**Bulgarian Diploma Thesis**

**TimeTree: A Lightweight Local Version Control System**

**Viktor Logodazhki, ID# 200190541**

**Student:**   **, Date:**

*signature*

**Supervisor:**   **, Date:**

*signature*

**Department of Computer Science, AUBG**

**Blagoevgrad, 2025**

**Declaration template for Senior Project and Diploma Thesis**

**Title**: TimeTree: A Lightweight Local Version Control System

**Author**: Viktor Logodazhki

**Abstract:** Version control systems (VCS) like Git provide tools for managing code changes in collaborative development environments. However, many users find these systems complex, especially in terms of setup, configuration, and usage. This project, TimeTree, aims to address these challenges by implementing a custom, lightweight version control system. Built in Kotlin, TimeTree provides a simple command-line interface (CLI) for performing core VCS tasks such as initialization, file staging, commit creation, and branch management. At the heart of the system is a SHA-1 hashing algorithm and a Myers LCS implementation for diffing, and a rolling checksum algorithm for large files. The application’s features include essential VCS functionalities like commit, status, log, diff, and branching, with an intuitive CLI for user interaction. It also ensures data integrity and consistency through its use of object storage and atomic operations. TimeTree is designed to be extensible, with future improvements including interactive log graphs, commit statistics, and supporting more hashing algorithms. This project not only provides a simple, reliable local VCS but also demonstrates the potential for building lightweight, custom solutions tailored to user needs. With additional features and polish, TimeTree can serve as both a functional tool and a research prototype for future version control systems.

**Declaration of authorship:**

“The Senior Project/Bulgarian Diploma Thesis presented here is the work of the author solely, without any external help, under the supervision of ….. All sources, used in development, are cited in the text and in the Reference section.”

Author:

Table of Contents

[**1. Introduction** 5](#_Toc215398641)

[1.1 Motivation 5](#_Toc215398642)

[1.2 Project Overview 5](#_Toc215398643)

[**2. Specification of the Software Requirements and their Analysis** 7](#_Toc215398644)

[2.1. Functional Requirements 7](#_Toc215398645)

[2.2 Non-functional Requirements 8](#_Toc215398646)

[2.3 Requirements Analysis 9](#_Toc215398647)

[2.3.1 Analysis of Functional Requirements 9](#_Toc215398648)

[2.3.2 Analysis of Non-Functional Requirements 15](#_Toc215398649)

[**3. Design of the Software Solution** 16](#_Toc215398650)

[3.1 Main Algorithms 16](#_Toc215398651)

[3.1.1 Content Addressing with SHA-1 Hashing 16](#_Toc215398652)

[3.1.2 Myers Line-Based Diff Algorithm 18](#_Toc215398653)

[3.1.3 Rolling Checksum for Binary Deltas 19](#_Toc215398654)

[3.2 Algorithm Complexity 20](#_Toc215398655)

[3.3 User Interface (Command-Line UX) 22](#_Toc215398656)

[3.4 Software Architecture 23](#_Toc215398657)

[3.5 Storage Model & On-Disk Formats 26](#_Toc215398658)

[3.6 Security & Reliability Considerations 27](#_Toc215398659)

[3.7 Determinism & Cross-Platform Guarantees 28](#_Toc215398660)

[3.8 Design Decisions & Trade-offs 29](#_Toc215398661)

[3.9 Extensibility Strategy 29](#_Toc215398662)

[**4. Implementation** 31](#_Toc215398663)

[4.1 Development Environment and Programming Language 31](#_Toc215398664)

[4.2 External Libraries 32](#_Toc215398665)

[4.3 Code Fragments 36](#_Toc215398666)

[4.3.1 Incremental Hashing 36](#_Toc215398667)

[4.3.2 Myers Diff - Core Edit Computation 38](#_Toc215398668)

[4.3.3 Rsync-Style Delta Computation 39](#_Toc215398669)

[4.4 Interesting Design Patterns 41](#_Toc215398670)

[4.4.1 Command-Based CLI 42](#_Toc215398671)

[4.4.2 Strategy Interfaces for Hash, Diff, and Delta 42](#_Toc215398672)

[4.4.3 Purity and Testability 42](#_Toc215398673)

[4.5 End-to-End Execution Flow: init Command 43](#_Toc215398674)

[**5. Testing** 44](#_Toc215398675)

[5.1 Test Environment and Organization of the Test Suite 45](#_Toc215398676)

[5.2 Unit Tests for Core Algorithms 46](#_Toc215398677)

[5.2.1 SHA-1 Implementation 46](#_Toc215398678)

[5.2.2 Myers Diff Engine 47](#_Toc215398679)

[5.2.3 Rsync-Style Delta Engine and Supporting Utilities 47](#_Toc215398680)

[5.2.4 Object Identifiers and Domain Separation 48](#_Toc215398681)

[5.3 Integration Tests for Repository Behavior 49](#_Toc215398682)

[5.3.1 Repository Initialization and Detection 49](#_Toc215398683)

[5.3.2 Staging, Committing, and Inspecting History 50](#_Toc215398684)

[5.3.3 Diff, Checkout, Branch, and Delta Commands 50](#_Toc215398685)

[5.4 Acceptance Testing of User-Facing Scenarios 51](#_Toc215398686)

[5.5 Functional Requirements Testing 53](#_Toc215398687)

[5.6 Test Coverage 53](#_Toc215398688)

[**6. Results and Conclusion** 54](#_Toc215398689)

[6.1 Comparison with Initial Goals 55](#_Toc215398690)

[6.2 What Works Well and What Does Not 56](#_Toc215398691)

[6.3 Problems Encountered and How They Were Resolved 56](#_Toc215398692)

[6.4 Screenshots and Example Workflows 58](#_Toc215398693)

[6.5 Missing Features and Future Development 62](#_Toc215398694)

[6.6 Conclusion and Personal Reflection 64](#_Toc215398695)

[**7. References** 65](#_Toc215398696)

# 

# **1. Introduction**

## 1.1 Motivation

As a front-end developer, I use Git every day, but for years it felt like a powerful black box: I could memorize the workflows: add, commit, push - yet the why behind objects, refs, and history I could never understand. My primary goal in this project is to open that box by building a small, working system that mirrors Git’s core mechanics. By re-implementing the pieces myself - content-addressed objects for integrity, a staging area, deterministic trees for reproducible snapshots, commits with parents, and symbolic references (like HEAD) to identify the current branch - I’m forcing myself to internalize not just how commands behave but what data structures make them safe and predictable.[1] [2].

Finally, this project is a platform for experimentation. Mature VCSes must balance decades of history, ecosystem constraints, and backward compatibility; they are difficult places to explore and create new features. A compact, layered, Kotlin implementation makes it feasible to prototype ideas quickly, measure their trade-offs, and keep only what improves clarity, correctness, or robustness. For example, such experiments are the deterministic object storage I am creating to achieve the same repository state across different platforms, and the adaptive delta compression based on rolling checksums to optimize file synchronization. In short, TimeTree is both a learning and a practical tool: it helps me understand how Git works by building a working version of its core, while also paving the way for research-quality experiments in determinism, storage, and differencing - the latter grounded in well-studied techniques such as Myers’s O(ND) algorithm [3].

## 1.2 Project Overview

TimeTree is a lightweight, local version control system implemented in Kotlin with a small, command-line interface. The system focuses on the essential snapshot pipeline: init creates the repository skeleton under .timetree/, add streams file contents into a content-addressed object store and records paths in a staging index, and commit converts the staged state into a deterministic directory tree and a commit that updates the current branch. A helper command compares Git-style and TimeTree-style object IDs to illustrate domain separation. The goal is a tool that is easy to reason about end-to-end while remaining faithful to the core ideas behind mature VCSes like the most used one - Git.

At the heart of TimeTree is a compact object model. Files are stored as blobs - raw bytes addressed by a SHA-1 identifier computed over a namespaced header and the content. A tree serializes a directory snapshot as stable, human-readable lines that list modes, names, and object IDs; entries are sorted so identical input always yields the same tree ID. A commit records the root tree, optional parent commit, author/committer information, and the message, forming a simple history DAG (Directed Acyclic Graph). Objects are immutable and persist in .timetree/objects/, addressed by their IDs, which makes the repository auditable and easy to inspect with standard tools.

Architecturally, TimeTree is organized in clear layers. The CLI layer handles argument parsing and user feedback. The core services provide repository layout, object storage, index management, reference updates, and all the core functionality behind the CLI commands. Above them sit small writers/builders that translate staged entries into tree bytes and commit payloads. This separation keeps the codebase approachable, enables focused unit tests, and makes it straightforward to extend the system with additional commands without any concerns of breaking other code. Essentially following the separation of concerns principle. [4]

Testing accompanies the implementation throughout. The hashing module is verified against known vectors; the init and add commands are tested for correct repository structure and path normalization; the commit command is validated end-to-end by inspecting on-disk blobs, trees, commits, and updated refs. These are just some of the tests. Every command that is in the project has dedicated tests to ensure they are all working correctly.

TimeTree is deliberately built to be scalable and easy to extend. The command surface follows a clean command pattern, so adding a new feature - status, log, diff, branch, checkout, or a porcelain wrapper - means wiring a new subcommand that composes existing services rather than modifying core logic. The layered design keeps dependencies one-way and narrow, which makes changes local and lowers regression risk. Finally, comprehensive tests provide a safety net, so new commands can be added quickly without any problem. In short, TimeTree’s architecture favors incremental growth: small, composable commands over a stable, deterministic core, enabling the project to evolve in capability without increasing complexity.

# **2. Specification of the Software Requirements and their Analysis**

## 2.1. Functional Requirements

1. The application should initialize a new repository, creating the .timetree/ directory with objects/, refs/heads/, the HEAD file pointing to refs/heads/master.

2. The application should stage files with the add command by streaming their contents, storing a blob object in .timetree/objects/, and recording a repo-relative POSIX path and object ID line in .timetree/index. Files inside .timetree/ must be skipped.

3. The application computed deterministic object IDs using a SHA-1 hasher over a namespaced header and content, ensuring stable IDs and domain separation from Git.

4. The application created commit objects containing tree, optional parent(s), author/committer, timestamps, and message; the first commit had no parent, subsequent commits formed a history DAG.

5. The application advanced branch references atomically with branch or checkout command, writing the new commit ID to refs/heads/<branch> while keeping HEAD as a symbolic ref; the CLI printed Switched to branch [branch].

6. The application provided status, comparing working tree vs. index vs. HEAD and listing staged/unstaged/untracked changes.

7. The application provided log, walking parents from the current ref and displaying abbreviated IDs, timestamps, authors, and messages in reverse chronological order. Includes the option --all for showing all commits in the repository.

8. The application provided diff, showing line-level changes between blobs/commits/trees/branches based on a Myers O(ND) - algorithm, with clear +/- hunks.

9. The application included hash-object command to display/compare Git-style vs. TimeTree IDs.

## 2.2 Non-functional Requirements

1. Crash-safe repository state via atomic write-then-move for objects, index, and refs, object contents verified with cryptographic hashes.

2. Repository state (blobs, trees, commits, refs, index) stored as plain files, human-readable where possible, enabling manual inspection.

3. Works on macOS, Linux, and Windows. Index stores repo-relative POSIX paths, so behavior is consistent across platforms.

4. Concise CLI with --help or -h for each command, clear single-line confirmations/errors, and correct exit codes. Each command has a maximum of 10 sentences concise description, and max 5 flags.

5. Layered architecture and Command pattern allow adding new commands (e.g., merge, reset, stash) by composing existing services without modifying core logic.

6. Performance should feel instantaneous; every command should not have noticeable delays. Committing up to 100 small/medium files finishes in under a second; text diff should complete in under a second; updating/checkout of around 1000 tracked files should complete in under a second. Overall, near instantaneous performance.

7. Scalability should not be a problem. The project is designed to grow over time, handling thousands of files and objects; committing over 10000 should not be a problem, over 100000 objects should not be a problem, and repositories up to 5 GB should not be a problem. All of this is possible thanks to the rsync algorithm.

8. The system defends against partial writes, truncated objects, and path confusion. Writes are atomically renamed; object headers, sizes, and types are strictly validated; refs are updated atomically.

## 2.3 Requirements Analysis

### 2.3.1 Analysis of Functional Requirements

The core of the application lies with the selected commands that are implemented and the logic behind them. For that reason, the functional requirements follow logical order. They were selected because they are at the core of how a version control system (VCS) should behave. The algorithms that are running the project are also at the core of what a VCS is. Now I will be explaining what each of the requirements is about.

