Li'nage

Advanced Version Control System

Complete Technical Proposal

with Embedded Architecture Diagrams

Li'nage Development Team

December 23, 2025

Abstract

Li'nage is an innovative version control system designed to provide developers with unprecedented insights into code evolution, line-level tracking, and intelligent change detection. This comprehensive technical proposal presents the complete system architecture, including detailed class diagrams, entity-relationship models, sequence diagrams, deployment topology, and implementation strategies. The system implements advanced diff algorithms (Myers, Patient, Minimal), multi-strategy authentication (HTTP, SSH, OAuth), AI-assisted commit tracking, and sophisticated conflict resolution mechanisms.

Document Version: 1.0
Status: Complete Technical Proposal
Classification: Internal
Distribution: Project Stakeholders

Contents

1 Executive Summary

Li'nage represents a paradigm shift in version control systems by providing granular line-level tracking combined with intelligent change detection and visualization. Unlike traditional version control systems that track files as atomic units, Li'nage maintains detailed histories of individual line modifications, enabling developers to understand code evolution at an unprecedented level of detail.

1.1 Key Features

- **Line-Level Tracking:** Track individual line modifications across commits with complete history
- **Intelligent Diff Algorithms:** Choose from Myers, Patient, or Minimal diff strategies based on file characteristics
- **Multi-Strategy Authentication:** Support for HTTP tokens, SSH keys, and OAuth credentials
- **AI-Assisted Analysis:** Detect and track AI-generated code contributions
- **Advanced Conflict Resolution:** Visual conflict detection with merge suggestions
- **Version Graph Visualization:** Interactive DAG visualization of commit history
- **Recovery Capabilities:** Rollback to any previous snapshot with integrity verification
- **Remote Integration:** Seamless integration with GitHub, GitLab, and Bitbucket

1.2 Target Audience

- Individual developers seeking detailed code evolution insights
- Development teams requiring granular change tracking
- Code reviewers needing line-level blame information
- Project managers tracking AI-assisted development
- Research teams studying code evolution patterns

2 System Architecture Overview

2.1 Layered Architecture

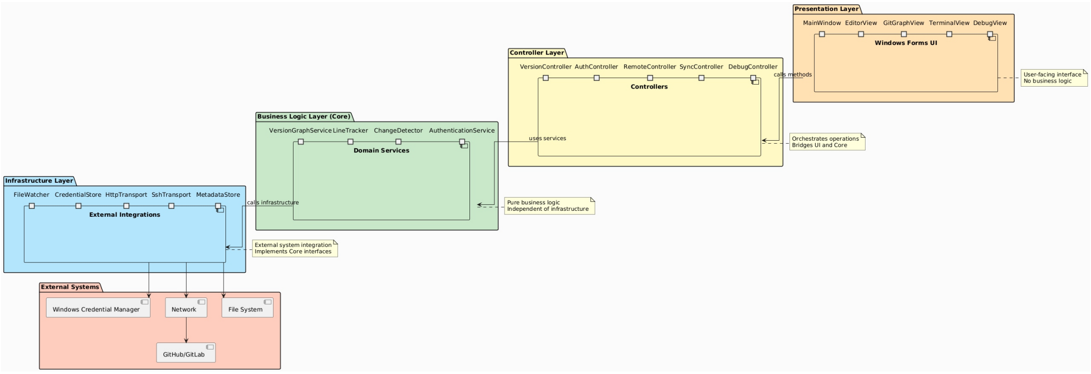

Figure 1: System Architecture - Layered Design with Presentation, Controller, Business Logic, and Infrastructure Layers

Li'nage follows a clean architecture pattern with four distinct layers, ensuring separation of concerns and maintainability:

2.1.1 Presentation Layer (Windows Forms UI)

The presentation layer provides the user interface without containing business logic:

- **MainWindow:** Primary interface for repository management and commit operations
- **EditorView:** Code editing interface with integrated line tracking visualization
- **GitGraphView:** Interactive visualization of the commit DAG (Directed Acyclic Graph)
- **TerminalView:** Command-line interface for advanced operations
- **DebugView:** Development and debugging tools for system analysis

2.1.2 Controller Layer

The controller layer orchestrates operations and bridges the UI with core services:

- **VersionController:** Manages version control operations (commit, branch, merge, rebase)
- **AuthController:** Handles authentication and credential management
- **RemoteController:** Controls remote repository interactions (push, pull, fetch)
- **SyncController:** Coordinates synchronization between local and remote repositories
- **DebugController:** Provides debugging and diagnostic capabilities

2.1.3 Business Logic Layer (Core Domain Services)

The business logic layer contains pure domain logic independent of infrastructure:

