async Task Execute(string[] args)
        {
            // Write your custom logic here.
            Logger.Log("Plugin Demo - hello world");
        }
    }
}
```

Compile this plugin into a .dll and ensure it references the proper Janus assemblies so that the ICommand, ILogger, Paths, and BaseCommand types are available.

Deployment

1. Compile Your Plugin: Ensure your project targets a compatible .NET runtime version with Janus.

2. Place the DLL:

   For local plugins: Copy the compiled DLL to /.janus/plugins.

   For global plugins: Copy the DLL to /.janus/plugins.

3. Run Janus: When you start Janus, it will scan the plugin directories and automatically include your plugin command.

4. Test Your Plugin:

   List available commands with:

   janus help

Additional Considerations

- Constructor Dependencies: Your plugin classes should have a constructor that accepts an ILogger and Paths. Janus uses reflection to instantiate your plugin, so ensure your constructor parameters match those expected types.
- Error Handling: Use try catch blocks within your Execute method to manage any runtime errors gracefully.
- Documentation: Document your plugin's usage string and commands clearly for users who will execute your command.
- Multiple Commands: An assembly can contain multiple classes implementing ICommand. Each will be loaded and listed as separate commands in Janus.
- Versioning and Compatibility: Keep track of changes in the Janus plugin interfaces. When updating Janus, verify that your plugins are still compatible.

Appendix D - Diagrams

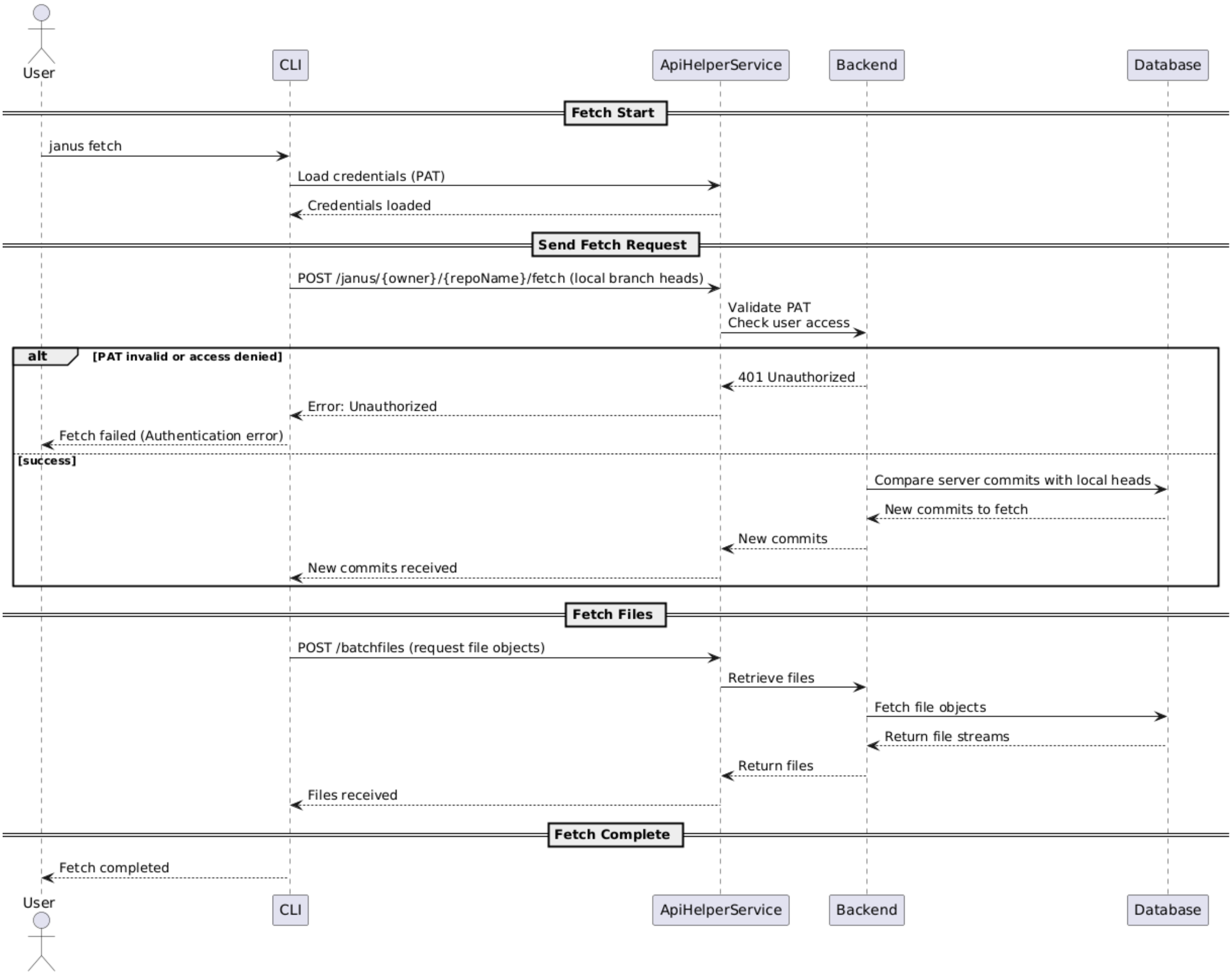


Figure 13.1: Fetch Sequence Diagram

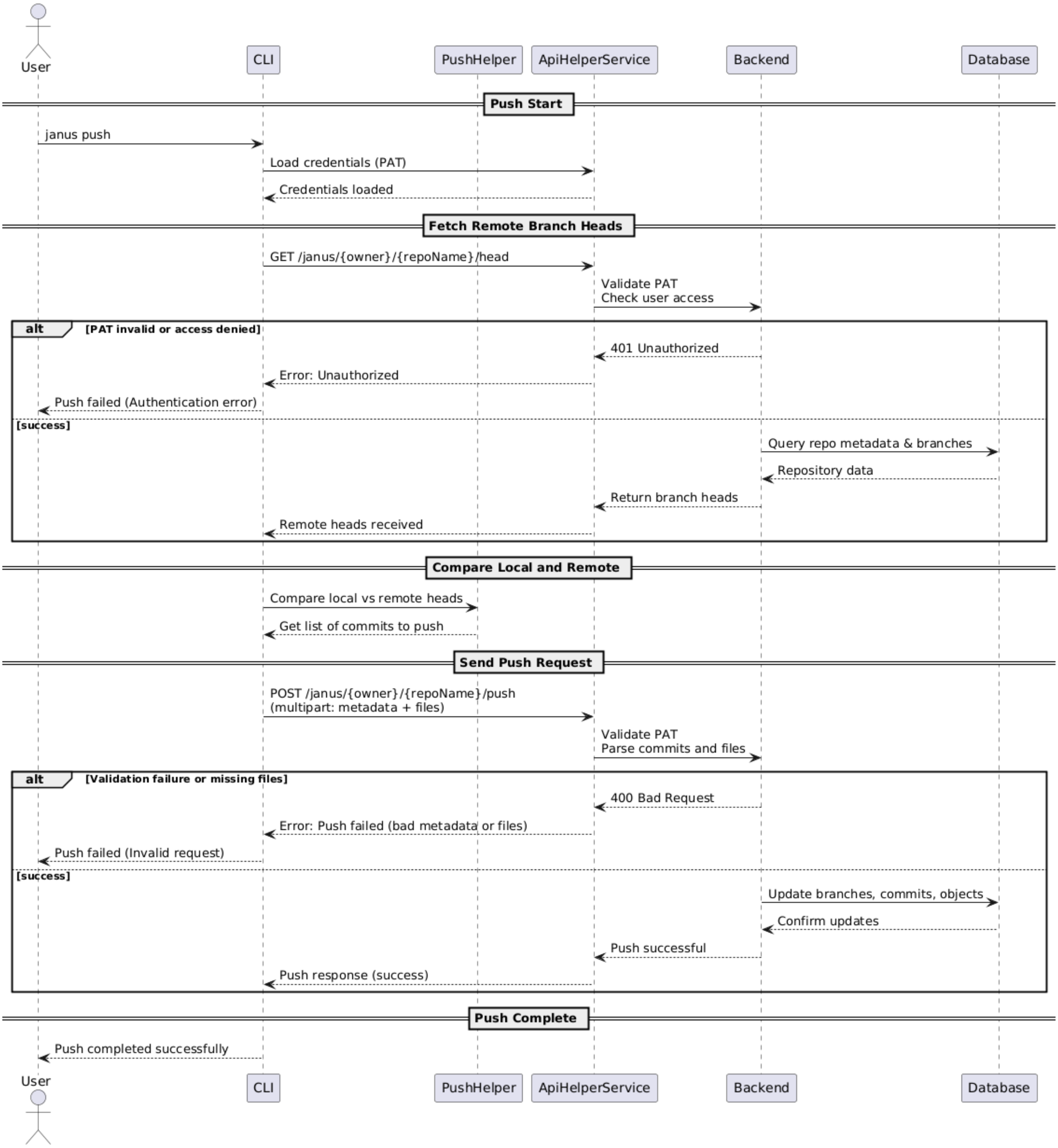


Figure 13.2: Push Sequence Diagram

Appendix E - Project Management

User Stories

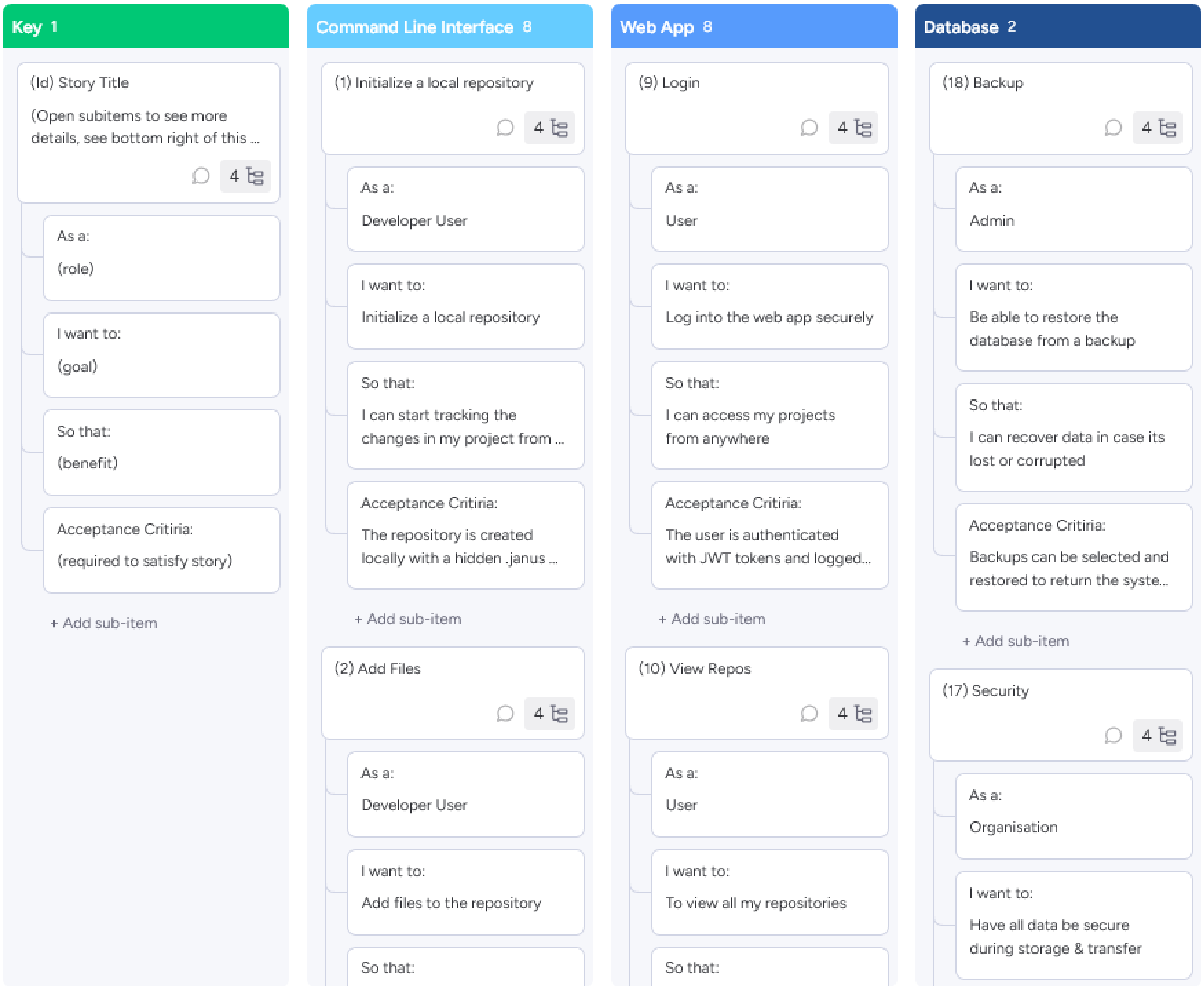


Figure 13.3: User Stories Screenshot

Backlog

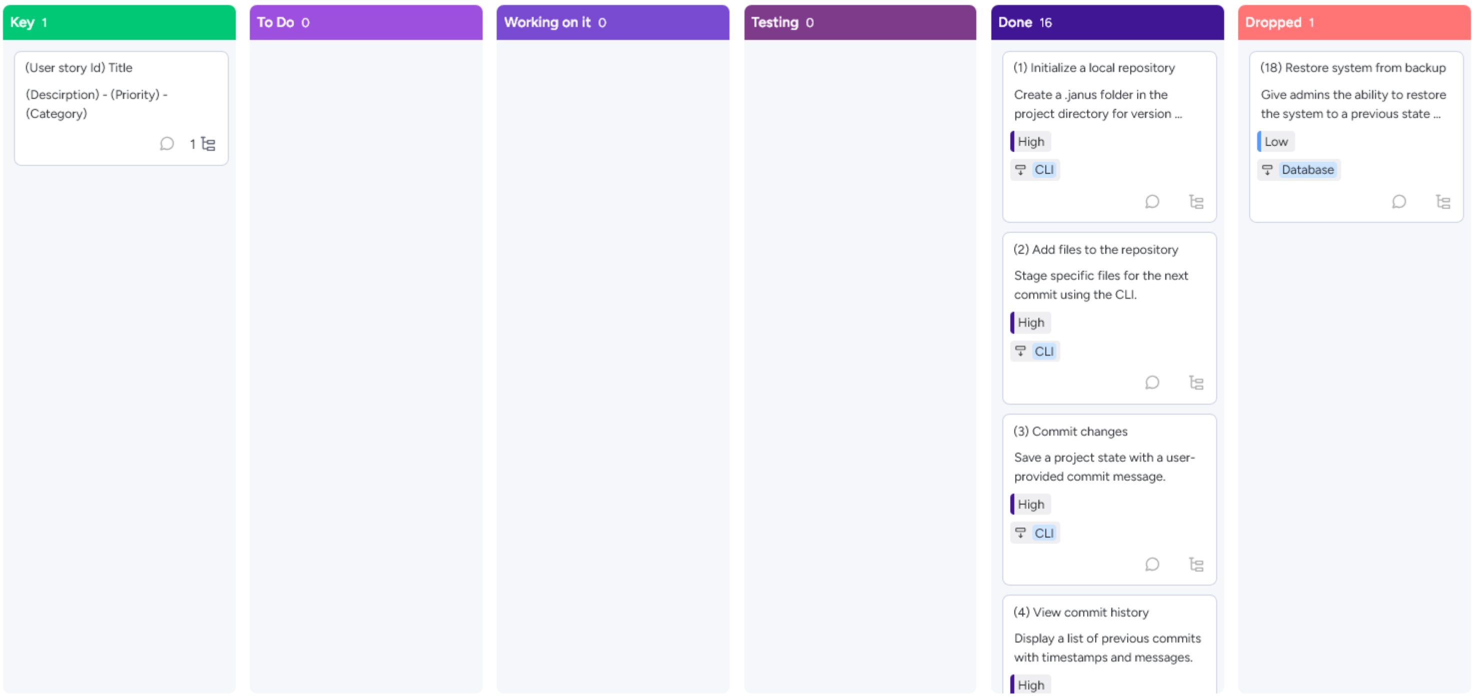


Figure 13.4: Project Backlog Screenshot

Sprints

Link to all sprints:

https://view.monday.com/1651099335-03ff606c28ae82fa4067fa3fc5464da1?r=euc1

Sprint 0:

Sprint 0

Item	Status	Timeline	Long text
Research 3	Done	Sep 23, '24 - Oct 20, '24	

Subitem	Status	Description	Involved	Likelihood	Impact	Mitigation
Existing version control systems	Done	Reseach how each existing version control syst...		Medium	High	Document findings and trade offs and validate ...
Databases for storage	Done	Research possibilitys of how repositories can b...	Database	Low	Medium	Test many database options.
Diff algorithms	Done	Research algorithms and libraries of how to pro...	Backend, Command Line Interface	Medium	Medium	Test existing libraries and evaluate their trade o...
+ Add subitem						

Item	Status	Timeline	Long text
Planning 4	Done	Sep 23, '24 - Oct 20, '24	

Subitem	Status	Description	Involved	Likelihood	Impact	Mitigation
MVP CLI commands	Done	Research and deciede what commands would ...	Command Line Interface	Medium	High	Define clear acceptance criteria and validate wi...
MVP frontend	Done	Research and decide on what functionality the ...	Frontend	Medium	Medium	Define clear acceptance criteria and validate wi...
Communication between system parts	Done	Research and decide on the best method to en...	Backend, Databa..., Comm..., Fronte...	High	Critical	Standadise REST API and mock endpoints early.
Entities for the database	Done	Plan out what needs to be stored and how they...	Database	Medium	High	Use lucidcharts to design ERD
+ Add subitem						

Item	Status	Timeline	Long text
Documentation 1	Done	Sep 23, '24 - Oct 20, '24	

Subitem	Status	Description	Involved	Likelihood	Impact	Mitigation
Write project initiation	Done	Write the project initiation document for submi...	Documentation			
+ Add subitem						

Item	Status	Timeline	Long text
Diagrams 1	Done	Sep 23, '24 - Oct 20, '24	

Subitem	Status	Description	Involved	Likelihood	Impact	Mitigation
System component architecture	Done	Create a system component architecure for ho...	Fronte..., Backend, Comm..., Databa...	Low	Medium	Validate with supervisor
+ Add subitem						

Item	Status	Timeline	Long text
Implementation 4	Done	Sep 23, '24 - Oct 20, '24	

Subitem	Status	Description	Involved	Likelihood	Impact	Mitigation
Setup Command line interface	Done	Create simple console app	Command Line Interface	Low	Medium	
Setup Backend	Done	Create a simple backend	Backend	Low	Low	Use docker and setup build script