1. The **init** command creates the repository’s deterministic, inspectable scaffold. When run in a writable working directory, it builds .timetree/ with objects/ for content-addressed blobs/trees/commits, refs/heads/ for branch pointers, and a HEAD file that is a symbolic ref to refs/heads/master. This keeps the user on an explicit branch from the start and aligns the later workflow.

A diagram of a program

AI-generated content may be incorrect.

*Figure 1 - Use Case Diagram of the Core TimeTree Commands*

The operation is idempotent and crash-safe. If .timetree/ already exists, the command verifies/repairs the expected structure instead of failing, ensuring objects/, refs/heads/, and a well-formed HEAD. Files are written via atomic “write-then-move” to prevent torn writes. This satisfies reliability and transparency goals by leaving the state consistent even on failure. Edge cases are handled with clear outcomes: non-writable directories or a non-directory .timetree yield a descriptive error and non-zero exit; partially initialized repos are repaired by creating only missing pieces.

2. The **add** command stages one or more paths by streaming file bytes through the SHA-1 hasher (with TimeTree’s namespaced header) and writing a blob to .timetree/objects/. For each successfully hashed file, it records a single line in the index as <40-hex-id> <relative-path>. The command prints a concise confirmation per file so the user sees exactly what will be committed. Before processing, **add** validates inputs to protect repository integrity and user intent. Any path outside the repo root and under ./timetree is skipped. For duplicate paths, the latest blob ID replaces the previous entry in the index, which keeps the staging area a simple, last-writer-wins map. The index file itself is persisted atomically, so partial writes can’t corrupt the staged state.

3. TimeTree computes object identifiers by hashing a namespaced header concatenated with the exact object bytes, ensuring determinism and domain separation from Git. For a blob, the header is "timetree:v1\0blob <size>\0"; for trees and commits, analogous headers are used. The stream of bytes header and content is fed into my SHA-1 hasher, so identical inputs always yield the same 40-hex ObjectId, while the same content hashed with Git’s header ("blob <size>\0") yields a different ID. This prevents cross-tool collisions and makes IDs stable.

4. Given the root tree ID and the current tip (if any), the commit writer assembles a deterministic payload containing tree <treeId>, optional parent <commitId> (omitted on the first commit), author and committer identities with timestamps, and the message, then hashes header and payload using the TimeTree commit header to produce the immutable commit object, which is stored under .timetree/objects/. Empty messages are allowed with the flag --allow-empty.

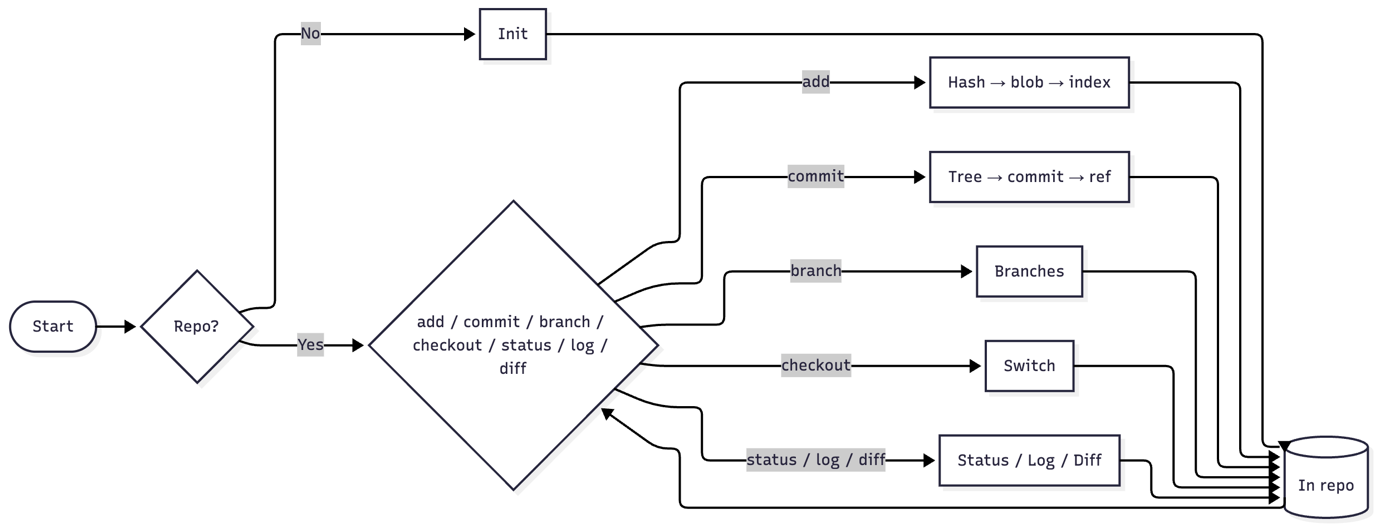
5. After writing the commit object, the system atomically advances the current branch by writing the new commit ID into refs/heads/<branch> while keeping HEAD as a symbolic reference (ref: refs/heads/<branch>). The update uses a write-to-temp-then-rename strategy to prevent torn writes, so a crash never leaves a half-written ref. On fresh repositories, <branch> is the default from HEAD (master).

6. The status command compares three views of the project - the working tree, the index, and the HEAD tree - to classify paths as staged (index different than HEAD), unstaged (working hash different than index), or untracked (present in working tree, absent from index).

7. The log command shows the commit history starting from the tip of the current branch, repeatedly reading each commit object, and printing its information. Each entry displays the abbreviated commit ID, timestamp, author, and the first line of the message, in reverse chronological order. Because commits are immutable and linked by parent IDs, traversal is simple and deterministic.

8. The diff command shows line-level changes between two versions - either two blobs directly or two commits/trees by matching files with the same path. It uses a Myers O(ND) - algorithm to compute a near-minimal edit script and renders a unified diff with - (deletions), + (additions), and a few context lines for readability.

9. The hash-object command demonstrates domain separation and determinism by computing two IDs for the same file: a Git-style ID and a TimeTree-style ID. Because the headers differ, the resulting 40-hex digests are intentionally not equal even when the content is identical, proving that TimeTree objects cannot collide with Git’s namespace.



*Figure 2 - Activity Diagram of the TimeTree Command Workflow*

Finally, let us analyze some specific use case narratives for the application. This is important because the project is user-focused and designed with the end user in mind throughout the whole implementation process.

i. **Name:** Initialize Repository

* **Actor:** User
* **Entry Condition:** The user is in a writable working directory without a valid .timetree/.
* **Flow:** The user runs timetree init. The system creates .timetree/objects, .timetree/refs/heads, and writes HEAD -> refs/heads/master.
* **Exit Condition:** A valid repository skeleton exists and the CLI confirms initialization.
* **Exceptions:** If .timetree exists but is broken, the system repairs missing pieces. If the directory is not writable or .timetree is a file, the command fails with an error.

ii. **Name:** Stage File(s)

* **Actor:** User
* **Entry Condition:** A TimeTree repo exists and at least one file is present in the working directory.
* **Flow:** The user runs timetree add <filename> or “.”. For each file, the system validates location, streams the file to the object store, and records <id> < relative-path> in the index file.
* **Exit Condition:** Selected files are staged and listed in .timetree/index
* **Exceptions:** Files outside the repo or under .timetree/ are skipped with a message.

iii. **Name:** Commit Staged Changes

* **Actor**: User
* **Entry Condition**: The repository has at least one staged file in the index.
* **Flow**: The user runs timetree commit -m "<message>". The system builds a deterministic tree from the index, creates a commit, writes it to the object store, and atomically updates refs/heads/<branch>.
* **Exit Condition:** A new commit exists, and the branch tip points to it.
* **Exceptions:** If nothing is staged (nothing to commit), the system prints “Nothing to stage - all files are already up to date.”.

iv. **Name:** View Status

* **Action**: User
* **Entry Condition:** A repository exists, and changes may have been made.
* **Flow:** The user runs timetree status. The system compares working tree, index, HEAD tree and lists staged, unstaged, and untracked files.
* **Exit Condition:** The user sees a summary of the current state.
* **Exceptions:** None beyond normal I/O errors, unreadable files are reported and skipped from classification.

v. **Name:** View Commit Log

* **Actor:** User
* **Entry Condition:** The repository has at least one commit.
* **Flow:** The user runs timetree log. The system shows the current branch and prints entries with abbreviated ID, author, timestamp, and commit message.
* **Exit Condition:** A reverse-chronological list of commits is displayed.
* **Exceptions:** If there are no commits yet, the command reports an empty history.

vi. **Name:** Show Diff Between Versions

* **Actor:** User
* **Entry Condition:** Two versions of a file exist
* **Flow:** The user runs timetree diff. The system matches files by path, computes line-level differences with the Myers diff algorithm and renders unified -/+ hunks.
* **Exit Condition:** A readable textual diff is shown for changed files.
* **Exceptions:** No staged changes meaning nothing to commit, I/O error.

### 2.3.2 Analysis of Non-Functional Requirements

TimeTree maintains a crash-safe repository state by using atomic write-then-move strategy for every update - objects, the index, and refs. Bytes are written to a temp file in the same directory, flushed, and only then renamed into place, so the repo can’t be left half-written after a failure. The objects are content-addressed with a cryptographic hash, so corruption is detectable. All repository items are plain files and intentionally human-inspectable. Blobs store raw bytes, trees and commits use compact line-based encodings, refs are single-line hex files, and HEAD is a readable symbolic reference - this makes debugging and auditing straightforward. The system is cross-platform: it runs on macOS, Linux, and Windows, and it normalizes every staged path to a repo-relative POSIX form. That guarantees identical inputs produce identical IDs across operating systems. The CLI is concise and friendly. Each command supports -h/--help, with all commands returning a result message - either a success one or a failure one.

Finally, the project is extensible by design. A layered architecture separates the CLI, core services, and builders/writers. Each subcommand is a small Command object that composes these services, so new features like merge, reset, or stash can be added without modifying low-level modules. Dependencies run one way (CLI to services to storage), keeping changes local and regression risk low. In terms of performance, everyday tasks are designed to feel instantaneous. Commits, diffs, and checkouts don’t stall the system, and there is no noticeable slowdown. Every operation is performed in well under one or a maximum of two seconds. For scalability, the project is optimized, and there should be no problem with repositories under 5 GB. This size is perfect for the scope of the project. The rsync algorithm helps immensely with the optimization of storage. Security is not limited only to the hashing of the files, but every operation in the project is also strictly validated before proceeding. These are the functional and non-functional requirements of the project.

A diagram of a computer program

AI-generated content may be incorrect.

*Figure 3 - System Workflow during a Commit Operation*

# **3. Design of the Software Solution**

## 3.1 Main Algorithms

The application has been developed using three algorithms. These three have been selected to fulfill all the needs of a fully working version control system. Those three are a SHA-1 hash for content addressing, the Myers Longest Common Subsequence algorithm for line-by-line diffs, and a rsync-inspired rolling-checksum algorithm for binary delta generation and application.

### 3.1.1 Content Addressing with SHA-1 Hashing

TimeTree names every blob, tree, and commit by hashing a canonical header plus the exact bytes - timetree:v1\0<kind> <length>\0 || content - using a streaming SHA-1 compressor. The 160-bit digest becomes the filename under .timetree/objects/, giving an immutable, content-addressed store where identical bytes always map to the same ID and any bit flip yields a different one. Here, the explicit namespace (timetree:v1) provides domain separation from Git [5]. This is basically what my custom SHA-1 does, but why have I chosen it?

There are a couple of reasons. Well, a NIST-standardized SHA family gives a well-studied primitive with a clean, verifiable storage model [5]. Another reason is that it mirrors Git’s historical object model, which makes the mechanics easy to learn and validate. It’s small enough to implement transparently as a streaming compressor, so large files don’t need buffering, and the code stays relatively simple. The 160-bit IDs are ergonomic with abundant test vectors. In my local, single-user threat model, known SHA-1 collisions aren’t a blocker [6]. The timetree:v1 namespace keeps IDs separate from Git and preserves a clean upgrade path to SHA-256/BLAKE3 later. A reason for not choosing SHA-256, which Git will use in its upcoming 3.0 version, is that it strengthens collision resistance but doesn’t change the mechanics I’m implementing and increases digest size/cost - so I treat it as a drop-in upgrade, the header format already anticipates [7]. BLAKE3 follows the same reasoning as SHA-256. While it may be significantly better, it adds complexity I don’t need for the current scope of the project. However, as mentioned beforehand, the migration to better hashing algorithms is open and can be done.