- **VersionGraphService:** Manages the commit DAG, branch operations, and history traversal
- **LineTracker:** Tracks line-level modifications using configurable diff strategies
- **ChangeDetector:** Detects file changes and identifies conflicts
- **AuthenticationService:** Manages credential lifecycle and validation

2.1.4 Infrastructure Layer

The infrastructure layer implements external system integrations:

- **FileWatcher:** Monitors file system for changes using Windows file system APIs
- **CredentialStore:** Integrates with Windows Credential Manager for secure storage
- **HttpTransport:** Implements HTTP/HTTPS protocol for remote operations
- **SshTransport:** Implements SSH protocol with key-based authentication
- **MetadataStore:** Persists commit history, snapshots, and metadata

3 Complete Data Model

3.1 Entity-Relationship Diagram

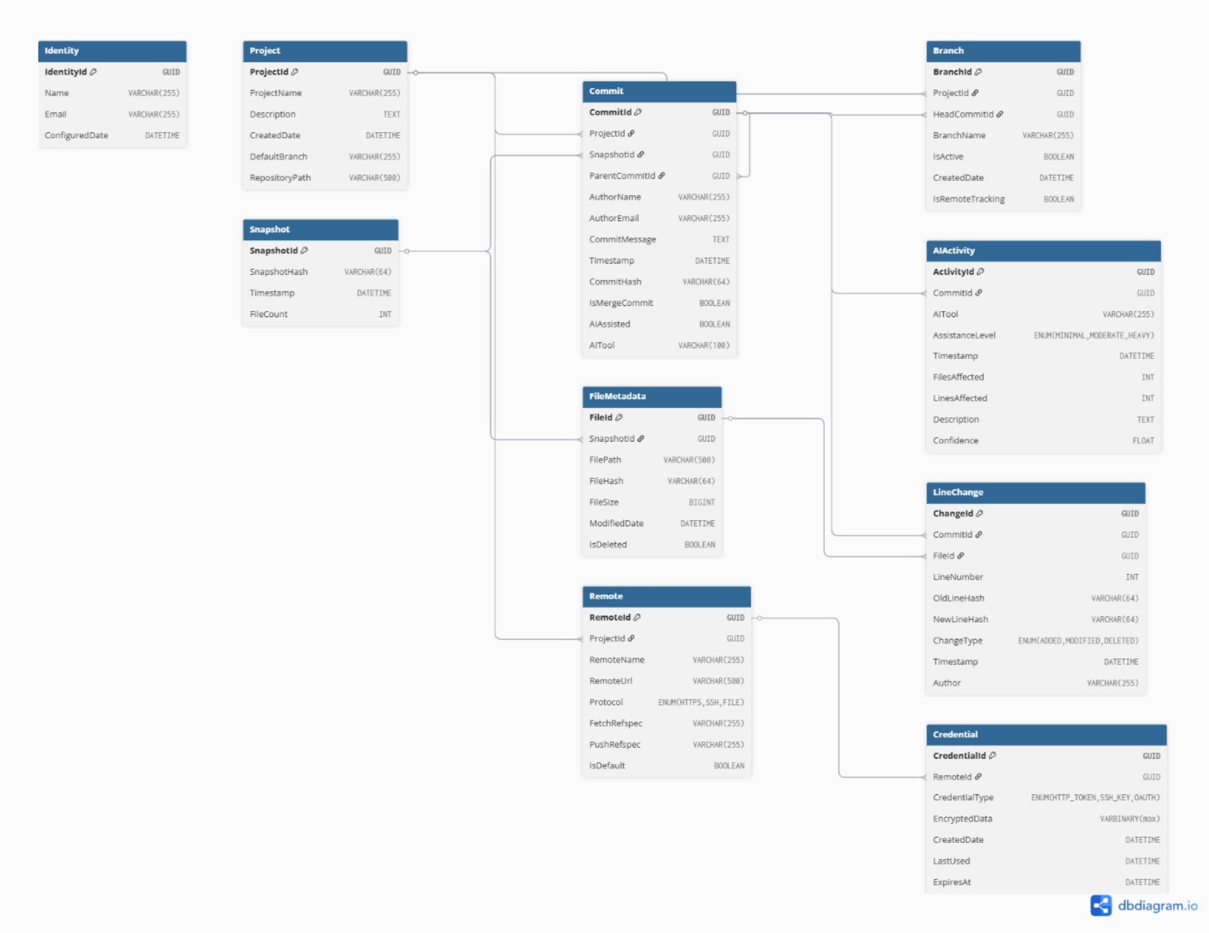

Figure 2: Entity-Relationship Diagram - Complete Database Schema with All Entities, Attributes, and Relationships