Setup Database	Done	Create a simple database	Database	Low	Low	Use docker and setup build script
Setup Frontend	Done	Create a simple frontend	Frontend	Low	Low	Use docker and setup build script
+ Add subitem						

+ Add item

Figure 13.5: Sprint 0

Sprint 1:

Sprint 1

Item	Status	Timeline	Long text
Implementation 8	Done	Oct 21, '24 - Nov 3, '24	

Subitem	Status	Description	Involved	Likelihood	Impact	Mitigation
Implement base command interface	Done	Create a interface which each command can us...	Command Line Interface	High	Critical	Modular design with thorough testing.
Add command	Done	Create an add command which adds files, to a ...	Command Line Interface	Medium	Medium	Test edge cases. (large files, deleted folders)
Commit command	Done	Create a commit command which takes the file...	Command Line Interface	High	Critical	Validate metadata being stored with thorough t...
Log command	Done	Create a command which displays the commit ...	Command Line Interface	Low	Medium	Thorough testing
Banch command	Done	Create a command which can handle branch co...	Command Line Interface	Medium	High	Test branching early and edge cases (empty, no...
Prototype endpoints	Done	Setup basic endpoints for the backend, to crea...	Backend	Medium	Medium	Document API, Mock responses for frontend & ...
Setup testing for CLI	Done	Create tests for the CLI commands.	Command Line Interface	Low	High	Automate tests using GitHub Actions.
Prototype database tables	Done	Setup basic tables for the database. Including p...	Database	Low	Low	Use migrations with Entity Framework Core, giv...
+ Add subitem						

Item	Status	Timeline
Plugin 3	Done	Oct 21, '24 - Nov 3, '24

Subitem	Status	Description	Involved	Likelihood	Impact	Mitigation
Implement use of plugins	Done	Utalising the command interface allow for the ...	Command Line Interface	High	High	Use the same interface as base commands.
Create Demo	Done	Create a demo to show how you can create plu...	Command Line Interface	Low	Low	Provide clear examples which can be used in do...
Write documentation for creating plugins	Done	Document the intrecies of how you would creat...	Command Line Interface	Low	Medium	Peer review document.
+ Add subitem						

Item	Status	Timeline
Diff (find delta) 1	Done	Oct 21, '24 - Nov 3, '24

Subitem	Status	Description	Involved	Likelihood	Impact	Mitigation
Implement diff algorithm for comparing files	Done	Create a prototype diff algorithm for comparin...	Command Line Interface	Medium	Medium	Test with files. Benchmark performance, and ha...
+ Add subitem						

Item	Status	Timeline
Docker 3	Done	Oct 21, '24 - Nov 3, '24

Subitem	Status	Description	Involved	Likelihood	Impact	Mitigation
Put backend into container	Done	Dockerise the backend to run inside a container...	Backend	Medium	High	Use docker compose
Put database into container	Done	Dockerise the database to run inside a containe...	Database	Medium	High	Use docker compose
Put frontend into container	Done	Dockerise the frontend to run inside a containe...	Frontend	Medium	High	Use docker compose
+ Add subitem						

Item	Status	Timeline
+ Add item		
		Oct 21, '24 - Nov 3, '24

Figure 13.6: Sprint 1

Sprint 2:

Sprint 2

Item	Status	Timeline	Long text
Implementation 7	Done	Nov 4, '24 - Dec 1, '24	

Subitem	Status	Description	Involved	Likelihood	Impact	Mitigation
Setup CORS policy	Done	Setup the cors policy to allow only desired com...	Backend, Frontend, Command Line Inte...	Medium	Medium	Whitelist trusted domains, test with other dom...
Register page	Done	Create register page components.	Frontend	Low	Medium	Follow WCAG guidelines and test with screen r...
Login page	Done	Create login page components	Frontend	Low	Medium	Follow WCAG guidelines and test with screen r...
Register logic creating user	Done	Implement functionality for register page form ...	Frontend, Backend, Database	High	Critical	Enforce rate limiting
Login page logic	Done	Implement functionality for login page form to ...	Frontend, Backend, Database	High	Critical	Enforce rate limiting
Password salting & hashing	Done	Store the password as its salt and hash.	Backend, Database	Medium	Critical	Follow OWASP guidelines
CLI login functionality	Done	Implement login functionality for the command...	Command Line Int..., Backend, Database	High	Critical	Enforce rate limiting, ensure PAT is revokable.