Let’s compare to other known algorithms - MD5, SHA-512, and CRC32. The first one, MD5, is cryptographically broken. Chosen-prefix collisions have been demonstrated, meaning an attacker can create two different contents with the same MD5, resulting in different contents with the same hash [8], [9]. Continuing with SHA-512, it offers collision/preimage margins far beyond the project’s single-user model and adds digest size/CPU cost without any practical benefit here. NIST recommends matching the hash strength to the application needs [10]. Finally, the CRC32 is not a cryptographic hash. It’s linear and easy to collide; it’s only suited for random-error detection [11]. Given all these trade-offs, SHA-1 with domain-separated headers keeps the design and implementation simple with a clean path for upgrades.

### 3.1.2 Myers Line-Based Diff Algorithm

TimeTree computes textual differences using Myers’s O(ND) algorithm, a dynamic programming approach for finding the shortest edit script (SES) between two sequences. It works by having two files represented as an ordered list of lines that the algorithm uses to identify the minimal sequence of insertions and deletions that transforms one list into the other. In practice, it creates a search space of diagonals - each representing a fixed offset between positions in two files. Then it advances along the furthest-reaching paths in that space until the target endpoint is reached. After that, the resulting trace is backtracked to produce a precise edit script of “keep”, “insert”, and “delete” operations. This algorithm is the same one that Git uses for its diff command implementation, still standing as the default choice, while having the option to choose from other diffing algorithms such as patience [12]. Unix diff implementations also choose it for its balance between theoretical optimality and real-world performance [3].

There are several reasons why I chose the Myers algorithm for my diff implementation. First, it guarantees a minimal edit script in O(ND) time, where N is the combined number of lines and D is the minimal number of edits. This makes it efficient even for large text files with small differences. Second, it creates clean, human-readable diffs without the noise that simpler methods often generate. Third, it aligns perfectly well with TimeTree’s layered architecture. The diff engine is in the core layer, doesn’t care about files, it sees only two lists of lines, and is called by the CLI command layer, the diff command.

Alternative algorithms we considered. The dynamic programming matrix that is used in the traditional LCS algorithm also guarantees the same minimal result but consumes O(N²) memory, which is not good for files beyond a few thousand lines [13]. The Patience diff algorithm, the other option that git allows, offers the same but sacrifices optimality and determinism. It relies on experimentations that can skip valid matches under certain patterns [12]. Myers's method, on the other hand, returns exact results with modest memory use (O(N+M)) and deterministic output. Those properties are crucial in my project. Overall, Myers’s diff was chosen because it is proven, optimal, and predictable. It fulfils all the needs of the project and can allow for further improvements, such as the conflicts visualization.

A diagram of a process

AI-generated content may be incorrect.

*Figure 4 - Simplified workflow of the Myers Diff Algorithm*

### 3.1.3 Rolling Checksum for Binary Deltas

TimeTree implements a rsync-style rolling-checksum algorithm to efficiently compute binary deltas between file versions. The goal of its implementation in the project is to avoid saving entire files when only small portions of the file have changed. A simple explanation of how it works is the following. The algorithm divides a basis file into fixed-size blocks and computes for each block both a weak rolling checksum and a strong hash - my own implementation of the SHA-1. After that, during delta generation, the target file is scanned with a sliding window of the same block size. The rolling checksum then enables quick detection of matching blocks without rereading or rehashing the entire basis. When a match is found, the hash confirms it, and unmatched regions become insert operations. The result sequence of “copy” and “insert”, called delta, reconstructs the target file from the basis. This entire approach is directly inspired by the original rsync algorithm by Tridgell and Mackerras [14]

Again, there are several reasons why I have chosen this algorithm. First, the rolling-checksum method scales well. It runs in linear time with respect to file size (O(n)) and uses a small amount of memory. Second, the algorithm is easy to implement and reason about. Its correctness can be proven from simple arithmetic transformations of the rolling checksum. Finally, it supports incremental synchronization and the possibility of future improvements and extensions.

Like the previous two algorithms, alternative options here were considered as well. One option is the block-wise cryptographic hashing. It requires recomputing all block digests after every byte shift. This makes it significantly slower for large files with small edits [15]. Binary differencing via bsdiff/vcdiff is also another option, but it produces smaller deltas at a much higher computational cost and implementation complexity [16]. The rsync-inspired approach is the most useful for TimeTree. It is fast, incremental, and deterministic.

## 3.2 Algorithm Complexity

All those algorithms implemented in Timetree were selected not only for their correctness and determinism but also for their efficiency. Each one of them - hashing, text differencing, and binary delta computation is designed to run in linear or near-linear time relative to the size of the processed data. This ensures the scalability to big repositories while keeping the implementation simple and not highly complex.

The SHA-1 hashing algorithm computes and digests in O(n) time, where n is the size of the input file. Here, each byte is processed once through 64-byte block updates, on which are performed a fixed number of arithmetic and bitwise operations. Since the algorithm works as a streaming compressor, it does not buffer the entire file in memory. From this, the memory footprint it has remains constant O(1), determined only by the 160-bit initial state and a 512-bit message block. All of this allows TimeTree to hash large files efficiently and is limited only by the disk I/O bandwidth.

The second algorithm, Myers O(ND), computes the shortest edit script between two sequences in O(ND) time and O(N + D²) space, where N is the total number of lines and D the minimal number of differences [3]. Even better is when we have changes that are small compared to the file size, we get near-linear performance. The implementation I have tracks the furthest-reaching position on each diagonal using two compact integer arrays that swap each iteration and record minimal backtrack metadata per step. This means that memory usage grows only with the number of lines. For ordinary files of a few thousand lines, the algorithm completes in milliseconds.

The third algorithm, the rsync-inspired rolling-checksum, runs in O(n) time and O(b) space, where n is the length of the target file and b is the block size. Each sliding-window update needs only a few arithmetic operations to roll the checksum forward, which enables efficient scanning without recomputing full hashes for every shift. The delta application is also linear, as the operations it performs, “copy” or “insert”, are done one by one. Because it does only a small, fixed amount of work per byte, this works well for large files and incremental updates.

In contrast, the other alternatives for the algorithms have quadratic or higher costs in the worst-case scenario. An example of this is the cryptographic block hashing or suffix-array-based binary differencing (bsdiff) [16]. Those algorithms can achieve slightly smaller deltas or better guarantees, but are not suitable for TimeTree’s scope. Overall, the chosen algorithms, SHA-1 hashing, Myers line-based diff, and rolling-checksum delta provide a great balance between speed, determinism, and simplicity while maintaining linear or near-linear runtime and constant memory usage.

## 3.3 User Interface (Command-Line UX)

TimeTree is deliberately **CLI-first**: every action is an explicit verb, and every verb has a single predictable responsibility (outcome). The interface favors short commands with stable flags over fancy or overcomplicated wizards, so behavior is easy to reason about and test. When launched without arguments, the tool shows a compact “start screen” with all the commands shown and their descriptions. Each command supports -h/--help flags so that users know what the command does. This makes the learning curve shallow while keeping the mental model clear: the user chooses a verb (init, add, commit, status, log, diff, branch, checkout, to name a few) and passes paths or options. After that, the program prints a single result line and exits.

The UX is designed around **determinism** and **transparency**. The read-only commands never mutate the repository state. The mutating commands describe exactly what will happen and then perform it atomically. Paths provided by the user are normalized to repo-relative POSIX form, which guarantees consistent output across macOS, Linux, and Windows and prevents accidental writes outside the repository. The output is intentionally minimal and inspectable: status summarizes working vs. index vs. HEAD; log walks parents from the current ref and prints abbreviated IDs, authors, and timestamps; diff renders unified hunks. Some of the commands use color formatting (ANSI accents) to improve legibility, but the formatting never depends on the color and the output of the commands as well.

Error handling follows the same principles - fail fast, explain once, and suggest a next step. Invalid paths, attempts to add files under ./timetree/, or malformed refs are caught early and messages are printed, and the program exits non-zero. Success is equally explicit: commit prints the new object ID; branch prints the created or selected branch; checkout confirms the target ref or commit. Because the messages are single-line and stable, they work both for users and for lightweight shell pipelines. The result is a small, consistent CLI that exposes the core ideas - hashing, staging, committing, and reading history - without hidden state or noisy output.

A screenshot of a computer program

AI-generated content may be incorrect.*Figure 5 - TimeTree CLI Interface*

## 3.4 Software Architecture

The application has been developed following a layered architectural pattern, ensuring the separation of responsibilities, testability, and deterministic behavior. This design enables the system to be divided into two main layers and a sublayer. Those are the **CLI layer**, the **Core layer,** and within the Core, the storage sublayer**.** Each one serves a different purpose and depends only on the one directly below it. The chain is this one: CLI -> Core -> Storage. This allows each part of the application to evolve independently without breaking the functionality elsewhere. This overall structure is especially well-suited for command-line tools and version control systems since they need simplicity and robustness rather than heavy UI frameworks.

The **CLI layer** is responsible for the user interaction. It does it through individual command objects. The core commands in the project are init, add, commit, status, log, diff, branch, and checkout. This layer handles the argument parsing, validation, and feedback messages or exit codes. Here, one of the most significant things is that it does not access the file system directly. Instead, it delegates all such changes to the Core layer. This ensures that we have a predictable interface between the user input and system logic.

The **Core layer** contains all the essential logic behind TimeTree. It implements the algorithms and the deterministic operations that define the repository. Within this layer, all the packages, such as hash, index, checkout, delta, and diff, to name a few, work together to support all the version-control operations. Each one of them performs its purpose through clear and straightforward interfaces. For example, the hashing package converts file content into immutable object identifiers using my SHA-1 implementation. The Diff engine computes line-by-line textual differences using Myers’s O(ND) algorithm, and the delta engine generates binary differences through a rolling-checksum algorithm inspired by rsync. By keeping all the logic here, the Core layer provides the foundation for determinism - essential for TimeTree so that identical inputs always give the same output, regardless of operating system or environment.

The **storage sublayer** inside the Core layerrepresents the lowest level of the architecture. It is responsible for the direct interaction with the file system - .timetree/ main directory and its subfolders, such as the objects/, refs/, and index. All the writes performed follow the atomic write-then-move principle. The principle is an absolute must in the project because its essence is that the data is first written to a temporary file in the same directory, flushed, and then renamed into place. This guarantees crash-safety and prevents partial updates. An example is if the computer suddenly powers off, the file won't be corrupted - it's either fully updated or remains unchanged. Another thing that the storage sublayer enforces is the normalization of the file path to the POSIX format. This ensures that the repository remains portable and consistent across macOS, Linux, and Windows environments.

The **layered architecture** was chosen over alternatives such as monolithic or service-oriented designs because it gives the best balance between clarity and extensibility. This clear division of responsibilities allows for easy debugging, easier updates, and simple testing of each layer. For example, new commands such as merge, reset, or stash can be simply added by using the existing Core packages without the need to modify anything else but extend upon them. In a similar fashion, migration to stronger hashing algorithms like SHA-256 or BLAKE3 will require only changes to the hashing package, leaving the other layers without any core changes. This structure not only simplifies future development but also improves maintainability and fault isolation - thus making the architecture both robust and scalable for future work on the project. The following figure showcases how the layered architecture works. A simple illustration of the lifecycle of some of the core commands.

A diagram of a structure

AI-generated content may be incorrect.*Figure 6 - Layered architecture command flow*

## 3.5 Storage Model & On-Disk Formats

TimeTree stores all repository state under .timetree/ using a layout chosen for readability: objects/ holds immutable data (blobs, trees, and commits), refs/ contains branch pointers, the index is a plain-text file describing what's staged, and HEAD is a symbolic ref. All paths recorded are repo-relative POSIX (i.e., forward slashes), so equal input means equal output on macOS, Linux, and Windows. All mutator operations have the same durability rule - write -> atomic rename - which makes it impossible for a crash to leave half-written data.

Objects are content-addressed with a 40-hex ID computed over a namespaced header plus the exact payload bytes: timetree:v1\0<kind> <size>\0 || payload. Including the header in the digest makes IDs deterministic and domain-separated from Git; any bit flip changes the ID and is detectable on read by recomputation. This keeps the storage verifiable without extra metadata.