3.2 Entity Descriptions

Entity	Description	Key Relations
Project	Root container for version control, stores project metadata and configuration	1:N Branches, 1:N Remotes
Commit	Immutable version snapshot with author, timestamp, message, and parent references	1:1 Snapshot, N:1 Branch, N:N Parent
Snapshot	Complete filesystem state at a specific point in time with file count and hash	1:N FileMetadata

Entity	**Description**	**Key Relations**
FileMetadata	Individual file properties including path, hash, size, and modification date	N:1 Snapshot, 1:N LineChanges
LineChange	Individual line-level modification with line number, hashes, and change type	N:1 Commit, N:1 FileMetadata
Branch	Logical development line with name, head commit reference, and activity status	N:1 Project, 1:1 HeadCommit
Remote	External repository reference with URL, protocol, and fetch/push refspecs	N:1 Project, 1:N Credentials
Credential	Authentication material with type, encrypted data, expiration, and usage tracking	N:1 Remote
AIActivity	Tracks AI-assisted code generation with tool name, assistance level, and confidence	N:1 Commit
Identity	Author/committer identity with name, email, and configuration date	Referenced by Commits

Table 1: Core Entity Descriptions and Relationships

4 Core Domain Model

4.1 Class Diagrams

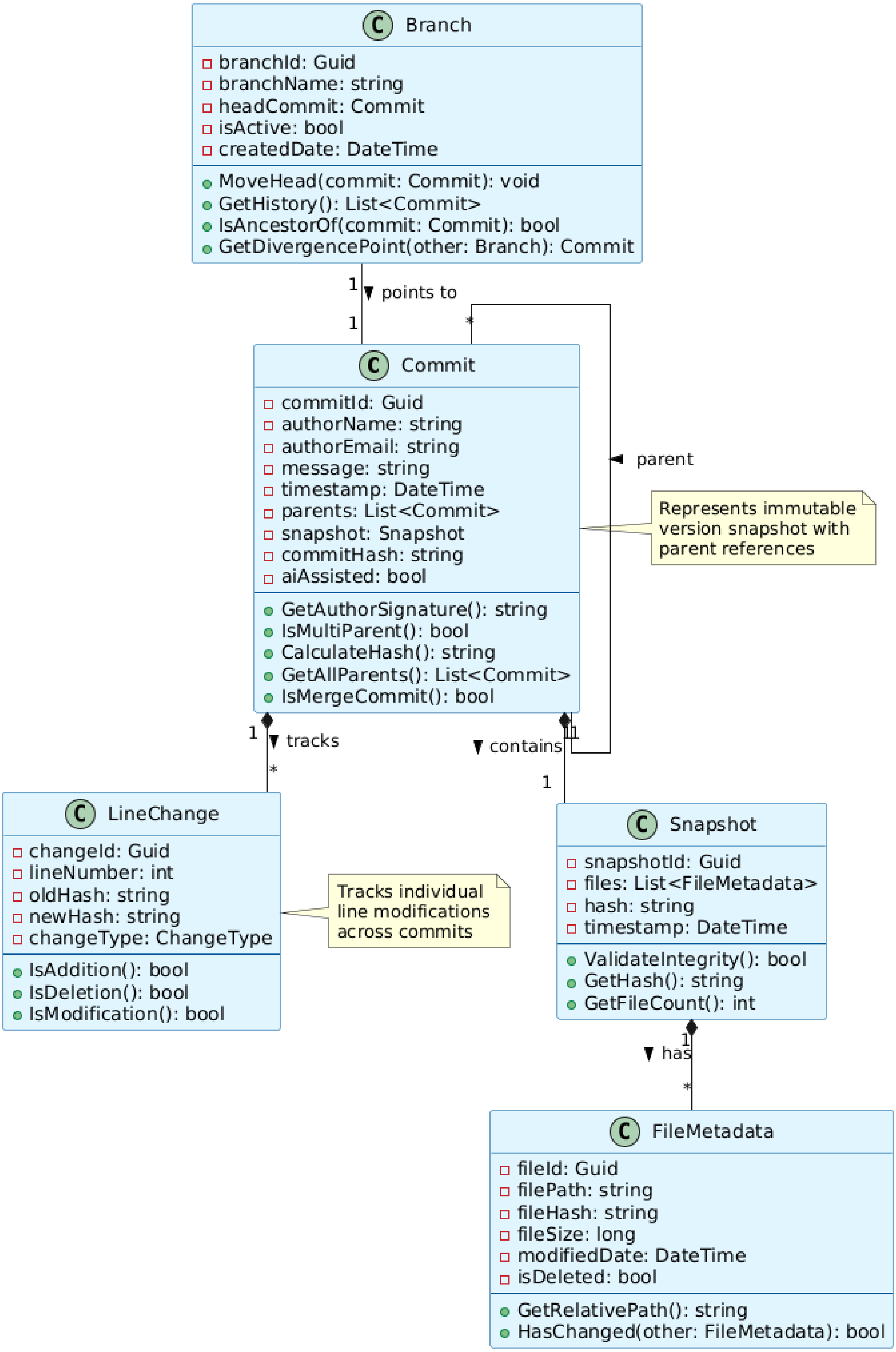

Figure 3: Core Domain Model - Class Structure for Commit, Snapshot, FileMetadata, LineChange, and Branch