+ Add subitem						

Item	Status	Timeline
Diagrams 2	Done	Nov 4, '24 - Dec 1, '24

Subitem	Status	Description	Involved	Likelihood	Impact	Mitigation
Database Entity Relationship Diagram	Done	Create an entity relationship diagram for the pr...	Database			
Database UML Diagrams	Done	Create UML diagrams for the proposed final da...	Database			
+ Add subitem						

Item	Status	Timeline
Usability 4	Done	Nov 4, '24 - Dec 1, '24

Subitem	Status	Description	Involved	Likelihood	Impact	Mitigation
Usability testing on wireframes	Done	Perform usability testing on the low fidelity desi...	Frontend	Low	Low	Conduct A/B testing, iterate based on feedback.
Low fidelity wireframes	Done	Create multiple differing low fidelity designes f...	Frontend	Low	Low	Conduct A/B testing, iterate based on feedback.
Usability testing on designs	Done	Perform usability testing on the high fidelity de...	Frontend	Low	Low	Conduct A/B testing, iterate based on feedback.
High fidelity mockup designs	Done	Create multiple differing high fidelity designes ...	Frontend	Low	Medium	Conduct A/B testing, iterate based on feedback.
+ Add subitem						

Item	Status	Timeline
+ Add item		
		Nov 4, '24 - Dec 1, '24

Figure 13.7: Sprint 2

Sprint 3:

Sprint 3

Item	Status	Timeline	Long text
Implementation 6	Done	Nov 18, '24 - Dec 8, '24	

Subitem	Status	Description	Involved	Likelihood	Impact	Mitigation
Command testing	Done	Setup tests for each of the CLI commands, to e...	Command Line Interface			
Create self signed certificates	Done	Create self signed certificates using openssl for...	Backend, Frontend	Low	Medium	Document certificates (how to replace with rea...
Setup HTTPS	Done	Enable HTTPS for cummunication between diff...	Backend, Frontend	High	Critical	Enforce TLS 1.3, use HSTS headers.
Handle token generation	Done	Create methods to generate tokens, which will ...	Backend	Medium	High	Ensure expiry.
Handle PAT generation	Done	Create methods to generate personal access to...	Backend	Medium	High	Implement token revocation and ensure expiry.
Setup token authentication	Done	Rework the backend login endpoints to return t...	Backend, Command Line Inte..., Frontend	High	Critical	Test only valid tokens are accepted. Handle inv...
+ Add subitem						

+ Add item

	Nov 18, '24 - Dec 8, '24

Figure 13.8: Sprint 3

Sprint 4:

Sprint 4

Item	Status	Timeline	Long text
Implementation 8	Done	Dec 2, '24 - Dec 22, '24	

Subitem	Status	Description	Involved	Likelihood	Impact	Mitigation
Github Actions Workflow	Done	Create a workflow to allow tests to run automa...	Command Line Interface			
Cross platform CLI	Done	Update workflow to run tests on windows, linux...	Command Line Interface	High	High	Test on windows, macOS and Linux, automate ...
Create-Branch command	Done	Implement the create branch commmand whic...	Command Line Interface	Medium	Medium	Test edge cases.
Create-Branch command tests	Done	Create tests to ensure create-branch functions...	Command Line Interface			
Advance init command	Done	Improve the commit command to gracefully ha...	Command Line Interface	Low	Low	Ensure previous tests are still passed.
Advance add command	Done	Improve the add command to gracefully handle...	Command Line Interface	Low	Low	Ensure previous tests are still passed.
Advance commit command	Done	Improve the commit command to gracefully ha...	Command Line Interface	Low	Low	Ensure previous tests are still passed.
Create Home page	Done	Create a frontend home page. With the ability t...	Frontend	Low	Low	Test that navigation to protected routes are onl...
+ Add subitem						

Item	Status	Timeline	Long text
Stylng 4	Done	Dec 2, '24 - Dec 22, '24	

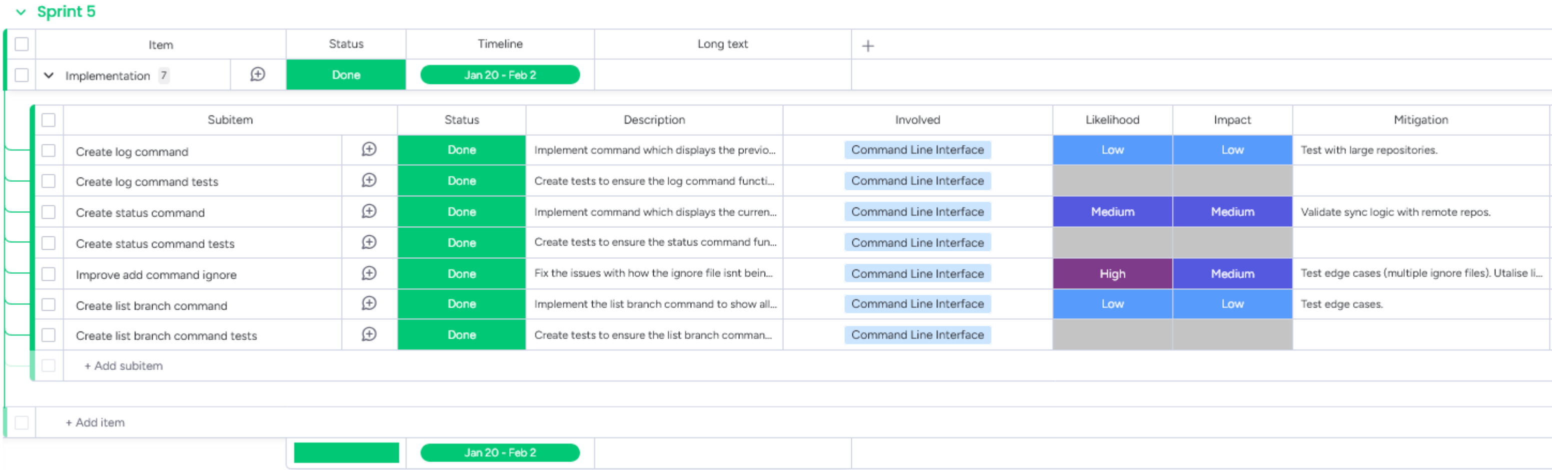
Subitem	Status	Description	Involved	Likelihood	Impact	Mitigation
Style login page	Done	Style the login page with clear and simple desig...	Frontend			
Resgister page	Done	Style the register page with clear and simple de...	Frontend			
Home page	Done	Style the home page	Frontend			
Light & Dark theme	Done	Use css variables with a light theme and dark th...	Frontend	Low	Low	Use CSS variables for themes (allows for easy c...
+ Add subitem						

+ Add item

	Dec 2, '24 - Dec 22, '24

Figure 13.9: Sprint 4

Sprint 5:



Sprint 5

Item	Status	Timeline	Long text
Implementation 7	Done	Jan 20 - Feb 2	

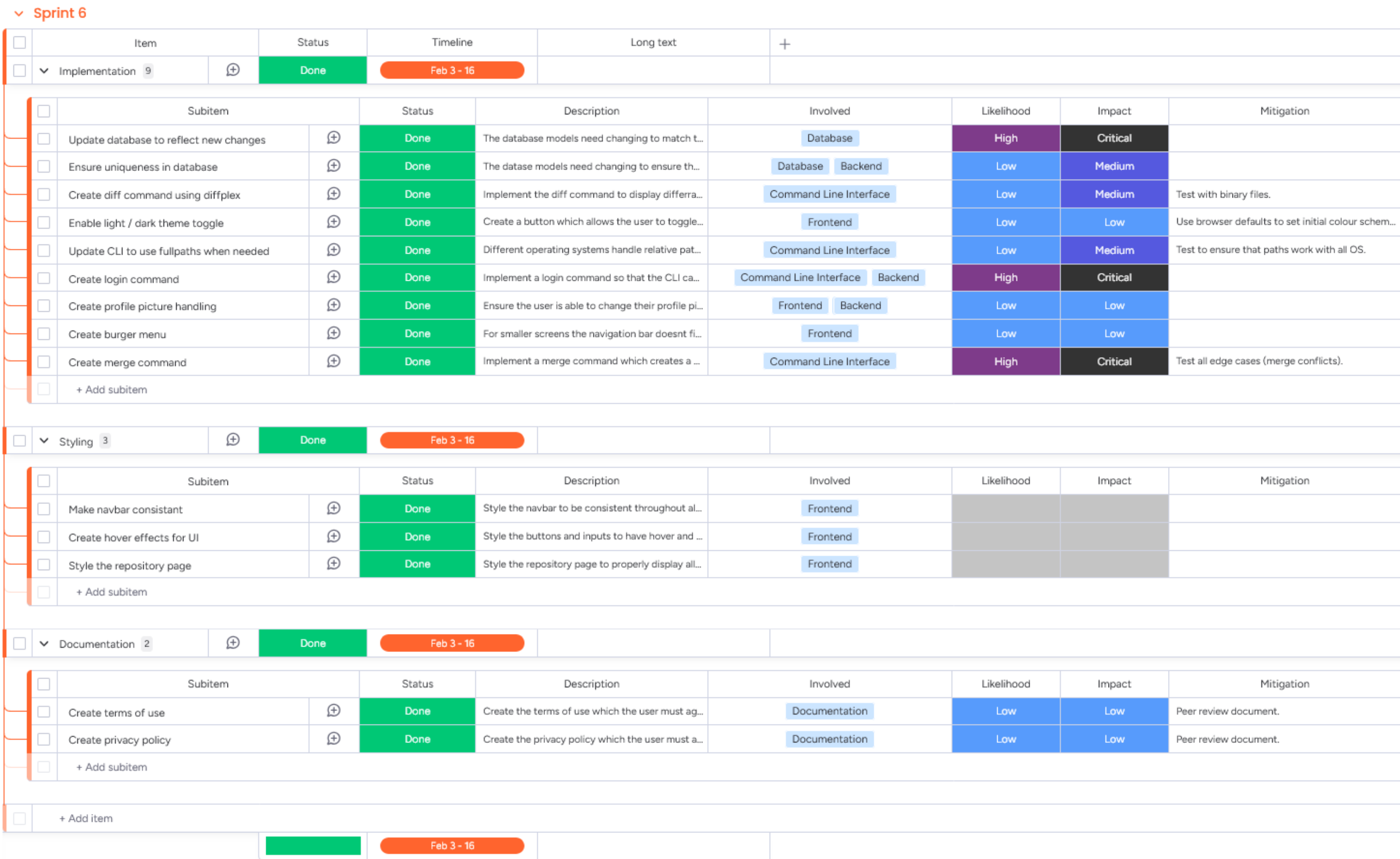
Subitem	Status	Description	Involved	Likelihood	Impact	Mitigation
Create log command	Done	Implement command which displays the previo...	Command Line Interface	Low	Low	Test with large repositories.
Create log command tests	Done	Create tests to ensure the log command functi...	Command Line Interface			
Create status command	Done	Implement command which displays the curren...	Command Line Interface	Medium	Medium	Validate sync logic with remote repos.
Create status command tests	Done	Create tests to ensure the status command fun...	Command Line Interface			
Improve add command ignore	Done	Fix the issues with how the ignore file isnt bein...	Command Line Interface	High	Medium	Test edge cases (multiple ignore files). Utalise li...
Create list branch command	Done	Implement the list branch command to show all...	Command Line Interface	Low	Low	Test edge cases.
Create list branch command tests	Done	Create tests to ensure the list branch comman...	Command Line Interface			
+ Add subitem						

+ Add item

	Status	Timeline	
		Jan 20 - Feb 2	

Figure 13.10: Sprint 5

Sprint 6:



Sprint 6

Item	Status	Timeline	Long text
Implementation 9	Done	Feb 3 - 16	

Subitem	Status	Description	Involved	Likelihood	Impact	Mitigation
Update database to reflect new changes	Done	The database models need changing to match t...	Database	High	Critical	
Ensure uniqueness in database	Done	The datase models need changing to ensure th...	Database, Backend	Low	Medium	
Create diff command using diffplex	Done	Implement the diff command to display differra...	Command Line Interface	Low	Medium	Test with binary files.