Each object kind uses a compact, text-friendly encoding. Blobs are raw file bytes. Trees list entries one per line as <mode> <path>\t<id>, sorted by <path> for stable IDs. Commits contain tree, optional parent lines, author/commiter with timestamps, a blank line, then the UTF-8 message. The index (used by add/commit) maps <id> <path> per line and is last-writer-wins if a path repeats - it is also persisted atomically.

Branches and the current branch are plain text for clarity and easy recovery. HEAD stores ref: refs/heads/<branch>, and each refs/heads/<branch> file holds a single 40-hex commit ID. A commit’s objects are written first; only then is the branch ref updated, so a branch never points to a non-existent object. On read, headers, sizes, paths, and IDs are validated; malformed or missing targets are reported clearly, while read-only commands never mutate state. The timetree:v1 prefix reserves a forward-compatible path (e.g., v2) without breaking existing repositories.

## 3.6 Security & Reliability Considerations

TimeTree’s threat model is local and single-user. The repository lives on the user’s local machine, and not on a server. The primary risks are data loss, state corruption, and path confusion, rather than network attacks that we can find on a remote server. Security, therefore, is enforced through deterministic storage, strict validation, and crash-safe writes, with design choices that make tampering and partial updates immediately visible.

All mutating operations follow the same process: write -> atomic rename. Objects, the index, and refs are first written to a temporary file in the same directory. They are flushed to the disk, and only then are they renamed. Refs are updated after their target objects are safely stored. This way, a branch can never point to a missing commit, even during a sudden power loss. Read-only commands (status, log, diff) do not change the state, so they cannot introduce inconsistency during inspection.

Integrity is built via content addressing. Every blob, tree, and commit filename is the 40-hex digest of a namespaced header + payload (timetree:v1….). On read, headers are re-validated and the digest is recomputed - any bit flip or truncation is detected. Namespace separation prevents accidental collisions with other tools, like Git, and keeps a clean upgrade path.

Input handling is defensive and portable. All paths are normalized to repo-relative POSIX form (no …, no escapes, skip .timetree/). Trees validate mode bits and referenced IDs; refs must be well-formed 40-hex. Large operations, like checkout, proceed file-by-file with clear errors, hashing/diff/delta are streaming with bounded memory. The header namespace also enables cryptographic agility, meaning that switching to better hashing algorithms is not a problem, and it is without making any architectural changes. In short, TimeTree’s security comes from simple, verifiable formats, one-way write ordering, and defensive validation that keeps the repository correct, recoverable, and predictable.

## 3.7 Determinism & Cross-Platform Guarantees

The project’s determinism target is: given the same staged bytes, paths, and metadata, produce the same IDs and trees on macOS, Linux, and Windows. The guarantees follow directly from the previous section 3.4, like the canonical headers, sorted tree entries, etc. Platform specifics are handled by POSIX path normalization and by making no implicit EOL (end-of-line) conversions.

Limits and assumptions are explicit. TimeTree models paths as case-sensitive. On case-sensitive filesystems, the working tree may not hold both a.txt/A.txt at the same time. The commit IDs include timestamps, and reproducibility assumes an identical message and time. Within these bounds, the repositories are byte-reproducible across OSes without re-describing the storage formats.

A diagram of a software process

AI-generated content may be incorrect.

*Figure 7: Inputs that define IDs and how TimeTree normalizes them.*

## 3.8 Design Decisions & Trade-offs

TimeTree is CLI-first and ships as a single-binary (JAR), while internally using a layered architecture (CLI -> Core with a storage sublayer). A desktop GUI or a client-server split could enable multi-user scenarios, but it adds a surface area that is not needed, and it will be hard to learn and implement. A small, scriptable CLI lets the system be transparent, testable, and easy to reason about. In addition to that, it also keeps failures crisp by having a single command interface and a single exit code instead of diffusing them across layers. The cost is that usability is text-driven, and advanced workflows like visual merge tools, staging previews, conflict resolvers, etc., are postponed.

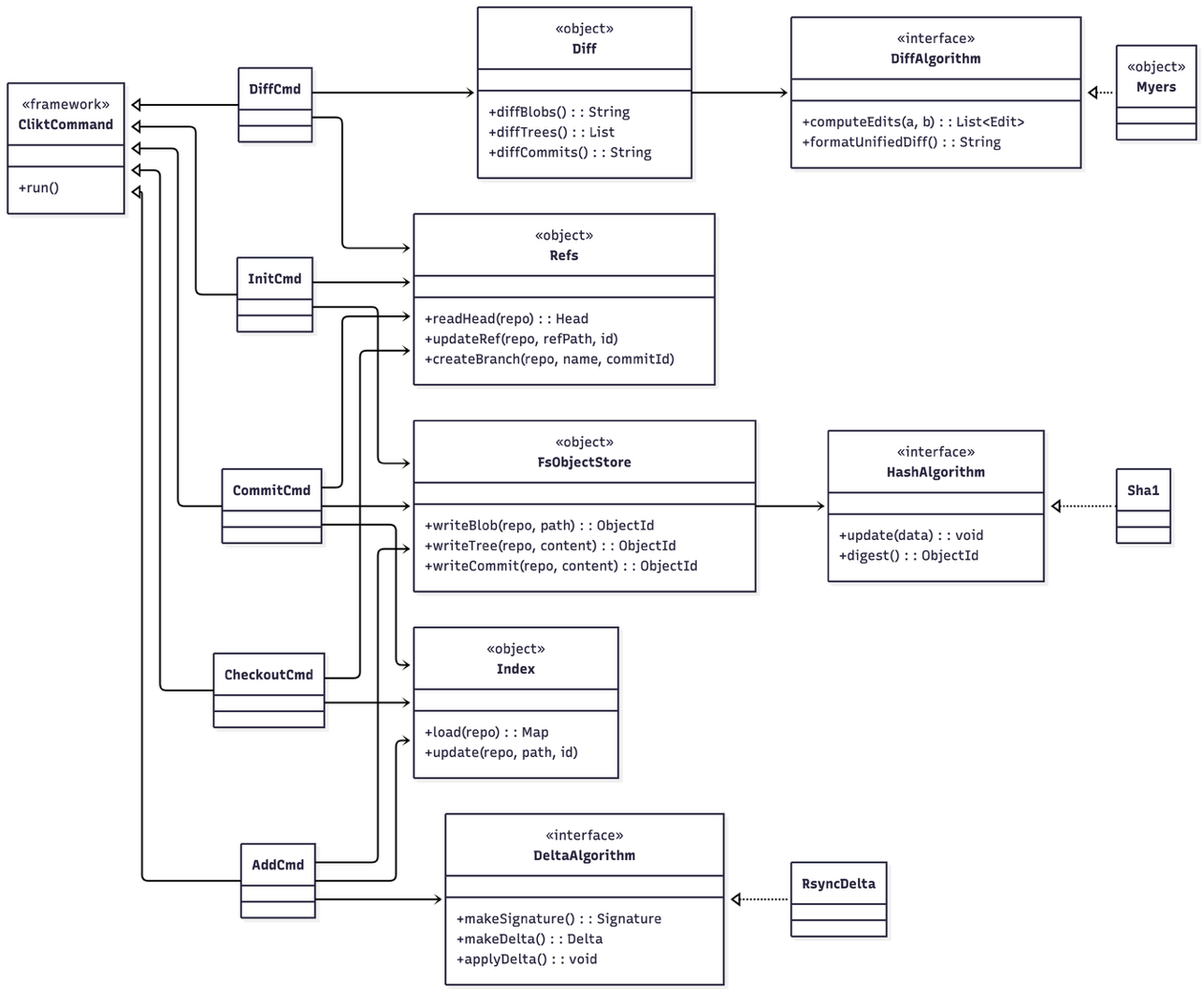
The repository is a plain-file store rather than a database. A DB would offer transactions and indexes, but it would also require schema, migrations, a query layer, and platform dependencies that don’t improve the core VCS semantics. Here, the trade-off is that features like global search or complex queries are not “free”. When needed, they will be built as streaming scans over objects rather than SQL.

## 3.9 Extensibility Strategy

TimeTree is built so that new features can be added without rewriting the core logic that we have. The CLI layer treats every subcommand as a small, composable Command object that depends only on the Core packages. Adding a command like stash or reset means implementing a new Command class, wiring it in the parser, and calling the existing packages (index, refs, object store) - thus keeping growth incremental and futureproof.

Inside the Core, changes are easily made there as well. The hashing algorithms (content addressing), diff strategy (text diffs), and the delta engine (binary deltas) are examples. Currently, they are being implemented using SHA-1, Myers diff, and rsync rolling checksum, but they can be improved and replaced with other algorithms without having to rewrite the whole logic. The repository on-disk format is explicitly namespaced to allow forward-compatible object headers and a clean upgrade path.

The storage stays behind a narrow ObjectStore/Refs/Index façade. If in the future the files live in plain directories, compressed packs, or a DB, the callers won’t change - the core logic won’t change. All writers keep the same atomic principle (write -> atomic rename), so safety won’t ever be a problem. Finally, tests are organized by interface, so again, adding tests will not be a problem at all, and infinitely more can be added.



*Figure 8: Class diagrams showing how the packages connect.*

# **4. Implementation**

## 4.1 Development Environment and Programming Language

TimeTree targets the JVM (Java 21) and is implemented in Kotlin. Kotlin gives the project the right balance of concise syntax, null-safety, and strong modeling, like data classes, sealed hierarchies, etc., while compiling to a single, portable runnable JAR. The JVM (Java Virtual Machine) gives mature, cross-platform I/O - java.nio channels, and file attributes. This allows the code to implement the project’s crash-safety guarantees across macOS, Linux, and Windows without native binding. Determinism also benefits from the object IDs since they are computed from byte-exact streams with no platform-specific code paths.

The development is done in the IntelliJ IDE (integrated development environment) with the Gradle Wrapper and Toolchains pinned to JDK 21. This ensures the same builds across machines. IntelliJ’s Kotlin support, like inspections, inline type hints, and data-flow analysis, helps keep the core small and correct. The project encoding is UTF-8, and source/target bytecode are set to Java 21. Run configurations exist for the CLI entry point and for the tests.

Why Kotlin for TimeTree? Kotlin is a good fit because it keeps the code short and safe while still compiling to fast JVM bytecode. Compared to Java, it removes a lot of boilerplate code but keeps strong typing and great tooling in IntelliJ. Compared to Python, it avoids packaging/interpreter issues and gives a single runnable JAR with better performance for streaming I/O. Compared to Go, it integrates directly with java.nio (paths, channels) and lets me model states cleanly with sealed classes. And while Rust or C++ offer more low-level control, Kotlin gives much quicker iteration and painless cross-platform builds. In practice, Kotlin delivers the balance I need: concise code, strong types, and identical behavior on macOS, Linux, and Windows.

## 4.2 External Libraries

In compliance with the senior project guidelines, no ready-made libraries were used for the development of the main parts of the project, namely the three algorithms used. However, for the other parts of the project, such as the CLI interface, Kotlin formatting, building of the project, and testing of the project, external libraries are used, and here I will explain them.

TimeTree relies on a very small number of external tools, as mentioned above. All the core functionalities of the VCS - hashing, diffing, and delta computation are implemented directly in the project. No external library is used for the development of the three main algorithms as per the requirements.

The first one is: **Clikt**

Clikt is a lightweight Kotlin library for building command-line interfaces. It is used to structure TimeTree’s commands, parse arguments and options, and generate help messages. Clikt is limited to the CLI layer. The internal logic for creating the commits, traversing history, computing diffs, or applying deltas does not depend on it. Clikt was chosen because it integrates naturally with Kotlin, has minimal overhead, and avoids verbose boilerplate.

**Alternatives** are:

* **Picocli** - A powerful Java CLI framework with annotation-based command definitions. While feature-rich, it introduces more complexity, relies heavily on annotations, and is not Kotlin-friendly.
* **Kotlinx.cli** - It is not supported anymore. The last stable release was in 2023, and in the README, it’s clearly stated that it’s no longer supported and should not be used.

The second one is: **Kotest**

Kotest is used for all unit and property-based testing. It offers expressive matchers, structured test scopes, and integrated property-based testing. This makes it convenient for verifying determinism, object store behavior, hashing rules, and the correctness of the Myers diff and delta logic.