4.2 Class Specifications

4.2.1 Commit Class

The `Commit` class represents an immutable version snapshot:

- **commitId:** Globally unique identifier (GUID)
- **authorName, authorEmail:** Author identification
- **message:** Commit description
- **timestamp:** Creation time (DateTime)
- **parents:** List of parent commit references
- **snapshot:** Associated filesystem snapshot
- **commitHash:** SHA-1 or SHA-256 cryptographic hash
- **aiAssisted:** Boolean flag for AI involvement

Key Methods:

- `GetAuthorSignature(): string` - Returns formatted author information
- `IsMultiParent(): bool` - Detects merge commits
- `CalculateHash(): string` - Computes commit hash
- `GetAllParents(): List<Commit>` - Recursive parent traversal
- `IsMergeCommit(): bool` - Returns true if multiple parents exist

4.2.2 Snapshot Class

Captures complete filesystem state at commit time:

- **snapshotId:** Unique identifier
- **files:** List of FileMetadata objects
- **hash:** Content-based hash
- **timestamp:** Snapshot creation time

Methods:

- `ValidateIntegrity(): bool` - Verifies snapshot consistency
- `GetHash(): string` - Returns snapshot hash
- `GetFileCount(): int` - Returns number of files

4.2.3 FileMetadata Class

Tracks individual file properties:

- **fileId:** File identifier
- **filePath:** Relative or absolute path
- **fileHash:** Content hash (SHA-256)
- **fileSize:** Size in bytes
- **modifiedDate:** Last modification timestamp
- **isDeleted:** Deletion flag

4.2.4 LineChange Class

Represents individual line modifications:

- **changeId:** Change identifier
- **lineNumber:** Line position in file
- **oldHash:** Original line content hash
- **newHash:** Modified line content hash
- **changeType:** Enumeration (ADDED, MODIFIED, DELETED)

4.2.5 Branch Class

Development line representation:

- **branchId:** Branch identifier
- **branchName:** Human-readable name
- **headCommit:** Current HEAD reference
- **isActive:** Activity status
- **createdDate:** Creation timestamp

Methods:

- `MoveHead(commit: Commit): void` - Updates HEAD pointer
- `GetHistory(): List<Commit>` - Returns commit history
- `IsAncestorOf(commit: Commit): bool` - Ancestor check
- `GetDivergencePoint(other: Branch): Commit` - Finds common ancestor

5 Services Layer Architecture

5.1 Services and Strategies

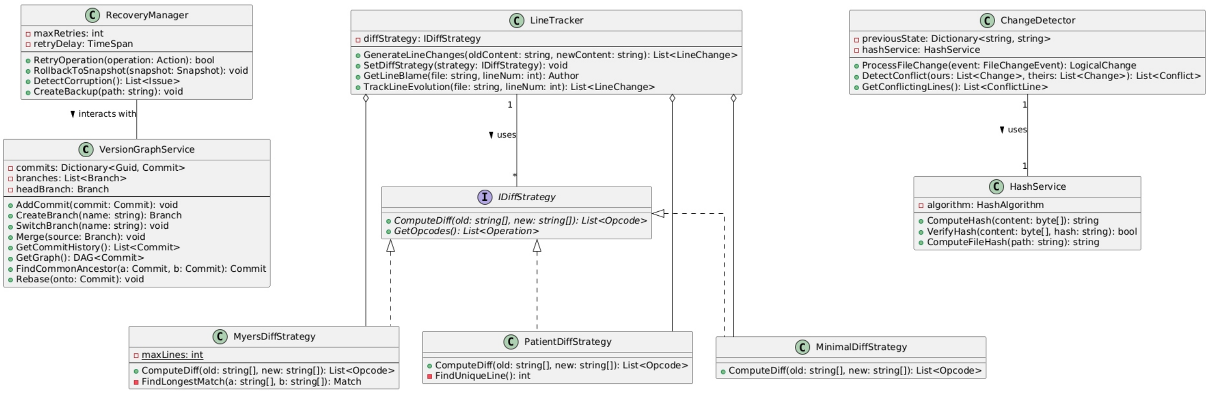
RecoveryManager
maxRetries: int
retryDelay: TimeSpan
RetryOperation(operation: Action): bool
RollbackToSnapshot(snapshot: Snapshot): void
DetectCorruption(): List<Issue>
CreateBackup(path: string): void
interacts with
VersionGraphService
commits: Dictionary<Guid, Commit>
branches: List<Branch>
headBranch: Branch
AddCommit(commit: Commit): void
CreateBranch(name: string): Branch
SwitchBranch(name: string): void
Merge(source: Branch): void
GetCommitHistory(): List<Commit>
GetGraph(): DAG<Commit>
FindCommonAncestor(a: Commit, b: Commit): Commit
Rebase(onto: Commit): void
LineTracker
diffStrategy: IDiffStrategy
GenerateLineChanges(oldContent: string, newContent: string): List<LineChange>
SetDiffStrategy(strategy: IDiffStrategy): void
GetLineBlame(file: string, lineNum: int): Author
TrackLineEvolution(file: string, lineNum: int): List<LineChange>
1
uses
*
IDiffStrategy
ComputeDiff(old: string[], new: string[]): List<Opcode>
GetOpcodes(): List<Operation>
ChangeDetector
previousState: Dictionary<string, string>
hashService: HashService
ProcessFileChange(event: FileChangeEvent): LogicalChange
DetectConflict(ours: List<Change>, theirs: List<Change>): List<Conflict>
GetConflictingLines(): List<ConflictLine>
1
uses
1
HashService
algorithm: HashAlgorithm
ComputeHash(content: byte[]): string
VerifyHash(content: byte[], hash: string): bool
ComputeFileHash(path: string): string
MyersDiffStrategy
maxLines: int
ComputeDiff(old: string[], new: string[]): List<Opcode>
FindLongestMatch(a: string[], b: string[]): Match
PatientDiffStrategy
ComputeDiff(old: string[], new: string[]): List<Opcode>
FindUniqueLine(): int
MinimalDiffStrategy
ComputeDiff(old: string[], new: string[]): List<Opcode>