Enable light / dark theme toggle	Done	Create a button which allows the user to toggle...	Frontend	Low	Low	Use browser defaults to set initial colour schem...
Update CLI to use fullpaths when needed	Done	Different operating systems handle relative pat...	Command Line Interface	Low	Medium	Test to ensure that paths work with all OS.
Create login command	Done	Implement a login command so that the CLI ca...	Command Line Interface, Backend	High	Critical	
Create profile picture handling	Done	Ensure the user is able to change their profile pi...	Frontend, Backend	Low	Low	
Create burger menu	Done	For smaller screens the navigation bar doesnt fi...	Frontend	Low	Low	
Create merge command	Done	Implement a merge command which creates a ...	Command Line Interface	High	Critical	Test all edge cases (merge conflicts).
+ Add subitem						

Item	Status	Timeline	
Styling 3	Done	Feb 3 - 16	

Subitem	Status	Description	Involved	Likelihood	Impact	Mitigation
Make navbar consistant	Done	Style the navbar to be consistent throughout al...	Frontend			
Create hover effects for UI	Done	Style the buttons and inputs to have hover and ...	Frontend			
Style the repository page	Done	Style the repository page to properly display all...	Frontend			
+ Add subitem						

Item	Status	Timeline	
Documentation 2	Done	Feb 3 - 16	

Subitem	Status	Description	Involved	Likelihood	Impact	Mitigation
Create terms of use	Done	Create the terms of use which the user must ag...	Documentation	Low	Low	Peer review document.