**Alternatives** are:

* **JUnit 5** - The default Java testing framework. While reliable, it is more verbose and does not provide native property-based testing. JUnit is still used indirectly as the execution platform for Kotest.
* **TestNG** - Like JUnit but more configuration-heavy and less idiomatic for Kotlin.
* **Spek** - Kotlin DSL-style testing, but development has slowed down, and the ecosystem is smaller. The latest commit in the repository of the framework is from four years ago.

Kotest provides stronger expressiveness, better Kotlin integration, and built-in property testing.

The third one is: **ShadowJar (Gradle plugin)**

ShadowJar is used to produce the final executable JAR (timetree.jar) that packs all required classes and dependencies. It also allows the project to ship a small wrapper script (tt) that behaves like a native terminal command. ShadowJar was chosen because it is reliable, widely used, and integrates seamlessly with Kotlin JVM projects.

**Alternatives** are:

* **Spring Boot’s fat-jar mechanism** - Intended for Spring applications and introduces significant overhead.
* **Gradle Application Plugin only** - Produces a runnable distribution folder but does not create a single bundled JAR, which is necessary for Git-like CLI tools.
* **Ktor’s packaging system** - Irrelevant for non-server applications.

ShadowJar provides exactly what TimeTree needs: a compact, distribution-ready artifact without adding unnecessary runtime frameworks.

The fourth one is: **ktlint (Gradle plugin)**

The ktlint plugin enforces a consistent Kotlin code style throughout the project. It integrates formatting checks into the project to ensure no messy code is left.

**Alternatives** are:

* **Detekt** - A static analysis tool that focuses more on code smells than formatting rules. It performs a different role, and it is not needed for the project.
* **Spotless** - A formatting tool capable of running ktlint internally, but generally preferred for multi-language projects. It introduces additional configuration overhead compared to using ktlint directly.

In all cases, all the libraries and plugins were chosen because they are minimal, integrate cleanly with Kotlin, reduce boilerplate, and provide stable tooling.

4.2. Installation Requirements

TimeTree is a JVM application and is distributed as a single runnable JAR plus two small wrapper scripts called timetree and tt. The project targets Java 21 and uses the Gradle Wrapper for build, so the only big dependency on the user’s machine is a compatible JDK (Java Development Kit). Everything else, like the Kotlin compiler, Gradle version, and the external libraries, is resolved through the wrapper.

In practice, any JDK that can run Java 21 bytecode should work. However, to achieve the same behavior and match the configured toolchains, it is best to install JDK 21, which is guaranteed to work perfectly. The build is managed entirely by the Gradle Wrapper, which is represented by ./gradlew on Unix-like systems and gradlew.bat on Windows, meaning that no separate Gradle installation is required. The wrapper will download the right Gradle distribution for the project. Since TimeTree is a CLI tool, there are no GUI frameworks or native libraries to install. Essentially, if Java runs, the program will also run. The CLI is platform-agnostic and works on all major operating systems, namely macOS, Linux, and Windows.

A single installation path is supported, and it is based on the bundled installer scripts. The workflow is the same across all platforms. The script builds the fat JAR via the Gradle Wrapper and then installs the executable and its wrappers into a directory that is added to the user’s PATH.

On Unix-like systems (macOS and Linux), the installation steps are:

1. Install **Java 21** if it is not already installed
2. **Clone the repository** and cd to the root of the project
3. **Run the installer script** - ./scripts/install.sh

The install.sh will invoke the ./gradlew shadowJar and the custom createTtWrapper task to produce timetree.jar and the tt wrapper. It then copies these artifacts into a suitable directory. If /usr/local/bin is writable, it is used. Otherwise, the script falls back to user-local locations such as $HOME/.local/bin or $HOME/bin, creating them if they do not exist. If the chosen directory is not already in the user’s PATH, the script prints export PATH=… line that can be added to .zshrc, .bashrc, or another shell profile file. After this, the commands timetree and tt are available in any terminal session.

On Windows, the installation process is the same, but it is performed with the PowerShell version of the installation script. Here, the script builds the shadow JAR using gradlew.bat shadowJar, then writes timetree.jar and the wrapper batch files into a per-user directory such as %LOCALAPPDATA%\TimeTree\bin. If this location is not on the Path, the script prints instructions for adding it via the Windows environment variables dialog or a one-line PowerShell command. After updating the Path and restarting the terminal, timetree and tt are available as normal commands.

Once installed, TimeTree commands need to be executed inside a valid TimeTree repository. Such a directory can become a repository after running the tt/timetree init command, which creates the internal .timetree folder. The other commands, such as tt status or tt log, are intended to be run only in such directories. If a command is executed in a folder that does not contain .timetree, the program returns an error: “Not a TimeTree repository (no .timetree directory)” and terminates. This check prevents accidental execution in invalid directories and ensures that all commands work only in an initialized repository.

## 4.3 Code Fragments

In this subsection, some of the most important algorithms used throughout the project are explored in detail. Code fragments from the three main algorithmic components of TimeTree: the custom SHA-1 hashing implementation, the Myers O(ND) diff algorithm, and the rolling-checksum-based delta engine. The aim here is to show how the theoretical ideas discussed earlier are made into Kotlin code.

### 4.3.1 Incremental Hashing

The first fragment shows the custom SHA-1 implementation that is used by TimeTree. It supports incremental updates, block buffering, and produces 160-bit digests compatible with the standard SHA-1. This implementation is used as the basis for namespaced object IDs.

|  |
| --- |
| private fun processBlock(     buf: *ByteArray*,     off: *Int*, ) {     // Prepare message schedule W[0..79] as 32-bit ints     val w = IntArray(80)     var j = 0     // The first 16 words come directly from the block (big-endian)     while (j < 16) {         val i = off + j \* 4         w[j] = ((buf[i].toInt() and 0xFF) shl 24) or             ((buf[i + 1].toInt() and 0xFF) shl 16) or             ((buf[i + 2].toInt() and 0xFF) shl 8) or             (buf[i + 3].toInt() and 0xFF)         j++     }     // Remaining words W[t] = (W[t-3] xor W[t-8] xor W[t-14] xor W[t-16]) <<< 1     while (j < 80) {         w[j] = rol1(w[j - 3] xor w[j - 8] xor w[j - 14] xor w[j - 16])         j++     }      // Initialize working vars     var a = h0     var b = h1     var c = h2     var d = h3     var e = h4      for (t in 0 until 80) {         val (f, k) =             when (t) {                 in 0..19 -> ((b and c) or ((b.inv()) and d)) to 0x5A827999                 in 20..39 -> (b xor c xor d) to 0x6ED9EBA1                 in 40..59 -> ((b and c) or (b and d) or (c and d)) to 0x8F1BBCDC.toInt()                 else -> (b xor c xor d) to 0xCA62C1D6.toInt()             }         val temp = rol5(a) + f + e + k + w[t]         e = d         d = c         c = rol30(b)         b = a         a = temp     }      // Add this chunk's hash to the result so far:     h0 += a     h1 += b     h2 += c     h3 += d     h4 += e } |

This method is the core of the SHA-1 implementation used in TimeTree. It processes a single 512-bit block by first expanding it into an 80-word message schedule and then running the 80 compression rounds that update the internal state (h0…h4). The four when branches correspond to the standard SHA-1 boolean functions and round constants. After the loop, the working variables are added back into the state, which accumulates the hash over all processed blocks. Thanks to this implementation, the deterministic object IDs are computed without depending on external cryptographic libraries.

### 4.3.2 Myers Diff - Core Edit Computation

TimeTree uses a custom implementation of the Myers O(ND) diff algorithm to compute the shortest edit script between two versions of a file. The algorithm works over lines and produces a minimal sequence of insertions, deletions, and keeps (unchanged lines), which is later formatted as a unified diff. The following fragment illustrates some of the code of the algorithm.

|  |
| --- |
| fun computeEdits(    a: *List*<*String*>,    b: *List*<*String*>, ): List<DiffAlgorithm.Edit> {    val n = a.size    val m = b.size    val maxD = n + m     // Track furthest x-coordinate reached on each diagonal    val offset = maxD + 1    val vSize = 2 \* maxD + 3    var v = IntArray(vSize)    var newV = IntArray(vSize)    v[offset + 1] = 0     // Store step information for each d-iteration    val trace = mutableListOf<MutableMap<Int, Step>>()     for (d in 0..maxD) {        val stepsForD = mutableMapOf<Int, Step>()         for (k in -d..d step 2) {            // Choose whether to move down (insert) or right (delete)            val fromDown = (k == -d) || (k != d && v[offset + k - 1] < v[offset + k + 1])             val prevK = if (fromDown) k + 1 else k - 1            val startX = if (fromDown) v[offset + k + 1] else v[offset + k - 1] + 1            val startY = startX - k             // Extend along matching lines            var x = startX            var y = startY            while (x < n && y < m && a[x] == b[y]) {                x++                y++            }             newV[offset + k] = x             stepsForD[k] =                Step(                    k = k,                    x = x,                    y = y,                    prevK = prevK,                    prevX = startX,                    prevY = startY,                )             // Check if we've reached the end            if (x >= n && y >= m) {                trace.add(stepsForD)                return backtrack(a, b, trace, d, k)            }        }         trace.add(stepsForD)        val temp = v        v = newV        newV = temp    }     return emptyList() } |

This function implements the core of the algorithm. For each edit distance d, it iterates over diagonals k and decides whether to advance from “down” (an insertion) or “right ” (a deletion) based on which previous diagonal reaches further. From that decision point, it extends a diagonal ”snake” over matching lines (while (a[x] == b[y])). The furthest x value per diagonal is stored in the v arrays, while trace records the steps needed for reconstruction. Once both sequences are fully covered (x >= n && y >= m), the function stops and backtrack produces the final list of Edit operations that are later used to create the unified diff by TimeTree.

### 4.3.3 Rsync-Style Delta Computation

TimeTree uses a rsync-style delta engine to synchronize a target file against a basis using rolling checksums and strong hashes. The core of this algorithm is the makeDelta method, which scans the target stream with a sliding window, matches blocks against the basis signature, and emits a sequence of Copy and Insert operations. The following fragment shows the main delta computation loop.

|  |
| --- |
| override fun makeDelta(    target: *InputStream*,    sig: *Signature*, ): Delta {    validateBlockSize(sig.blockSize)     if (sig.blocks.isEmpty()) {        return makeDeltaForEmptySignature(target, sig.blockSize)    }     val weakMap = buildWeakMap(sig)    val ops = mutableListOf<DeltaOp>()    val roller = RsyncRoller(sig.blockSize)    val window = RingBuffer(sig.blockSize)    val pendingInsert = ByteAccumulator()     val firstByte = target.read()    if (firstByte < 0) {        return Delta(sig.blockSize, emptyList())    }     initWindowWithByte(firstByte.toByte(), window, roller)    fillWindow(target, window, roller, sig.blockSize)     while (true) {        val weak = roller.weak()        val match = findMatch(window, weak, weakMap)         if (match != null) {            flushPending(pendingInsert, ops)            ops.add(DeltaOp.Copy(match.index.toLong() \* sig.blockSize, window.size()))             window.clear()            val nextByte = target.read()            if (nextByte < 0) break             initWindowWithByte(nextByte.toByte(), window, roller)            fillWindow(target, window, roller, sig.blockSize)             if (window.isEmpty()) break        } else {            pendingInsert.add(window.get(0))             val nextByte = target.read()            if (nextByte < 0) {                for (i in 1 until window.size()) {                    pendingInsert.add(window.get(i))                }                break            }             window.add(nextByte.toByte())            roller.roll(nextByte.toByte())        }    }     flushPending(pendingInsert, ops)    return Delta(sig.blockSize, coalesceOps(ops)) } |

This method implements the main rsync-style delta algorithm in the project. It starts by validating the block size and building a weak-hash index over the basis blocks (buildWeakMap). A rolling window and RsyncRoller are then initialized over the target stream using initWindowWithByte and fillWindow. For each window position, the algorithm computes a weak checksum, looks up candidate basis blocks in weakMap, and verifies them with the SHA-1 hash inside findMatch. On a successful match, any accumulated literal bytes are flushed as a single Insert, and a Copy operation referencing the matching basis block is appended. On a miss, the oldest byte in the window is buffered into pendingInsert, the window is advanced by one byte, and the rolling checksum is updated in O(1) time. At the end of the stream, any remaining window bytes are emitted as a final Insert, and coalesceOps merges adjacent Copy operations to keep the delta compact.