Figure 4: Services Layer - VersionGraphService, Diff Strategies (Myers, Patient, Minimal), LineTracker, ChangeDetector, HashService, and RecoveryManager

5.2 VersionGraphService

Core version management service that maintains the commit DAG:

Responsibilities:

- Maintains commit graph and branch references
- Performs merge and rebase operations
- Computes common ancestors for conflict detection
- Provides commit history traversal
- Manages DAG integrity

Key Methods:

- `AddCommit(commit: Commit): void`
- `CreateBranch(name: string): Branch`
- `SwitchBranch(name: string): void`
- `Merge(source: Branch): void`
- `GetCommitHistory(): List<Commit>`
- `GetGraph(): DAG<Commit>`
- `FindCommonAncestor(a: Commit, b: Commit): Commit`
- `Rebase(onto: Commit): void`

5.3 Diff Strategy Pattern

Li'nage implements three pluggable diff algorithms using the Strategy pattern:

Strategy	**Algorithm**	**Best For**	**Complexity**
Myers	Myers' O(ND)	Most cases, balanced performance	$O(N + D^2)$
Patient	Patient diff	Long files, minimal noise, unique lines	$O(N^2)$ worst
Minimal	Greedy minimal	Speed, approximate diffs	$O(N \log N)$

Table 2: Diff Algorithm Comparison

5.3.1 IDiffStrategy Interface

```
interface IDiffStrategy {
    List<Opcode> ComputeDiff(string[] old, string[] new);
    List<Operation> GetOpcodes();
}
```

5.4 LineTracker Service

Generates and tracks line-level changes:

- Processes file changes using configured diff strategy
- Generates LineChange objects with metadata
- Tracks line evolution across commits
- Provides blame information per line
- Supports runtime strategy switching

5.5 ChangeDetector Service

Intelligent change detection and conflict identification:

- Monitors file system changes via FileWatcher
- Detects conflicts between concurrent modifications
- Maps logical changes to physical file modifications
- Provides conflict resolution suggestions
- Generates conflict reports with context

5.6 HashService

Cryptographic hashing for content integrity:

- Supports SHA-1 and SHA-256 algorithms
- Computes file-level and content-level hashes
- Verifies snapshot and commit integrity
- Pluggable hash algorithm implementation

5.7 RecoveryManager

Implements recovery and rollback capabilities:

- Retry logic with exponential backoff
- Rollback to previous snapshots
- Corruption detection and repair
- Backup creation before destructive operations
- Maximum retry count configuration