Create privacy policy	Done	Create the privacy policy which the user must a...	Documentation	Low	Low	Peer review document.
+ Add subitem						

+ Add item

	Status	Timeline	
		Feb 3 - 16	

Figure 13.11:Sprint 6

Sprint 7:

Sprint 7

Item	Status	Timeline	Long text
Implementation 10	Done	Feb 17 - Mar 16	

Subitem	Status	Description	Involved	Likelihood	Impact	Mitigation
Create clone command	Done	Implement the clone command to recreate the ...	Command Line Interface, Backend	High	High	Mock with expected results first and then conn...
Create discover page	Done	Create a discover page which shows all public r...	Backend, Database	Low	Low	
Update file explorer to use breadcrumbs	Done	Implement breadcrumbs for repository file expl...	Frontend	Low	Low	Test with deep file structures.
Create get all user repos frontend endpoint	Done	Implement an endpoint which returns all the re...	Backend, Database	High	High	Document endpoint and ensure it follows stand...
Create get repo frontend endpoint	Done	Implement an endpoint which returns the all m...	Backend, Database	High	High	Document endpoint and ensure it follows stand...
Create get branch frontend endpoint	Done	Implement an endpoint which returns the data ...	Backend, Database	High	High	Document endpoint and ensure it follows stand...
Update searching to use debounce	Done	Implement debouncing to stop frequent unnec...	Backend	Low	Low	
+ Add subitem						

Item	Status	Timeline	Long text