## 4.4 Interesting Design Patterns

Beyond the algorithms, TimeTree uses several small but deliberate design patterns to keep the implementation simple and testable. The command-line layer is structured as a set of commands, the core algorithms are exposed through strategy interfaces, and many operations are written as pure functions over immutable data. All of this makes it easier to extend and program, and to test each behavior of each part of the project in isolation.

### 4.4.1 Command-Based CLI

The CLI interface follows a simple command pattern. Each command is implemented as a small class. For example, init, status, commit, and diff are all separate command classes that extend a common base from Clikt and override a run() method. The classes are responsible only for parsing the user input and delegating to the core services. This keeps the CLI thin and allows internal logic to improve and evolve without changing what the user sees and experiences.

### 4.4.2 Strategy Interfaces for Hash, Diff, and Delta

The implementation of hashing, diffing, and delta compression is structured around interfaces. The hashing layer is exposed via HashingAlgorithm, the diff engine via the DiffAlgorithm, and the delta engine via the DeltaAlgorithm abstraction.

TimeTree’s current implementation uses:

* SHA-1 as the implementation of HashAlgorithm
* RsyncDelta as the implementation of DeltaAlgorithm
* The Myers diff engine as the concrete DiffAlgorithm

Because the core packages depend on these abstractions instead of concrete classes, it is possible to replace them with alternative algorithms like a different hash, different diff algorithms, or a different delta scheme without changing the rest of the project. This also makes it easy to test each algorithm alone.

### 4.4.3 Purity and Testability

Wherever possible, TimeTree’s core logic is implemented as pure functions that take inputs and return outputs without any hidden side effects. The Myers diff engine operates on a list of strings and returns an edit script. The rolling checksum and delta engine take byte arrays or streams and produce Delta objects. The object store takes serialized bytes and returns object identifiers. All filesystem and process-specific details are pushed to the edges of the system. This separation makes unit tests easy and simple to write and reason about, and also ensures the deterministic behavior across platforms.

## 4.5 End-to-End Execution Flow: init Command

Here, I will show how the different implementation layers of the project work together on a simple operation - the init command. The init command turns a normal directory into a TimeTree repository.

When a user runs tt/timetree init in a directory, the command-line layer parses the subcommand name (init), validates that there are no unsupported flags, and then calls the run() method of the corresponding Init command class. The command does not implement any logic by itself. It is delegated to the core layer, where the rules for the repository creation are.

There, the repository file checks whether the current directory already contains a .timetree folder. If such a folder exists and has a valid structure, the command outputs that the directory is already initialized and has a valid structure without modifying anything. If a broken repository is found, the command repairs the repository. In a normal case, where no repository exists, the file proceeds to create the on-disk layout.

The storage sublayer then creates the internal .timetree directory and its subdirectories like objects/ and refs/, together with any configuration and reference files. The default branch reference and HEAD file are written using the same safe-write pattern used everywhere in the project. This makes the initialization step crash-safe. The directory is either a valid repository or left untouched, but never half-initialized.

Finally, the command prints a short confirmation message to the user indicating that a new TimeTree repository has been created in the current directory. From this point on, all the other commands rely on the presence of the .timetree directory to detect that they are operating inside a repository. This example showcases how the CLI layer, the core layer, and the storage sublayer cooperate to perform a single operation.

# **5. Testing**

The testing for TimeTree focuses on two main points. The first one is whether the system produces the correct results for its core operations, and secondly, whether it behaves predictably when given invalid or unexpected input. Since the project uses three algorithms (SHA-1, Myers diff, rsync-style delta) with a repository model and command-line interface, the testing is split between algorithm-level checks and end-to-end repository scenarios.

This approach combines unit tests, property-based tests, integration tests, and acceptance tests. Unit and property-based tests validate core algorithms for correctness and invariants, including hashing, diff, delta computation, and content-address storage. The integration tests exercise complete flows like “init -> add -> commit -> status -> log -> diff” and check that the on-disk state of the repository matches the expected object graph. The acceptance tests are defined in relation to the project's functional requirements and focus on realistic scenarios, such as initializing a repository, adding files, committing changes, and inspecting the history.

All tests are written in Kotlin using Kotest on top of the JUnit 5 platform. These tests run via the Gradle wrapper, and the coverage report feature of the IDE is used to ensure that the code is tested. In addition to automated tests, manual exploratory testing is done on different operating systems to confirm that the CLI interface behaves consistently across environments and that error messages are clear when commands are run in invalid contexts - like outside of the repository or with missing files.

The subsections that follow will describe the testing strategy in more detail, starting with algorithm-level unit and property-based tests, then moving to repository integration tests and acceptance tests for the main functional requirements.

## 5.1 Test Environment and Organization of the Test Suite

All the tests for TimeTree are written in Kotlin and run on the same JVM platforms as the main application. They are executed using the Gradle wrapper, which invokes the Junit 5 platform with Kotest as the testing framework. This setup keeps the environment that we have for the tests the same as the production one. The same JDK version, the same project structure, and the same build configuration are used for both running the application and running its tests. The tests can be run both from the command line via ./gradlew test and from the IDE, which also provides basic coverage information to verify that the code is exercised.

The tests are organized by package to mirror the structure of the main source code. Low-level units such as the SHA-1 implementation, the Myers diff engine, the rolling checksum, and delta logic live under dedicated test packages that correspond to:

* core.hash
* core.diff
* core.delta

Higher-level tests for command handling are placed under a test package that corresponds to the CLI layer. This mirroring makes it clear which parts of the system are covered by which tests and simplifies navigation when adding new ones.

Kotest is used in two different ways. For most components, tests are written in a straightforward “example-based” style, where each test case exercises a specific scenario with an explicit expected result. For ones with stronger invariants like diff and delta, Kotest’s property-based testing support is used to generate multiple random inputs and check that certain properties always hold. Temporary directories and files are created during integration tests to avoid pollution, and they are cleaned up automatically at the end of each test. Together, all of this ensures that tests are easy to run and easy to extend.

## 5.2 Unit Tests for Core Algorithms

Not a small part of the testing done in the project is directed at verifying the correctness of the core algorithms and low-level utilities on which all higher-level behavior depends. These components are tested in isolation under the core test packages. The goal of these tests is to ensure that hashing, diff, delta computation, and object identifier construction behave exactly as specified, before they are composed into repository operations and CLI commands.

### 5.2.1 SHA-1 Implementation

The custom SHA-1 implementation is tested in the Sha1Test file. The test uses a collection of well-known SHA-1 vectors such as the empty string, short strings like “abc”, and longer messages. For each case, the test computes the hash using the Sha1 class and compares the result against the expected 160-bit digest shown in hexadecimal form. This verifies that the compression function, padding logic, and big-endian length encoding are implemented and work correctly.

In addition to the fixed test vectors, the tests also check the incremental API. The same message is fed into the hasher both as a single update call and as a sequence of smaller chunks, with the final digest compared in both cases. This ensures that the internal buffering logic behaves correctly across boundaries and that partial updates do not affect the final result. Tests also confirm that calling digest() resets the internal state so that the same Sha1 instance can be reused safely for multiple messages without leaking state between calls.

### 5.2.2 Myers Diff Engine

The Myers diff implementation is tested in the MyersTest file. These tests work on lists of strings and are focused on verifying that the computed edit script is both correct and minimal for small inputs. Examples of simple scenarios are the insertion, deletion, and update of individual lines and the cases when the two lists are equal or totally different. In both cases, the test takes the returned sequence of Edit operations and applies them to the initial list and asserts that the result is identical to the target list.

To practice more edge cases, the tests also cover situations where changes occur at the beginning or end of the file, and where several unchanged regions are separated by small blocks of edits. In these cases, the expected behavior is that Myers finds long “snakes” of unchanged lines and produces a compact sequence of edits rather than many small changes. The tests additionally verify that the algorithm gracefully handles empty inputs and does not throw when the lists have very different lengths.

### 5.2.3 Rsync-Style Delta Engine and Supporting Utilities

The rsync-style delta engine and its helper classes are tested in the core.delta package. The main behavior of the delta algorithm is validated in the RsyncDeltaSpec test file. Each of the tests creates a basis byte array and a modified target version, runs makeSignature on the basis and makeDelta on the target, and then applies the resulting Delta back to the basis. The reconstructed output is compared byte-for-byte with the original target. This invariant is tested across several scenarios. Small text files with local edits, files where data is only added, and binary data with changes around block boundaries.

Other tests in the package focus on the supporting utilities. The RingBufferSpec file verifies that the ring buffer used by the rolling checksum behaves like a fixed size sliding window, always returning the bytes in the expected order as elements are added and dropped. The ByteAccumulatorSpec file checks that literal bytes are collected and flushed correctly into Insert operations without losing or reordering data. The VarIntSpec file, the SignatureIOSpec, and the DeltaIOSpec files ensure that variable-length integers, signatures, and delta streams can be encoded to and decoded from their binary representation without loss, and that malformed inputs are rejected with clear errors.

Together, these tests give reassurance that the rolling checksum, block indexing, and delta encoding are implemented correctly and that the engine will not corrupt data when used by higher-level commands.

### 5.2.4 Object Identifiers and Domain Separation

The creation of the object identifiers and the domain separation rules are tested in the DomainSeparationTest file under core.objects package. The file verifies that namespace prefixing prevents collisions with git-style object IDs by confirming that the same content produces different hashes when tagged with TimeTree's domain prefix versus git's standard header. This prevents collisions between different object types and mirrors the behavior expected from a content-addressed store.

Additional checks are created to ensure that object identifiers are stable. Serializing and hashing the same logical object in different runs get the same identifier, while any changes in content result in a different identifier. This is important not only for correctness, but also for reproducibility across different operating systems. By testing the object ID logic independently of the CLI and repository code, these tests ensure that the foundation behind the content-addressed storage sublayer is correct before they used by higher-level operations like commits and diffs.

## 5.3 Integration Tests for Repository Behavior

While the core algorithms are tested separately, the validity of TimeTree as a version control system depends on how these components work together when they are called from the command-line interface. They are located in the cli.commands package. They create temporary repositories, invoke commands such as init, add, commit, status, log, diff, branch, and checkout, and then assert both on the command output and on the on-disk .timetree layout. Those tests are closer to how a real user interacts with the program and are used to validate repository state and error handling.

### 5.3.1 Repository Initialization and Detection

InitCmdTest and parts of the StatusCmdTest cover the behavior around repository creation and detection. The tests usually start from an empty temporary directory, run the init command, and check that a .timetree folder has been created with the expected subdirectories, objects, and refs, for example. In addition to this, they also verify that after initialization, running status returns a clean repository with no commits and no tracked files.

Additional tests cover repeated and invalid initialization attempts. Running init a second time in an already initialized directory will succeed without modifying the existing history and print a message that the repository is reinitialized. If the .timetree directory exists but its internal layout is missing something or anything is corrupted, the command will repair the structure by adding the missing or recreating the files. The tests assert that after re-running init, the repository layout matches the structure. This ensures that the repository detection logic and repair one is correct, and the repository cannot be left in an unusable state.

### 5.3.2 Staging, Committing, and Inspecting History

The typical version control workflow - adding files, committing them, and inspecting them is tested in the files AddCmdTest, CommitCmdTest, StatusCmdTest, and LogCmdTest. The tests create one or more files in a temporary repository, run add to stage them, and then run commit with a message. After this is done, they assert that status reports a clean working tree, that the log shows a new commit with the correct message and identifier, and that blob and tree objects have been written under ./timetree/objects for the staged files.

Further tests are done to cover multiple commits and modifications over time. For example, tests create an initial commit, modify an existing file, and commit again, then check that log shows two commits in the expected order and that diff between them reflects only the changes made. There are also tests that test behavior when we have no staged changes. In such cases, commit should refuse to create an empty commit if the flag that allows an empty commit is entered and instead should display a message indicating that there is nothing to commit. Together, all these tests confirm the staging area, commit creation logic, and history view all work correctly and as expected.

### 5.3.3 Diff, Checkout, Branch, and Delta Commands

Commands that operate on existing history and perform more complex tasks are tested in DiffCmdTest, PatchCmdTest, BranchCmdTest, CheckoutCmdTest, SigCmdTest, and DeltaCmdTest. DiffCmdTest creates two commits with differences and checks that diff produces a unified diff consistent with the edits. The unchanged lines are preserved as context, insertions and deletions are marked with the expected prefixes, and the hunk headers match the line ranges being compared. The PatchCmdTest is used for the binary delta compression. This tests both the diff engine and the patch application logic.