6 Authentication and Credentials

6.1 Authentication Architecture

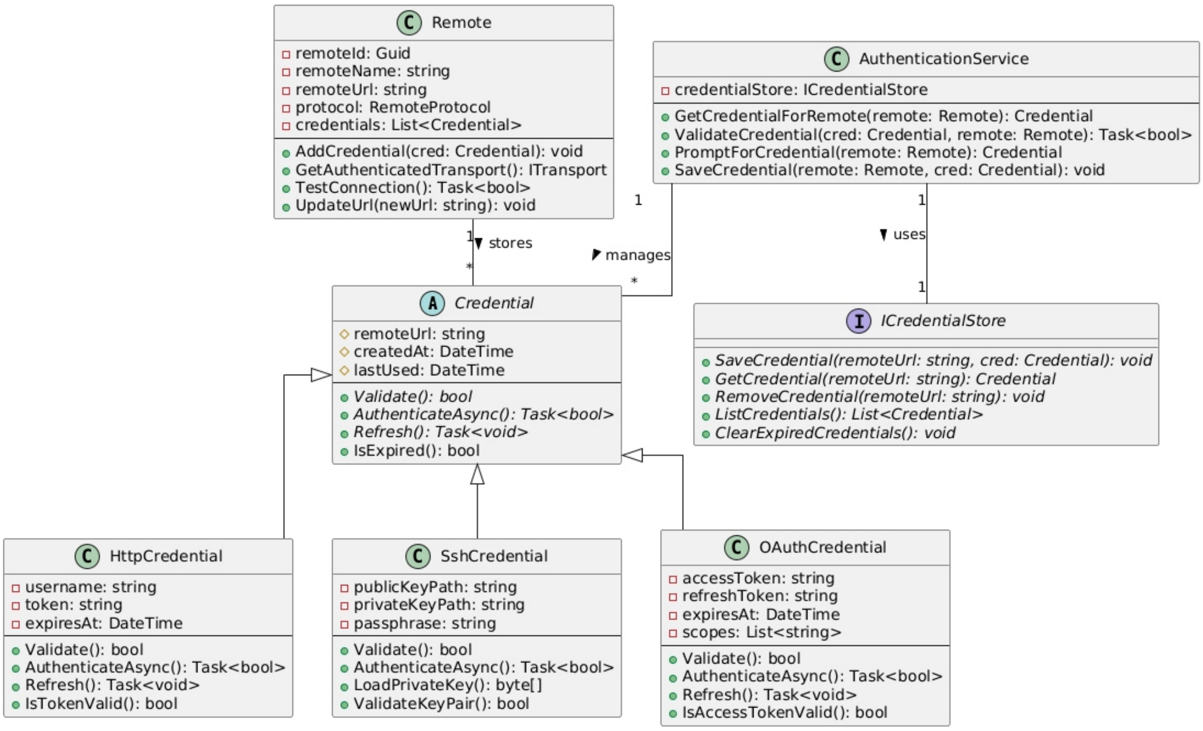
C Remote
remoteId: Guid
remoteName: string
remoteUrl: string
protocol: RemoteProtocol
credentials: List<Credential>
AddCredential(cred: Credential): void
GetAuthenticatedTransport(): ITransport
TestConnection(): Task<bool>
UpdateUrl(newUrl: string): void
C AuthenticationService
credentialStore: ICredentialStore
GetCredentialForRemote(remote: Remote): Credential
ValidateCredential(cred: Credential, remote: Remote): Task<bool>
PromptForCredential(remote: Remote): Credential
SaveCredential(remote: Remote, cred: Credential): void
1
▾ stores
*
1
▸ manages
*
1
▾ uses
1
A Credential
remoteUrl: string
createdAt: DateTime
lastUsed: DateTime
Validate(): bool
AuthenticateAsync(): Task<bool>
Refresh(): Task<void>
IsExpired(): bool
I ICredentialStore
SaveCredential(remoteUrl: string, cred: Credential): void
GetCredential(remoteUrl: string): Credential
RemoveCredential(remoteUrl: string): void
ListCredentials(): List<Credential>
ClearExpiredCredentials(): void
C HttpCredential
username: string
token: string
expiresAt: DateTime
Validate(): bool
AuthenticateAsync(): Task<bool>
Refresh(): Task<void>
IsTokenValid(): bool
C SshCredential
publicKeyPath: string
privateKeyPath: string
passphrase: string
Validate(): bool
AuthenticateAsync(): Task<bool>
LoadPrivateKey(): byte[]
ValidateKeyPair(): bool
C OAuthCredential
accessToken: string
refreshToken: string
expiresAt: DateTime
scopes: List<string>
Validate(): bool
AuthenticateAsync(): Task<bool>
Refresh(): Task<void>
IsAccessTokenValid(): bool

Figure 5: Authentication System - Credential Hierarchy with HTTP, SSH, and OAuth Implementations

6.2 Credential Hierarchy

Li’nage implements a polymorphic credential system supporting multiple authentication mechanisms:

6.2.1 Credential (Abstract Base Class)

Properties:

- `remoteUrl: string` - Associated remote repository URL
- `createdAt: DateTime` - Creation timestamp
- `lastUsed: DateTime` - Last usage timestamp