Documentation 2	Done	Feb 17 - Mar 16	

Subitem	Status	Description	Involved	Likelihood	Impact	Mitigation
Start work on poster	Done	Start creating the design of the poster	Documentation			
Start work on description	Done	Write the description which goes with the poster	Documentation			
+ Add subitem						

+ Add item

Feb 17 - Mar 16

Figure 13.12: Sprint 7

Sprint 8:

Sprint 8

Item	Status	Timeline	Long text
Implementation 9	Done	Mar 3 - 16	

Subitem	Status	Description	Involved	Likelihood	Impact	Mitigation
Ensure user data can be deleted safely	Done	Allow for users to delete account data and repo...	Frontend, Backend, Database	High	Critical	Test cascading deletion, conplies with GDPR.
Create commit page	Done	Implement the commit page which shows the c...	Frontend	Low	Low	
Create commit frontend endpoint	Done	Implement the commit endpoint which takes in...	Backend, Database	High	Critical	Document endpoint and ensure it follows stand...
Create contributors frontend endpoint	Done	Implement the contributors endpoint which tak...	Backend, Database	High	Critical	Document endpoint and ensure it follows stand...
Create contributors page	Done	Implement the contributors page which display...	Frontend, Backend	Medium	Medium	
Allow setting of PAT expirary time	Done	Update the PAT generation to allow the user to ...	Frontend	Low	High	Ensure token is invalid after expiry time has pas...
Create config command	Done	Implement the config command which allows t...	Command Line Interface	Low	Low	