Branching and checkout behavior is tested by BranchCmdTest and CheckoutCmdTest. They create new branches from existing commits, switch between branches, and ensure HEAD and the working tree reflect the selected branch. The behavior we are expecting is that creating a branch does not lose existing commits, and that checking out a branch gets the correct snapshot of the tree and updates log to show the appropriate history. Finally, SigCmdTest and DeltaCmdTest test the rsync-style commands. They generate signatures and deltas for the basis/target pairs and then verify that applying the delta again creates the target byte-for-byte. All these tests validate that the higher-level commands work correctly and all the algorithms are working correctly as well.

## 5.4 Acceptance Testing of User-Facing Scenarios

Beyond unit and integration tests, TimeTree was validated through a small set of acceptance tests written from a user’s point of view. Each of the following scenarios is defined by its preconditions, the commands executed, and the expected behavior. They also cover the most important functional requirements of the system and are marked as passed.

**AT-1**: Initialize a new repository - Passed

**Preconditions**: An empty directory

**Steps**: Run tt init, then tt status

**Expected** **result**: A .timetree directory is created with the correct layout. tt status returns a clean repository with no commits and no tracked files. Running tt status or tt log in a directory without .timetree outputs a clear error message that the directory is not a TimeTree repository and exits.

**AT-2:** Add and commit a file - Passed

**Preconditions:** Repository initialized as in AT-1

**Steps:** Create a file, run tt add <file name> or just a “.”, then tt commit -m “some message ”, followed by tt status or tt log.

**Expected result:** tt commit succeeds and prints the commit identifier. tt status reports a clean working tree. tt log shows one commit with the correct message. A blob, a tree, and a commit object exist under .timetree/objects, and repeating tt init shows a reinitialized message.

**AT-3**: Modify a file and inspect the diff - Passed

**Preconditions:** Repository with at least one commit as in AT-2

**Steps:** Edit the file, run tt status, then tt diff to see the differences

**Expected result:** Before committing, tt status shows the file as modified. tt diff between the working tree and HEAD shows a unified diff where unchanged lines appear as context, added lines are shown with +, and removed lines with -.

**AT-4**: Create and switch branches - Passed

**Preconditions:** Repository with at least one commit as in AT-2

**Steps:** Run tt branch <branch name> or tt checkout <branch name>, make a new commit, and then run tt checkout master to go back and tt log --all to show all commits in all current branches.

**Expected result:** After tt checkout <branch name>, new commits will only appear in this branch. Returning to master restores the state that master has, which does not include the other branch commits. The commits of either branch are not lost when switching between them.

**AT-5**: Prevent committing with no staged changes - Passed

**Preconditions:** Repository with at least one commit as in AT-2

**Steps:** Run tt status to confirm that there are no staged, modified, or untracked files. Run tt commit -m “some message” and then then tt log to see the result.

**Expected result:** tt commit should not create a new commit and just prints a message saying that there are no changes to commit. The commit history shown by tt log is unchanged. We have the last commit (the first one).

## 5.5 Functional Requirements Testing

The functional requirements are tested by the unit tests in the core package, the command-level integration tests in the cli.core package, and the acceptance scenarios mentioned above. Overall, the algorithm tests support the requirements around data integrity and deterministic behavior. The command tests and acceptance scenarios cover the repository workflows. From initializing a repository, staging and committing changes, inspecting history, viewing diffs, working with branches, and preventing invalid operations such as running commands outside of the repository. Taking all of them, we can see that all of the functional requirements have been thoroughly tested and verified.

## 5.6 Test Coverage

To get a better professional view of how much the code is covered by the tests, I used IntelliJ IDEA’s built-in coverage runner. The overall result is 90.5% class coverage, 84.3% method coverage, 72.8% branch coverage, and 84.6% line coverage for the project as a whole. This confirms that a huge part of the project and the most important parts are tested and covered.

The core algorithm packages have a very high coverage. The core.hash, core.diff, and core.delta all reach 93-100% method and line coverage. The other packages also reach or are close to 100% percent. This is a very good result for a project of this scale. The following figure shows exactly the results of those tests taken directly from IntelliJ’s built-in coverage HTML report.

A screenshot of a computer

AI-generated content may be incorrect.

*Figure 12 - Coverage Summary Report*

# **6. Results and Conclusion**

This section summarizes what I have achieved with TimeTree, how the final system compares to what I initially planned, and what I have learned from building it. The project delivered a working local version control system with its own content-addressed storage, custom implementations of SHA-1, Myers diff, and a rsync-style delta algorithm with a small CLI interface. TimeTree can initialize repositories, track files, create commits, inspect history, view diffs, and move between branches.

Functionally, the complete version provides:

* init to create or reinitialize a .timetree repository and repair missing files.
* add and commit to stage files, create blobs, trees, and commits
* status and log to inspect the working tree and the commit history
* diff to compare files using the custom Myers diff implementation and show unified +/- hunks.
* branch and checkout to create and switch between branches, moving HEAD to point to the latest commit of the branch.
* hash-object to expose and compare TimeTree and Git-style object IDs.
* sig, delta, and patch to generate signatures, compute deltas, and reconstruct files using the rsync-style engine.

## 6.1 Comparison with Initial Goals

At the start of the project, my main objective was to open the black box that Git is, since I use the tool every single day as a developer. Instead of memorizing commands like add, commit, and stash, I wanted to understand what really happens behind them. How content-addressed objects are stored, how trees and commits are built, how HEAD and branch references work, and how they all interact in such a huge system as Git. The goal was simple enough to build a small, working system that mirrors those core mechanics that Git has. Blobs, trees, commits with parents, a staging area, and symbolic refs, all implemented by me.

Alongside the learning goal, I also wanted TimeTree to be like a platform for experimentation. One of those things is that the project has a deterministic object storage that produces the same repository state across platforms. Another is that the project uses delta compression using rolling checksums to optimize file synchronization. Kotlin was the language I chose to create a project that can complete all of the mentioned things.

The final version of the project matches these goals. TimeTree implements its own content-address storage, staging index, deterministic tree creation, commit objects with parents, and symbolic refs, all powered by my algorithms that have been implemented from scratch. Namely, the SHA-1, the Myers diff, and the rsync-style delta algorithm. They are used and wired to real commands such as diff, hash-object, sig, delta, and patch. The repository layout is fully transparent on disk, and all of the files can be inspected.

Overall, the result is what I imagined the project to be. I have a small, working version control system that I understand the magic behind, and a codebase that is fully ready to be scaled and continuously expanded.

## 6.2 What Works Well and What Does Not

Looking from a practical point of view, the core workflow works as expected and is reliable. Initializing a repository with tt init, adding files, committing them, inspecting the history with tt log, and the state of the repo with tt status all work as expected in a day-to-day use. The repository is deterministic. Running the same commands on the same files always creates the same object IDs and tree/commit structure. The algorithms also work as expected. The custom SHA-1 passes all test vectors and incremental-update tests. The Myers diff creates readable unified diffs for both small and large files, and the rsync-style delta engine can reconstruct a file byte-for-byte from a basis and a delta. All of the tests, combined with the coverage report, additionally show that the behavior of the code holds across a wide range of inputs and not just hand-picked examples.

However, at the same time, there are clear limitations and gaps. Firstly, TimeTree is strictly local. There is nothing close to the concept of having remote functionality. There is no pushing, pulling, or fetching from a hosted server or the repo being hosted on GitHub. Branching and checkout work, but there is no merge command or any type of conflict resolution support. This means that there is not really a way to merge branches and resolve conflicts in general, since they cannot happen. The CLI is intentionally small, but that can still be viewed as a gap. There is no TUI (Terminal User Interface) or GUI (Graphical User Interface) to improve the experience of the user. Nevertheless, I have added color output to some of the commands to increase readability and visibility and to mimic Git’s behavior.

## 6.3 Problems Encountered and How They Were Resolved

Like any project, building TimeTree had its technical and design problems that were not obvious at the beginning and the initial plan. Most of them came from implementing the algorithms correctly, keeping the repository safe on the disk, and integrating the rsync algorithm into the project.

**SHA-1 edge cases and correctness**

The first version I had for the SHA-1 produced correct hashes for short inputs but failed for longer messages and certain chunk boundaries. The problem was happening because of incorrect padding and how the total bit length was encoded in the final 64 bits. I managed to fix this problem by lining the code up against the official SHA-1 test vectors, by writing tests for incremental updates with random chunk sizes, and refactoring the padding logic into a helper. After the code passed all the tests, then I it became stable enough to be used for the object IDs.

**Myers diff backtracking and off-by-one errors**

Understanding the Myers O(ND) algorithms from the paper was not a problem, and it was straightforward for the forward pass, but it became harder for the backtracking. The early versions of the code created edit scripts that we correct most of the time, but they happened to have a mistake, such as misplacing the first or last changed line in a hunk. The property-based tests that I created for the algorithm helped reveal this problem. The fix was to make the backtracking step explicit and reconstruct each snake of unchanged lines, rather than relying on assumptions.

**Rsync delta window management and DeltaStore integration**

The rsync-style delta engine was originally implemented as a standalone algorithm. When I integrated it into the store-to-store large blobs as deltas, several bugs appeared. The early versions sometimes dropped or duplicated bytes when a match occurred near block boundaries or produced suboptimal chains that were too deep. I managed to fix this by introducing small, focused helpers with their own tests and by adding integration tests that wrote a sequence of large blob versions through DeltaStore. After this, the code worked correctly, and the algorithm was completed.

**Repository repair semantics for init**

One design decision problem arose when I was wondering what should happen when tt init is run in a directory that already contains a ./timetree directory. It was not a big problem, but still something that posed me a question. A normal approach would be to simply fail with an error, but this would not be like how Git does it. In that regard, I decided to go with how Git does this by simply reinitializing the repository and repairing it if there are any errors with the structure. If there are any problems found or if any files are malformed, the code would repair them as needed. The test I have created covers these scenarios, so the code always works, and the command never leaves the repository in a broken state.