Methods:

- `Validate(): bool` - Validates credential format
- `AuthenticateAsync(): Task<bool>` - Performs authentication
- `Refresh(): Task<void>` - Refreshes expired credentials
- `IsExpired(): bool` - Checks expiration status

6.2.2 HttpCredential

Implements HTTP Basic or Token authentication:

- `username: string` - Username for HTTP Basic
- `token: string` - Personal access token
- `expiresAt: DateTime` - Token expiration

Additional Methods:

- `IsTokenValid(): bool` - Validates token before expiration

6.2.3 SshCredential

Implements SSH key-based authentication:

- `publicKeyPath: string` - Path to public key file
- `privateKeyPath: string` - Path to private key file
- `passphrase: string` - Optional key passphrase

Methods:

- `LoadPrivateKey(): byte[]` - Loads and decrypts private key
- `ValidateKeyPair(): bool` - Verifies public/private key match

6.2.4 OAuthCredential

Implements OAuth 2.0 flow:

- `accessToken: string` - Current access token
- `refreshToken: string` - Token for refreshing access
- `expiresAt: DateTime` - Access token expiration
- `scopes: List<string>` - Granted OAuth scopes

Methods:

- `IsAccessTokenValid(): bool` - Checks token validity
- `Refresh(): Task<void>` - Uses refresh token to obtain new access token

6.3 AuthenticationService

Manages credential lifecycle:

- `GetCredentialForRemote(remote: Remote): Credential`
- `ValidateCredential(cred: Credential, remote: Remote): Task<bool>`
- `PromptForCredential(remote: Remote): Credential`
- `SaveCredential(remote: Remote, cred: Credential): void`

6.4 ICredentialStore Interface

Abstraction for credential persistence:

```
interface ICredentialStore {
    void SaveCredential(string remoteUrl, Credential cred);
    Credential GetCredential(string remoteUrl);
    void RemoveCredential(string remoteUrl);
    List<Credential> ListCredentials();
    void ClearExpiredCredentials();
}
```

Implementation integrates with Windows Credential Manager for secure storage with encryption.

7 Workflow and Processes

7.1 Commit Creation Workflow

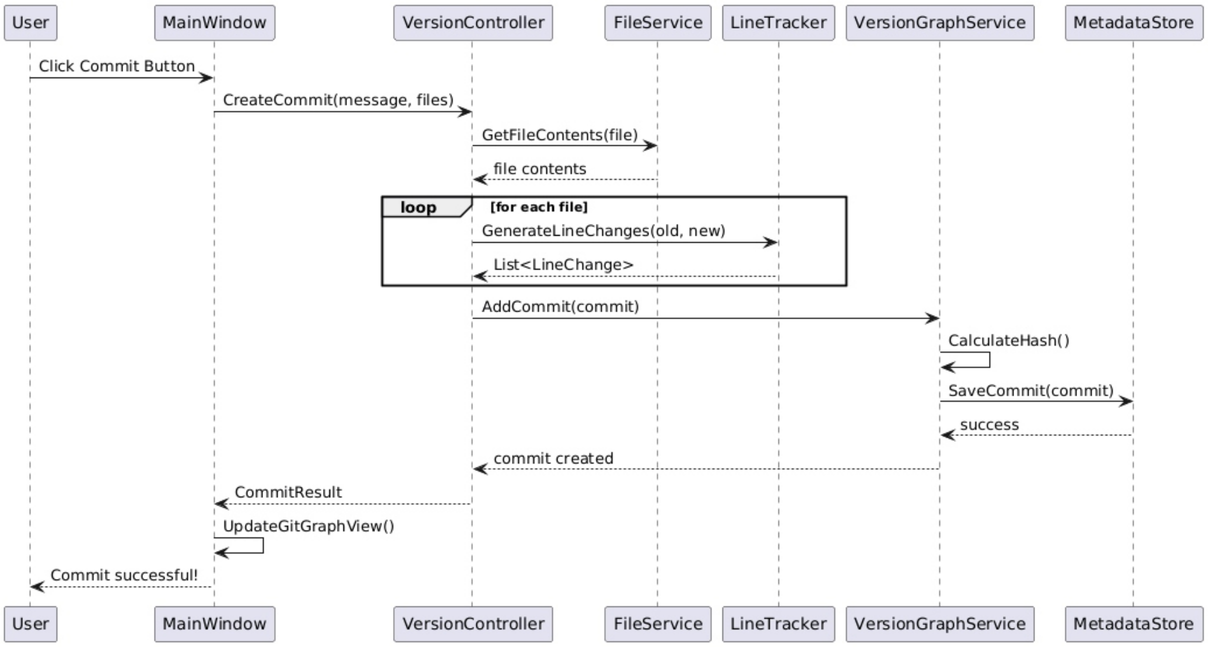

Figure 6: Commit Sequence Diagram - Complete Workflow from User Action to Successful Commit