Create pull command	Done	Implement the pull command which takes the l...	Command Line Interface	High	High	Mock with expected results first and then conn...
Create push command	Done	Implement the push command which sends ne...	Command Line Int..., Backend, Database	High	High	Mock with expected results first and then conn...
+ Add subitem						

Item	Status	Timeline	Long text
Documentation 2	Done	Mar 3 - 16	

Subitem	Status	Description	Involved	Likelihood	Impact	Mitigation
Submit poster	Done	Finish and submit the poster	Documentation			
Submit description	Done	Finish and submit the description	Documentation			
+ Add subitem						

+ Add item

Mar 3 - 16

Figure 13.13: Sprint 8

Sprint 9:

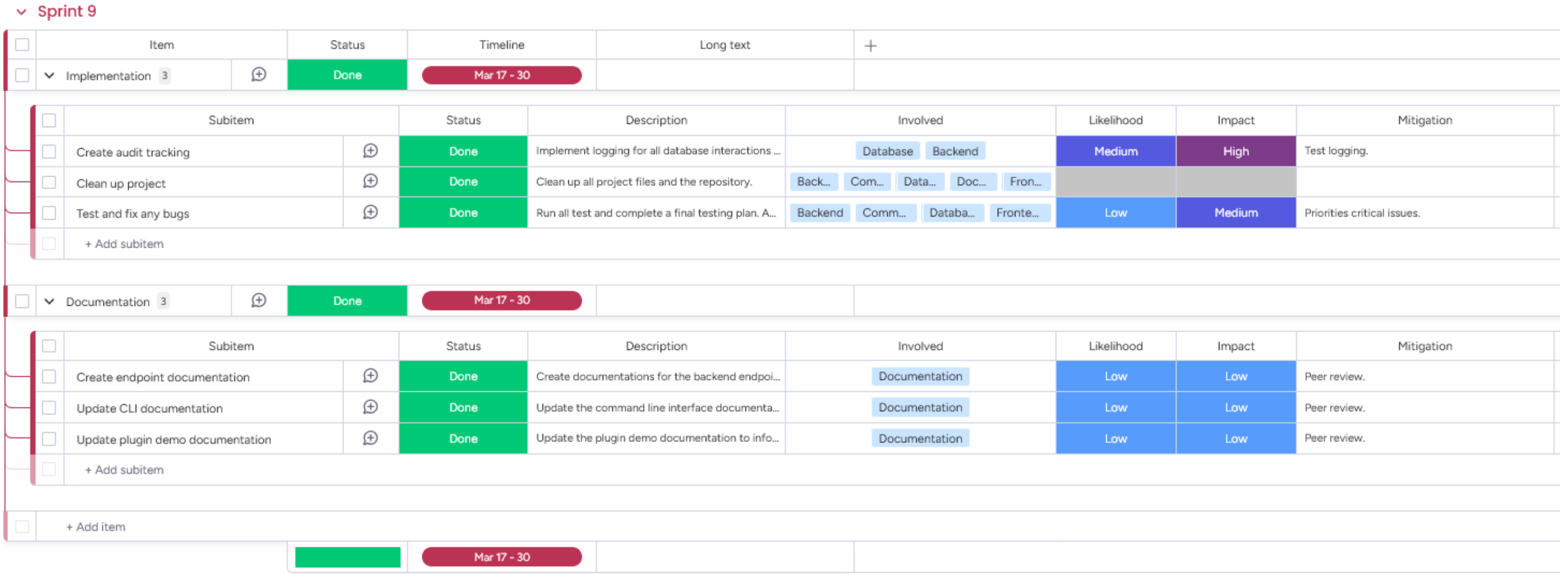


Figure 13.14: Sprint 9

Sprint 10:

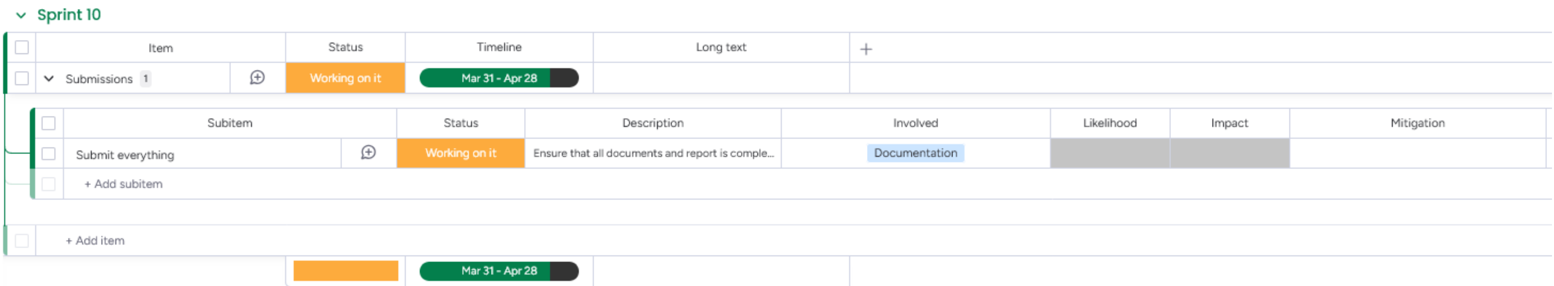


Figure 13.15: Sprint 10

Appendix F – Design Prototypes

All prototypes can be found on Figma:

https://www.figma.com/design/YrT9kQ0z15EGrFZo2Ik6QG/Janus?node-id=11-1836&t=fEYnjTZ9ns0Ml1yp-1

Low-Fidelity

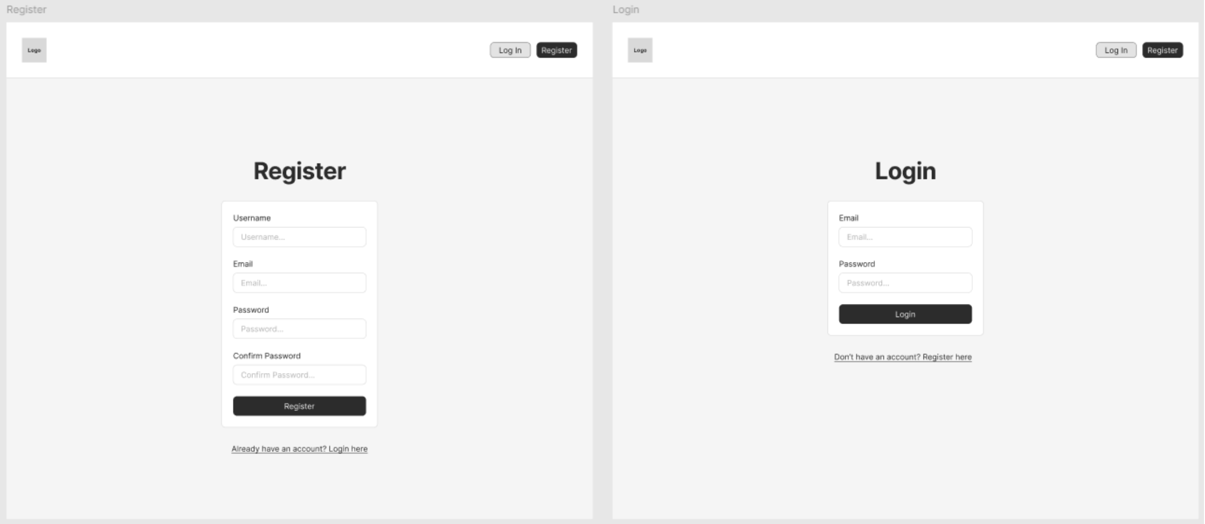


Figure 13.16: Low-Fidelity Login & Register Page Wireframes (Version 1)

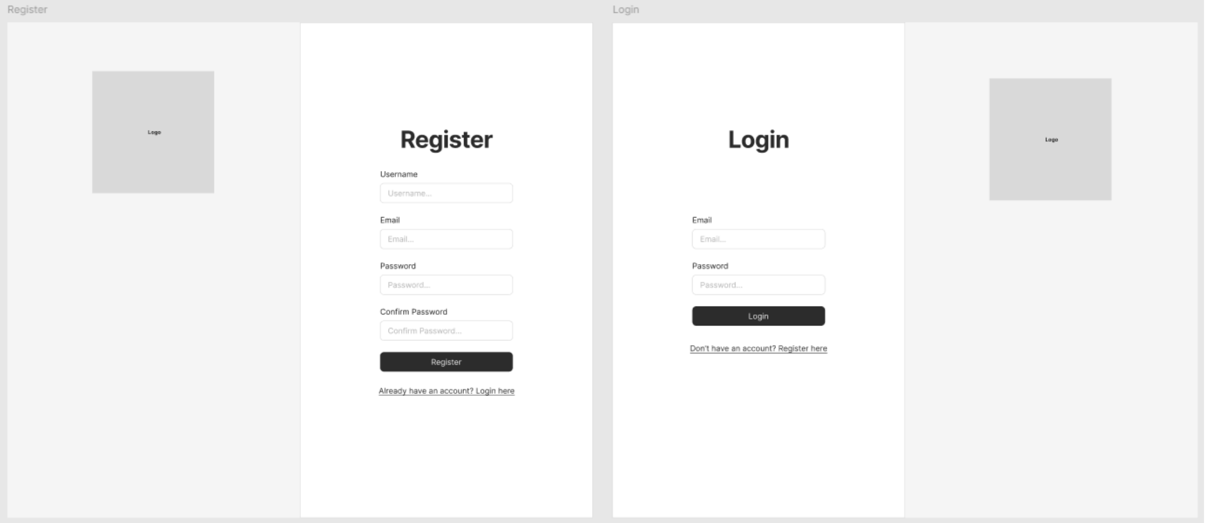


Figure 13.17: Low-Fidelity Login & Register Page Wireframes (Version 2)

High-Fidelity Prototypes

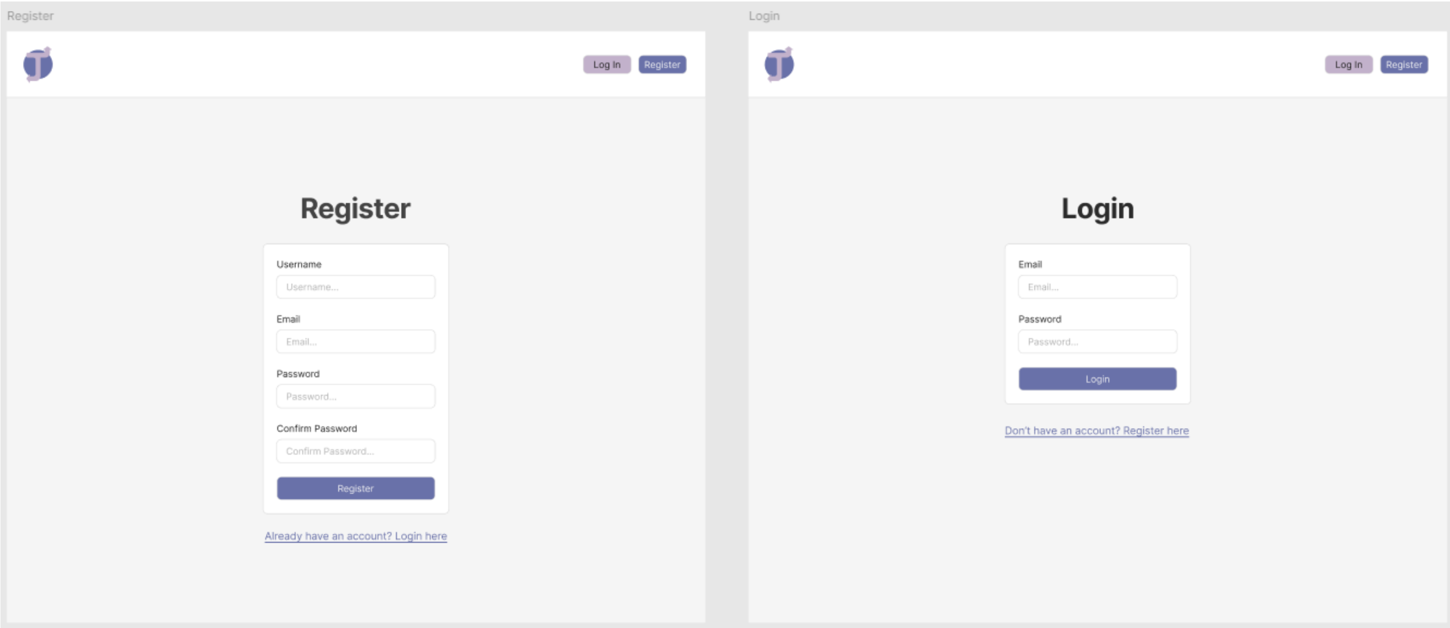


Figure 13.18: High-Fidelity Login & Register Pages Prototype

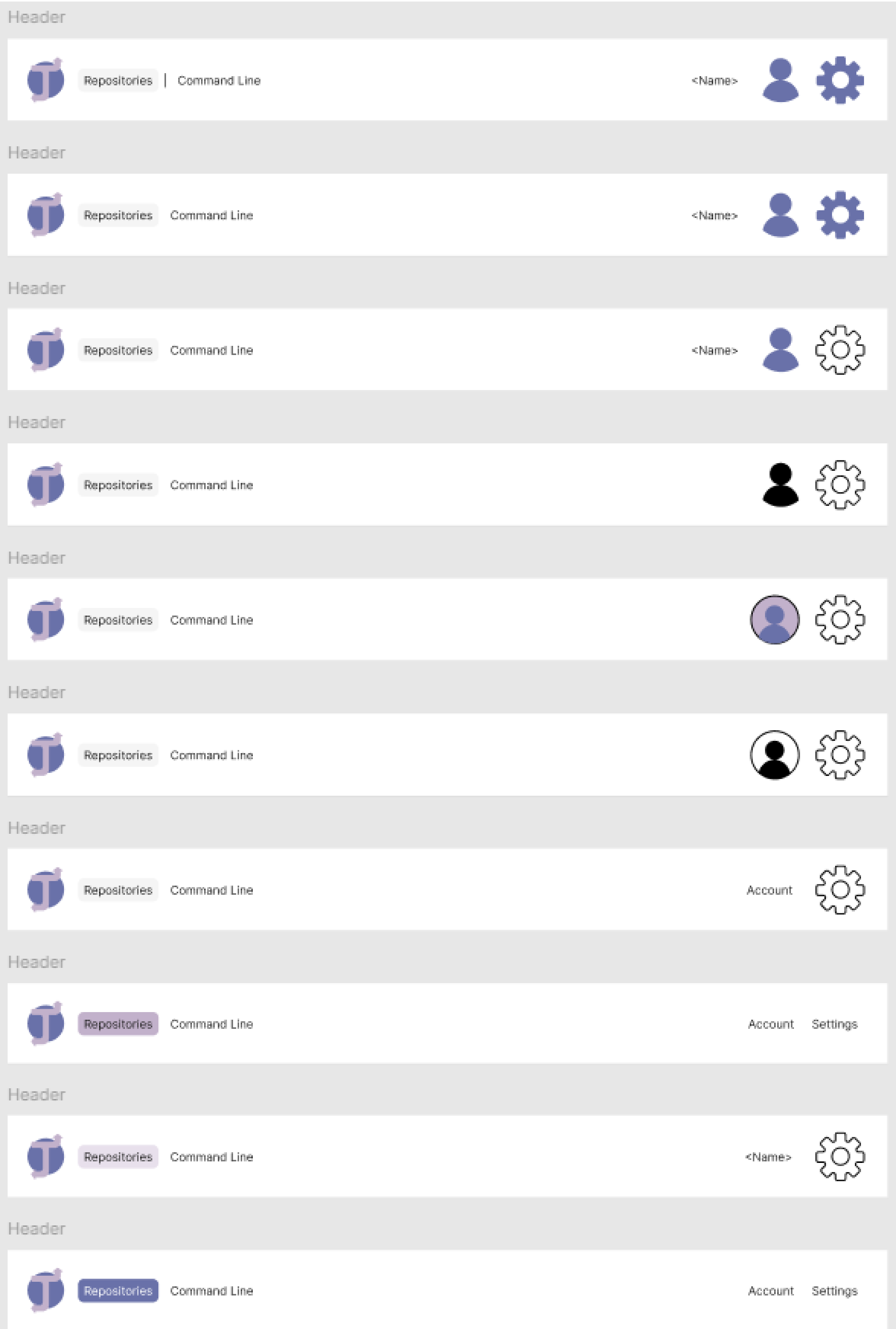