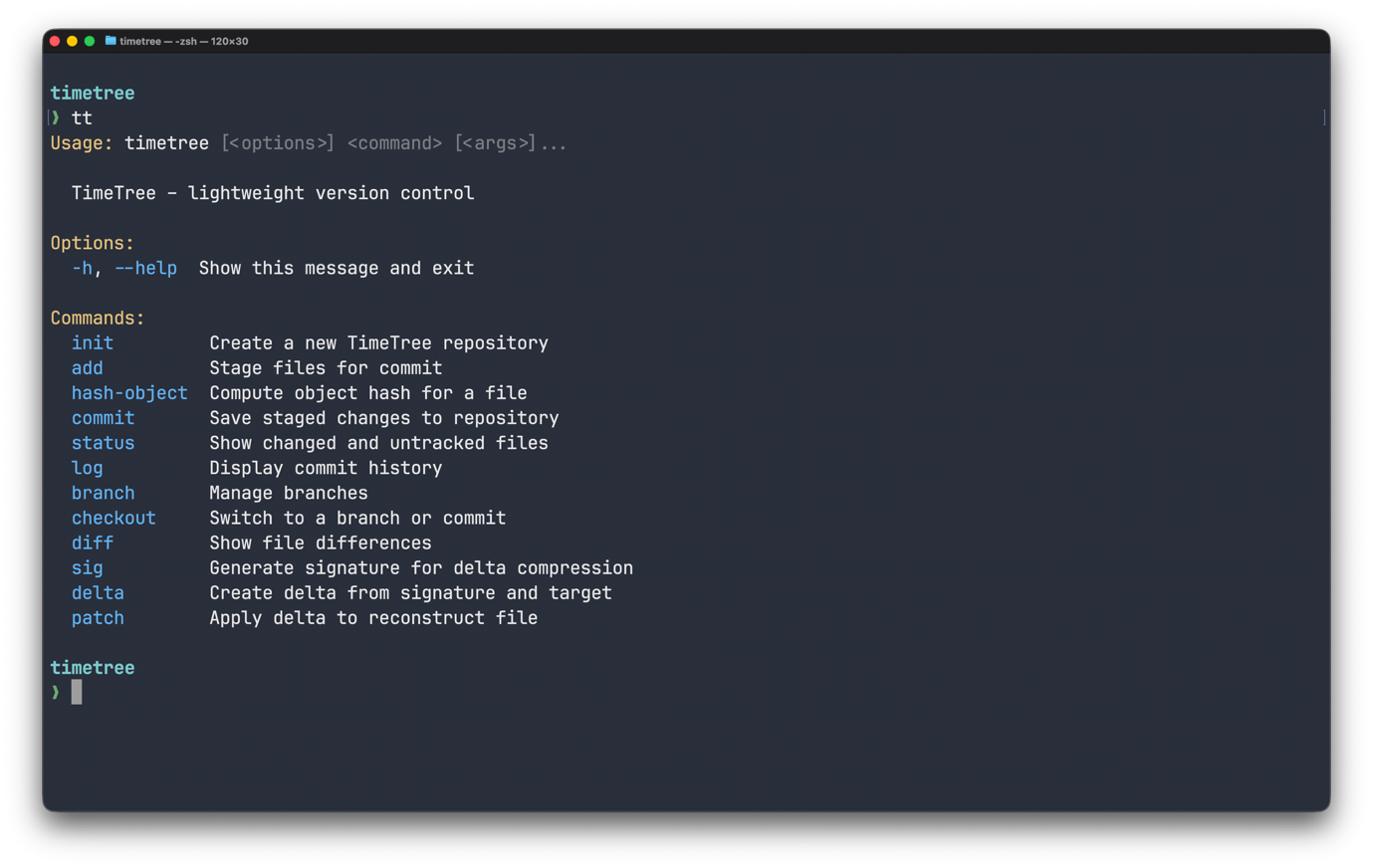
**Path normalization and determinism**

To keep the object IDs deterministic across different operating systems, paths inside trees, and the index must be normalized. Earlier, I had a mix of platform-specific separators and relative path handling that could have produced different tree objects for the same logical structure. I fixed this by applying POSIX-style / separators for the project, and that problem was fixed. Now everything runs smoothly on all operating systems.

These problems shaped the final architecture in important ways. In almost every case, the solution followed the same pattern: isolate the tricky component, write focused tests that cover edge cases, and only then integrate it back. This test-driven approach not only caught bugs early but made the codebase more maintainable.

## 6.4 Screenshots and Example Workflows

This subsection shows how TimeTree looks in practice. All screenshots are taken from a terminal session on my own machine (macOS). They illustrate how the commands are and what a typical workflow is. Essentially, they show all of the aforementioned arguments in practice.



*Figure 13 - Start menu of TimeTree*

This screenshot shows what the tt or timetree command does without arguments. It shows the full list of available subcommands and options.

A screenshot of a computer

AI-generated content may be incorrect.

*Figure 14 - Initializing a repository*

This screenshot shows what running tt init does in an empty repository. It creates the .timetree folder and prints a confirmation message with the repository path.

A screenshot of a computer program

AI-generated content may be incorrect.

*Figure 15 - First commit and log*

*A screenshot of a computer program

AI-generated content may be incorrect.*

Figure 16 - Detecting and showing changes

A screenshot of a computer

AI-generated content may be incorrect.

*Figure 17 - Branching and switching*

A screenshot of a computer program

AI-generated content may be incorrect.

*Figure 18 - Help and error output*

In the first two of the four screenshots, we can see a typical workflow of the project. In Figure 15, we can see the commands add, commit, and log in action. We create the file, add it to the staging area, commit the file, create a commit, and finally show the commit history of the branch. In Figure 16, we see how we make a change to the same file, and then using tt status shows that a change has been made in the working directory. Following this with tt diff, we see what the changes are.

Continuing with Figure 17, we can see how the branch and checkout commands work. With branch, we can create a branch, and with checkout, we can switch to it. This can also be simplified by just running checkout with the -b flag. With Figure 18, we can see how the program uses the --help flag of each command, showing a simple message of what the command does and its available flags that can be used. In addition to this, there is also automatic error detection when we make a typo in the commands. The code catches this and shows a suggestion of the command that we might have had in mind.

Finally, the claims that the operations are near instantaneous are verified. I have tested them by creating two scripts for benchmarking. The first is for a quick look, and the second is a bigger one, which takes between 5-10 minutes. For example, adding 100 files is ~257ms, committing 100 files is ~196ms, and testing status with 100 files is ~199ms. The bigger tests test more files. There, checking out 1000 files is ~170ms, logging 1000 commits is ~0.353ms, and finally, the most time-consuming is creating 1000 commits, which took me around ~6 minutes. In real usage, developers make 5-10 commits per day, not 1000 in sequence.

## 6.5 Missing Features and Future Development

TimeTree is deliberately built to be with a small, understandable core, so a number of features that are standard and an absolute must in mature VCS tools are not yet implemented. The most important missing things are also the most obvious and natural pieces to implement for future work.

On the **repository side**, the biggest gap is the lack of merge and conflict-resolution support. We already have branching and checkout, diff engine, and object model, so the next step would be to implement the merge command and be able to essentially merge different branches. Another major step for improvement is basic remote synchronization. This means to be able to push, pull, and fetch commits between repositories, together with a simple protocol, authentication model, and safety checks. Other improvements may also include log graphs and commit statistics. In a sense, to implement a way to mimic the push, pull, and fetch methods of Git.

On the **storage and algorithm side**, there is also room to extend TimeTree with more advanced repository maintenance tools. While DeltaStore already provides delta compression, Git-style monolithic pack files could be added for network transfer efficiency. Another improvement is to improve the hashing that we have. We can introduce stronger hashing algorithms, such as SHA-256, which is gradually being adopted by Git as well. The same goes for the inclusion of other diffing algorithms, like Patience diff, which is in Git.

On the **user-experience side,** TimeTree currently has a clean but minimal CLI. Future improvements here could include interactive staging (like tt add -p), shell completion scripts, and possibly a small text-based UI for browsing history and diffs. Here, even creating a GUI interface is a possibility as well. Further, a system configuration file like .gitconfig for usernames, aliases, or defaults is a good addition as well. Adding all those features would make the experience users get using TimeTree a lot more like how they would typically be using Git.

Overall, there are a lot of improvements to be made. In a sense, they are endless, since Git itself is still improving to this day, and we only mimic its basic behavior. Of course, adding more commands and refining the current ones to include more flags is by no means ignored. What I want to advocate is that TimeTree’s layered architecture makes it straightforward to add these features incrementally. The foundation is solid and ready for extension.

## 6.6 Conclusion and Personal Reflection

Looking back at the project, TimeTree did what I wanted it to do. It turned from a black box version control tool to something that I can build and talk about myself. I now have a working local VCS that uses content-addressed storage, a staging index, deterministic trees and commits, symbolic references, and implementations of SHA-1, Myers diff, and a rsync-style delta engine. It is not a replacement for Git and is not as feature-rich as Git, but it’s a coherent, test-backed system that has the original mechanics that I wanted to understand. The fact that the core workflow init -> add -> commit -> status -> log -> checkout works with all the algorithms is a great result I’m proud of.

On the technical side, the project gave me a deeper understanding of how a VCS actually works. Implementing SHA-1 forced me to think about padding, bit-level operations, and incremental hashing. Myers diff went from a theoretical algorithm from a paper to a piece of code I can debug and extend. Integrating the rsync-style delta engine showed me how rolling checksums, strong hashes, and chain depth limits work. Designing the object formats, making them deterministic, and then validating everything with tests and coverage report changed my thinking from “Git probably does something like this” to “I know how this works!”

From a software engineering perspective, I also learned a lot about process and trade-offs. The project showed me the value of a layered architecture. Keeping the CLI layer separated from the core layer and storage sublayer made it much easier to integrate new features without breaking any functionality. Writing unit tests, property-based tests, and integration tests early paid off significantly. Many of the bugs I found were thanks to the small, focused tests that I had. Choosing not to implement remotes or a full merge engine in this version was a good decision, so that it can become a solid, well-tested system.

Overall, the main outcome of TimeTree is not just the codebase or the CLI tool, but the confidence that I can open complex systems and rebuild them. Now I approach version control, hashing, and differencing as things I can design and argue about, not just use them. This mindset of being able to move between theory, implementation, and testing is the most important result of this project and something that I will carry into future work.

# **7. References**

[1] S. Perez De Rosso and D. Jackson, “What’s wrong with git? a conceptual design analysis,” presented at the Proceedings of the 2013 ACM International Symposium on New Ideas, New Paradigms, and Reflections on Programming & Software, New York, NY, USA: Association for Computing Machinery, 2013, pp. 37–52. doi: 10.1145/2509578.2509584.

[2] M. D. Beckman, M. Çetinkaya-Rundel, N. J. Horton, C. W. Rundel, A. J. Sullivan, and M. Tackett, “Implementing Version Control with Git and GitHub as a Learning Objective in Statistics and Data Science Courses,” *J. Stat. Data Sci. Educ.*, vol. 29, no. 1, pp. S132–S144, Jan. 2021, doi: 10.1080/10691898.2020.1848485.

[3] E. W. Myers, “An O(ND) difference algorithm and its variations,” *Algorithmica*, vol. 1, no. 1, pp. 251–266, Nov. 1986, doi: 10.1007/BF01840446.

[4] “IEEE/ISO/IEC International Standard for Software, systems and enterprise--Architecture description,” *ISOIECIEEE 420102022E*, pp. 1–74, Nov. 2022, doi: 10.1109/IEEESTD.2022.9938446.

[5] N. I. of S. and Technology, “Secure Hash Standard (SHS),” U.S. Department of Commerce, Federal Information Processing Standard (FIPS) 180-4, Aug. 2015. doi: 10.6028/NIST.FIPS.180-4.

[6] M. Stevens, E. Bursztein, P. Karpman, A. Albertini, and Y. Markov, “The first collision for full SHA-1,” 2017, 2017/190. Accessed: Oct. 12, 2025. [Online]. Available: https://eprint.iacr.org/2017/190

[7] “Git - hash-function-transition Documentation.” Accessed: Oct. 12, 2025. [Online]. Available: https://git-scm.com/docs/hash-function-transition

[8] M. Stevens *et al.*, “Short Chosen-Prefix Collisions for MD5 and the Creation of a Rogue CA Certificate,” in *Advances in Cryptology - CRYPTO 2009*, S. Halevi, Ed., Berlin, Heidelberg: Springer, 2009, pp. 55–69. doi: 10.1007/978-3-642-03356-8\_4.

[9] X. Wang and H. Yu, “How to Break MD5 and Other Hash Functions,” in *Advances in Cryptology – EUROCRYPT 2005*, R. Cramer, Ed., Berlin, Heidelberg: Springer, 2005, pp. 19–35. doi: 10.1007/11426639\_2.

[10] Q. Dang, “Recommendation for Applications Using Approved Hash Algorithms,” National Institute of Standards and Technology, NIST Special Publication (SP) 800-107 Rev. 1, Aug. 2012. doi: 10.6028/NIST.SP.800-107r1.

[11] P. Koopman, “32-bit cyclic redundancy codes for Internet applications,” in *Proceedings International Conference on Dependable Systems and Networks*, June 2002, pp. 459–468. doi: 10.1109/DSN.2002.1028931.

[12] “Git - git-diff Documentation.” Accessed: Nov. 01, 2025. [Online]. Available: https://git-scm.com/docs/git-diff

[13] D. S. Hirschberg, “A linear space algorithm for computing maximal common subsequences,” *Commun ACM*, vol. 18, no. 6, pp. 341–343, June 1975, doi: 10.1145/360825.360861.

[14] A. Tridgell and P. Mackerras, “The rsync algorithm,” 1996. Accessed: Nov. 01, 2025. [Online]. Available: https://www.semanticscholar.org/paper/The-rsync-algorithm-Tridgell-Mackerras/d9695436e01795fa572df1f01d8643056a96f205

[15] S. T. Ahmed and L. E. George, “Lightweight hash-based de-duplication system using the self detection of most repeated patterns as chunks divisors,” *J. King Saud Univ. - Comput. Inf. Sci.*, vol. 34, no. 7, pp. 4669–4678, July 2022, doi: 10.1016/j.jksuci.2021.04.005.

[16] C. Percival, “Matching with mismatches and assorted applications,” 2006. Accessed: Nov. 01, 2025. [Online]. Available: https://www.semanticscholar.org/paper/Matching-with-mismatches-and-assorted-applications-Percival/5e0c4cc10b295b86af0a409899a8ba0a524a9ae8