Figure 13.19: High-Fidelity Header Prototypes

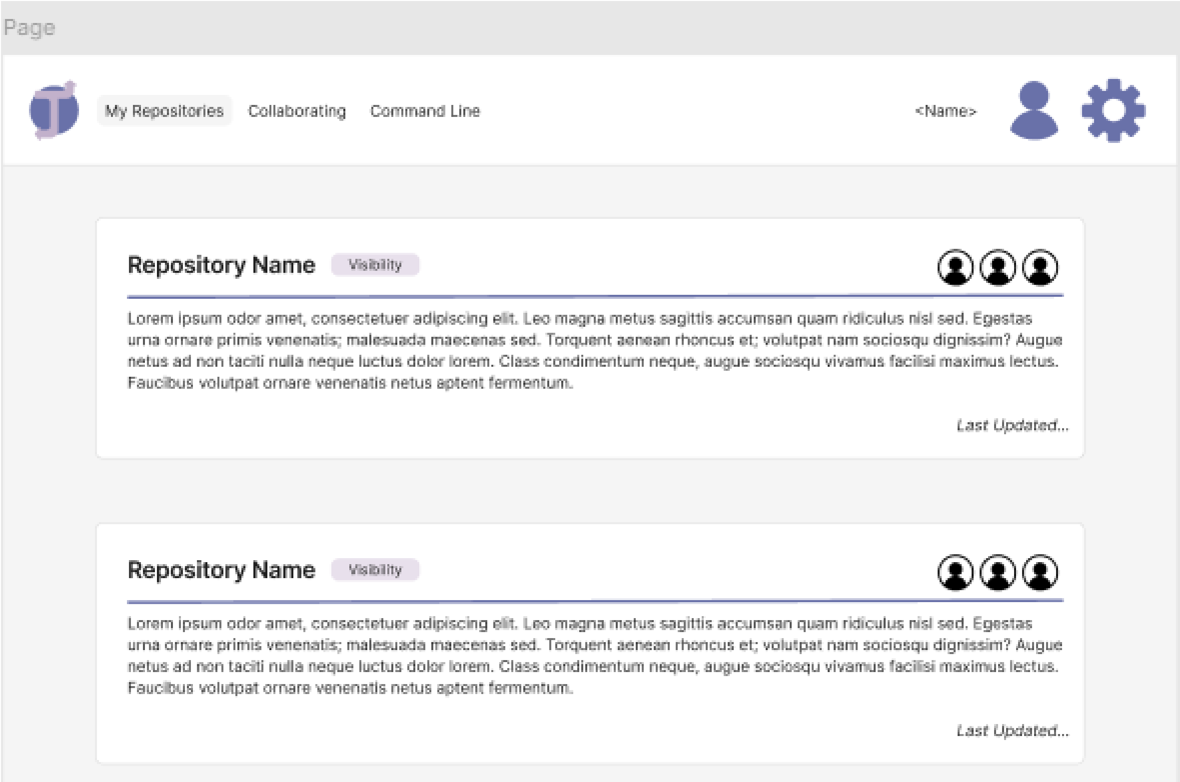


Figure 13.20: High-Fidelity Repositories Page Prototype (Version 1)

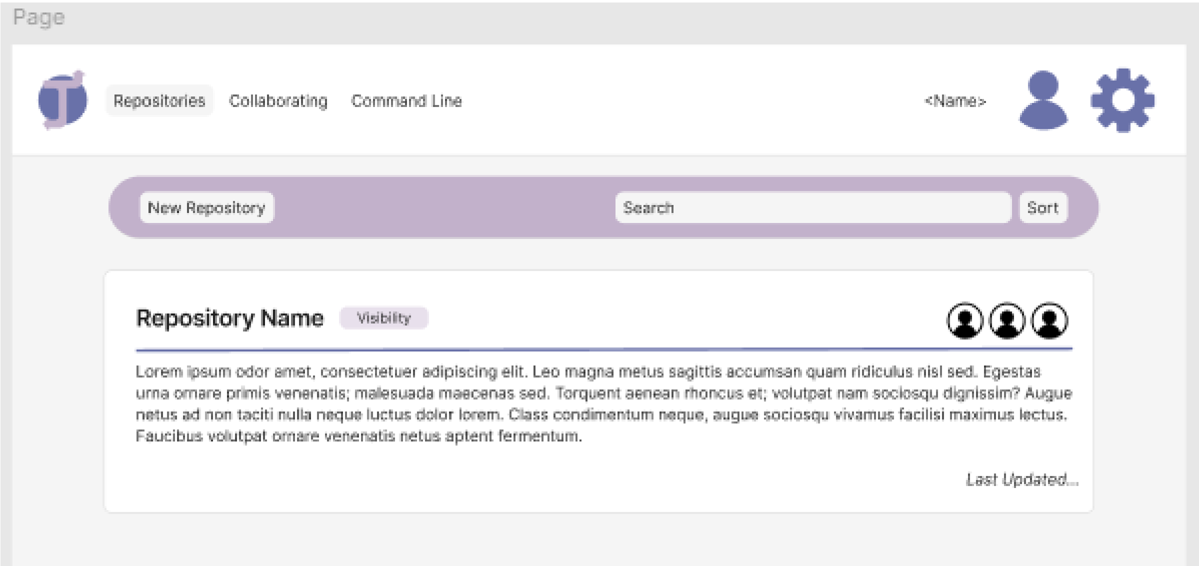


Figure 13.21: High-Fidelity Repositories Page Prototype (Version 2)

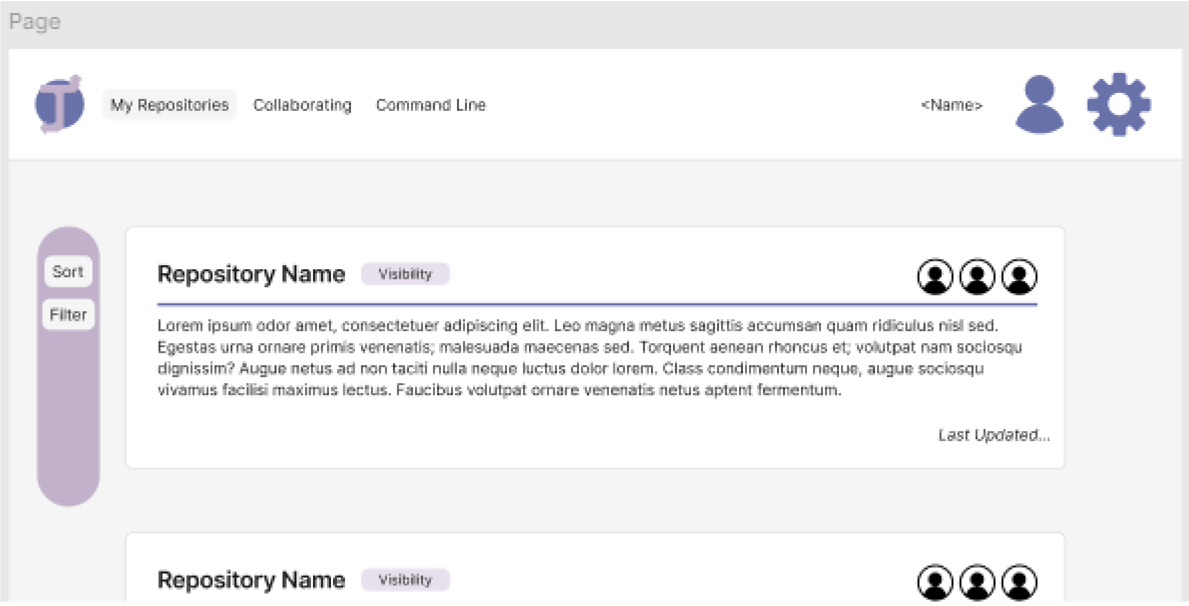


Figure 13.22: High-Fidelity Repositories Page Prototype (Version 3)

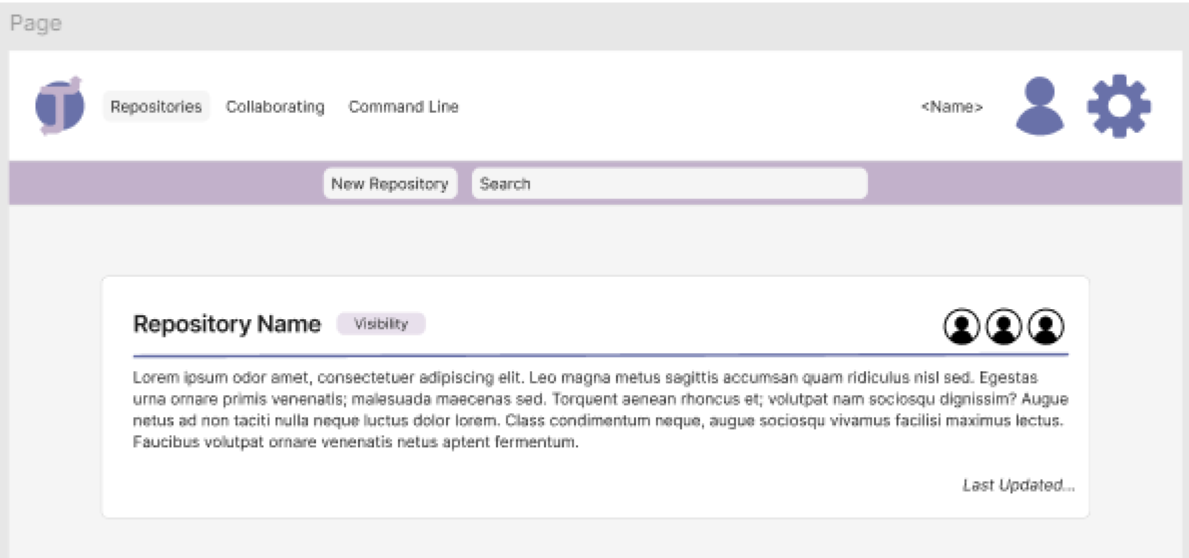


Figure 13.23: High-Fidelity Repositories Page Prototype (Version 4)

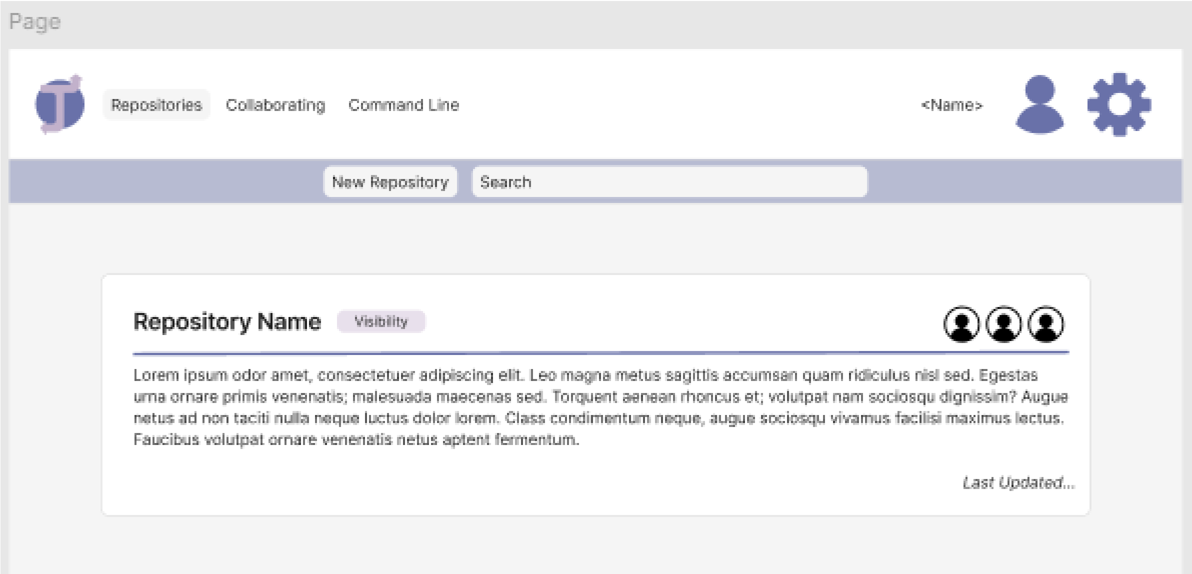


Figure 13.24: High-Fidelity Repositories Page Prototype (Version 5)

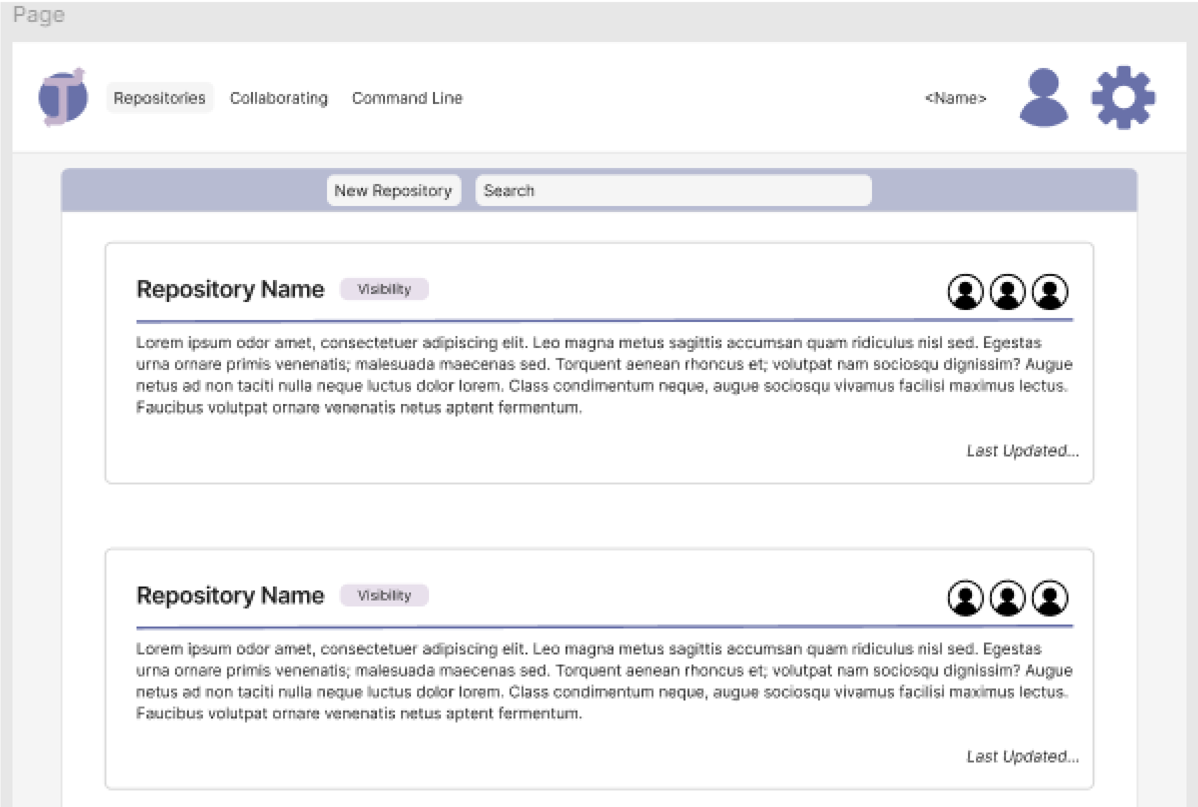


Figure 13.25: High-Fidelity Repository Card Prototype (Version 1)

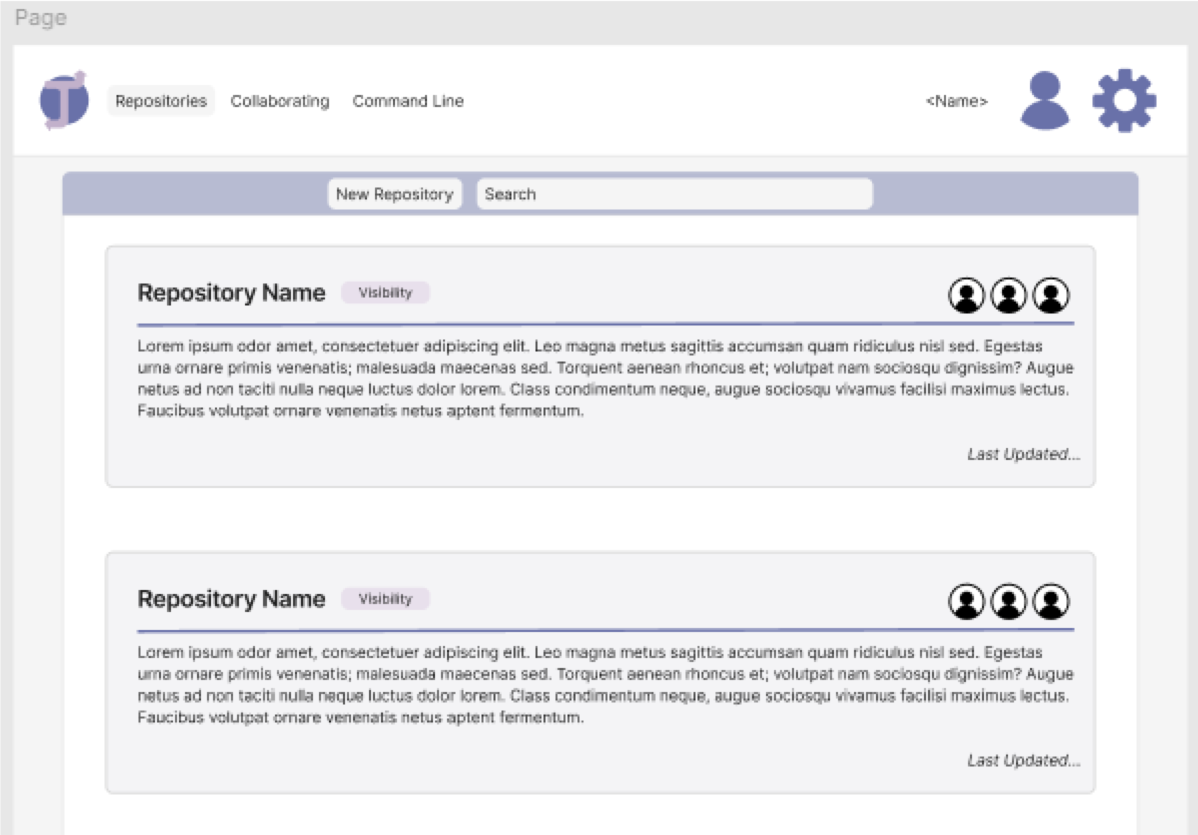


Figure 13.26: High-Fidelity Repository Card Prototype (Version 2)

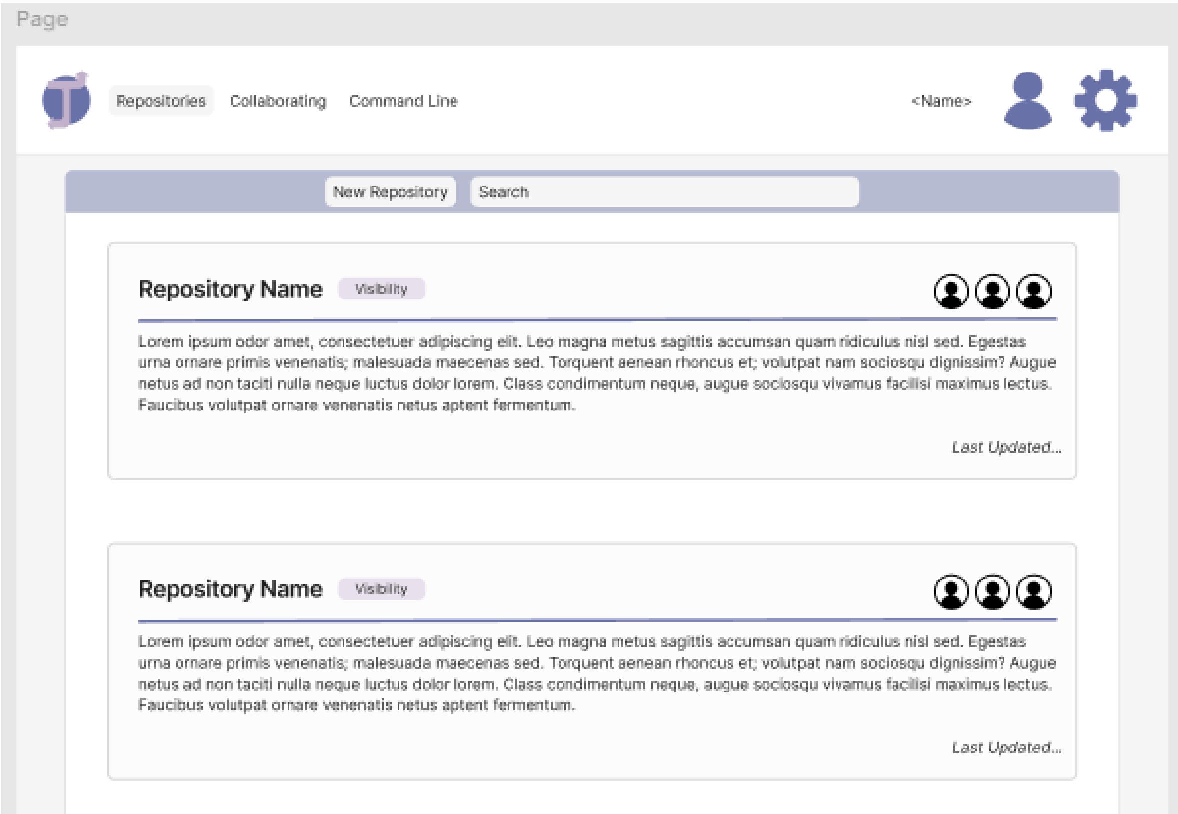


Figure 13.27: High-Fidelity Repository Card Prototype (Version 3)

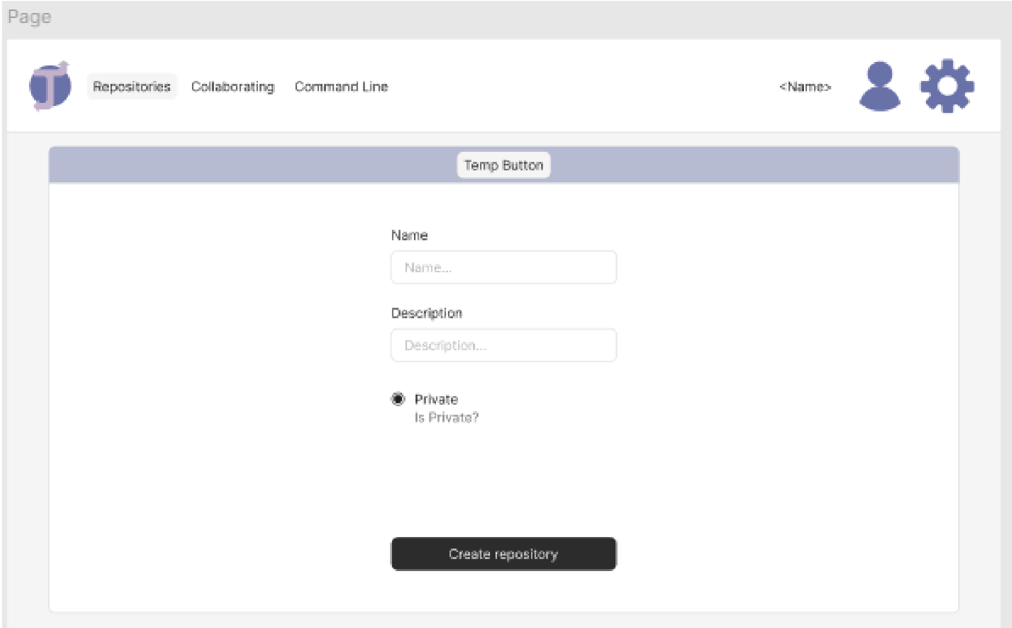


Figure 13.28: High-Fidelity Create Repository Page Prototype

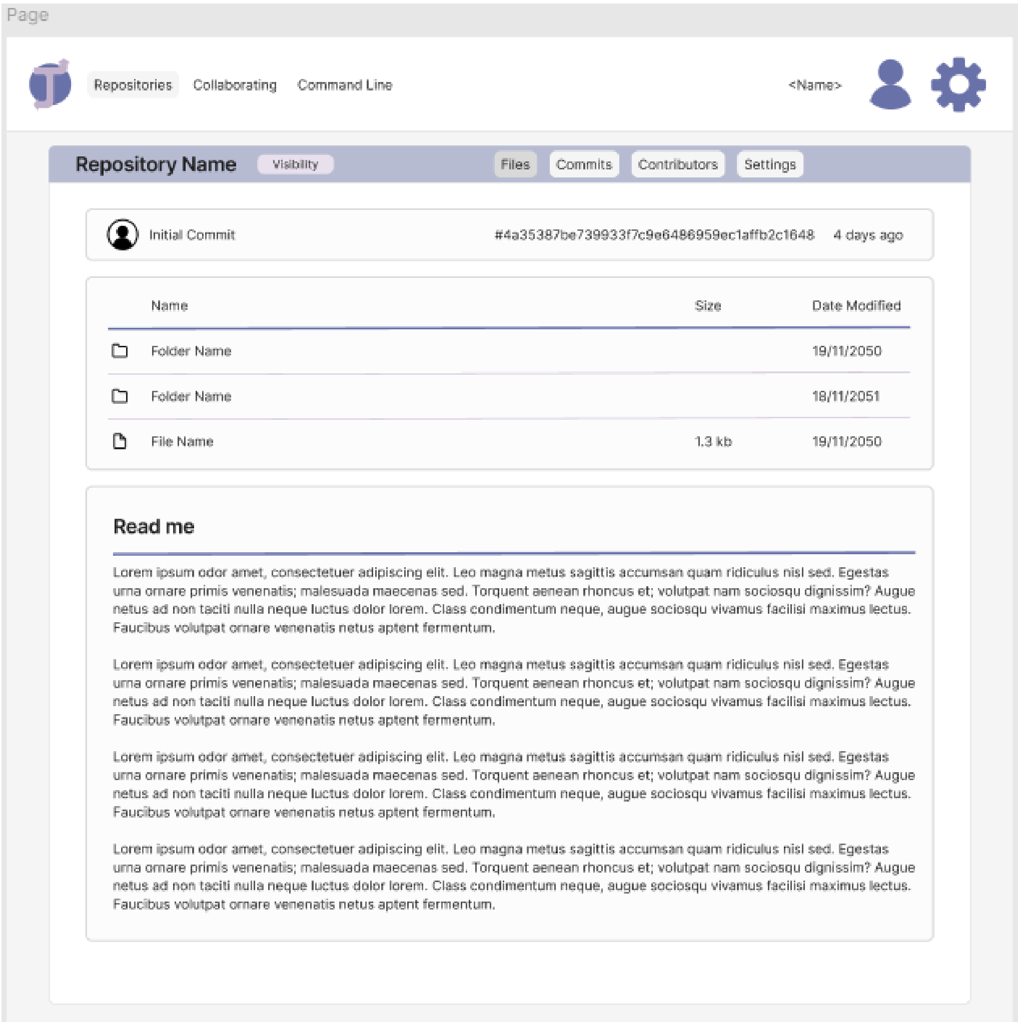


Figure 13.29: High-Fidelity Repository Page Prototype (Version 1)

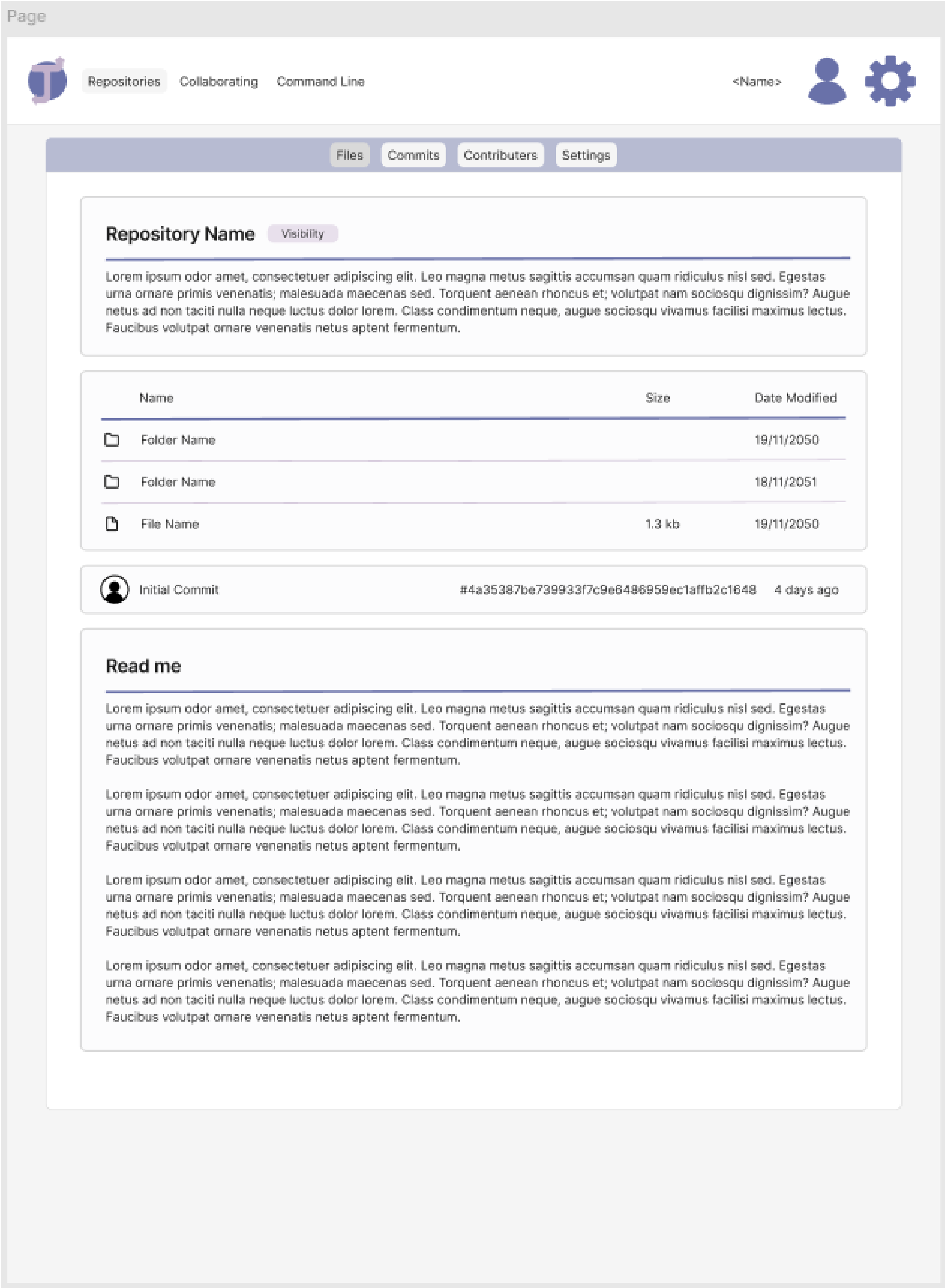


Figure 13.30: High-Fidelity Repository Page Prototype (Version 2)

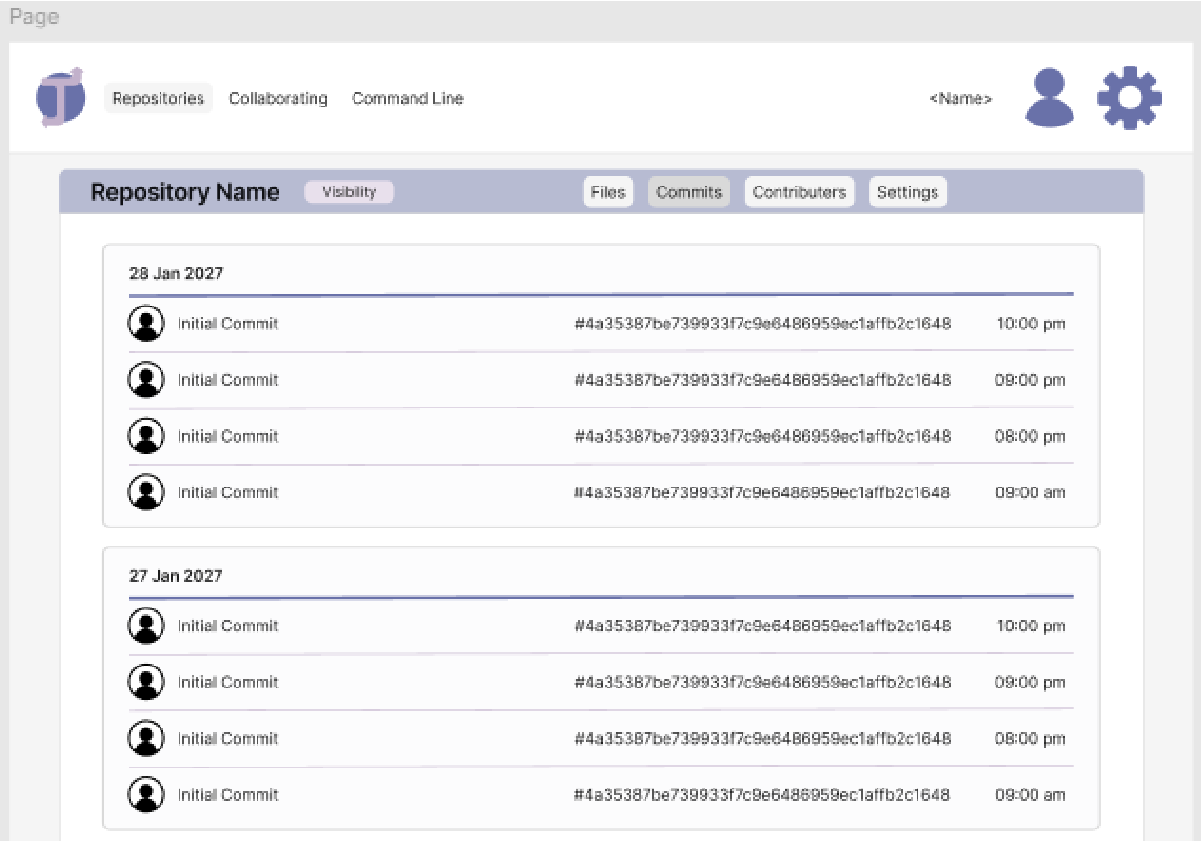


Figure 13.31: High-Fidelity Commit History Page Prototype

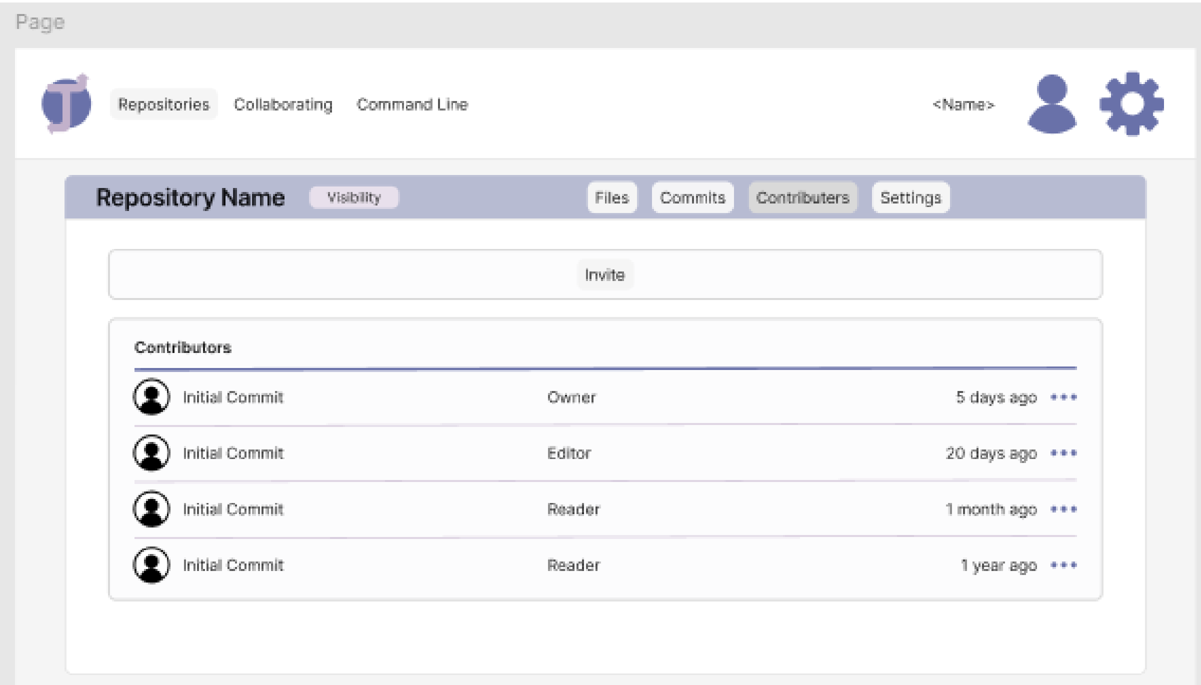


Figure 13.32: High-Fidelity Contributors Page Prototype