7.2 Commit Process Flow

The commit creation process involves multiple coordinated steps across all architectural layers:

1. **User Initiates Commit**
   - User clicks "Commit" button in MainWindow
   - UI collects commit message and file selection
2. **Controller Orchestration**
   - MainWindow calls `VersionController.CreateCommit(message, files)`
   - VersionController validates input parameters
3. **File Content Retrieval**
   - VersionController calls `FileService.GetFileContents(file)`
   - FileService reads current file state from disk
   - Returns file contents to controller
4. **Line-Level Analysis Loop**

- For each modified file:
    (a) Call `LineTracker.GenerateLineChanges(oldContent, newContent)`
    (b) LineTracker uses configured diff strategy (Myers/Patient/Minimal)
    (c) Generates list of opcodes (add, delete, modify operations)
    (d) Creates LineChange objects with metadata (line number, hashes, type)
    (e) Returns `List<LineChange>` to controller

5. **Commit Creation**
    - VersionController calls `VersionGraphService.AddCommit(commit)`
    - VersionGraphService performs:
        (a) Commit hash calculation using HashService
        (b) Snapshot creation with all file metadata
        (c) Association with current branch HEAD
        (d) Parent commit linking

6. **Persistence**
    - VersionGraphService calls `MetadataStore.SaveCommit(commit)`
    - MetadataStore operations:
        (a) Serializes commit data to storage format
        (b) Writes to metadata repository
        (c) Creates indexes for fast retrieval
        (d) Returns success confirmation

7. **UI Update**
    - VersionController returns `CommitResult` to MainWindow
    - MainWindow calls `UpdateGitGraphView()`
    - GitGraphView refreshes:
        (a) Version graph visualization
        (b) Branch pointer positions
        (c) Commit timeline

8. **User Notification**
    - MainWindow displays "Commit successful!" message
    - Shows commit hash and timestamp

7.3 Transaction Semantics

All commit operations follow ACID properties:

- **Atomicity:** Entire commit succeeds or fails as a unit
- **Consistency:** Commit preserves DAG integrity
- **Isolation:** Concurrent commits are serialized
- **Durability:** Committed data persists to storage

8 Component Architecture

8.1 Component Interactions

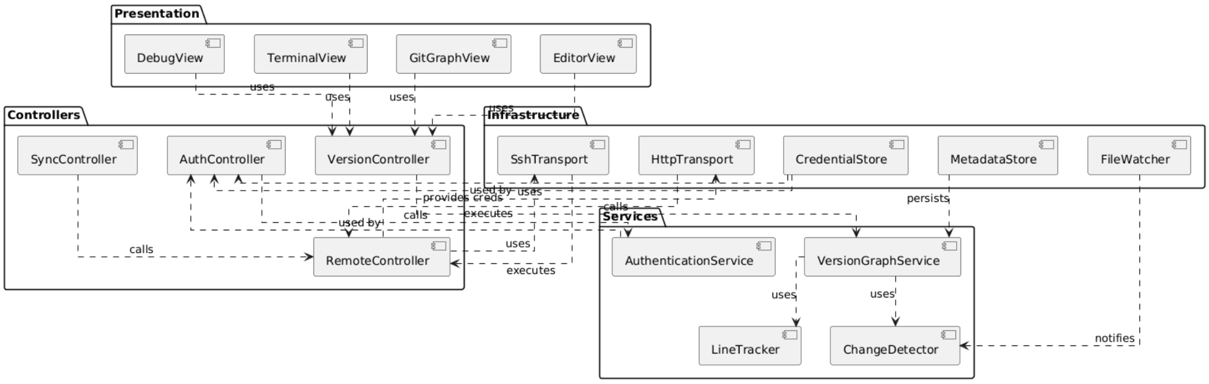

Figure 7: Component Diagram - All Components and Their Interactions with Dependencies

8.2 Component Dependency Matrix

Component	Depends On	Provides To
EditorView	VersionController, FileService	User Interface
GitGraphView	VersionController, VersionGraphService	User Interface
TerminalView	All Controllers	User Interface
VersionController	VersionGraphService, FileService, LineTracker	Presentation Layer
AuthController	AuthenticationService, ICredentialStore	Presentation, RemoteController
RemoteController	HttpTransport, SshTransport, AuthenticationService	SyncController
VersionGraphService	LineTracker, ChangeDetector, MetadataStore	Controllers
LineTracker	IDiffStrategy (Myers/Patient/Minimal)	VersionGraphService, ChangeDetector
ChangeDetector	HashService, FileWatcher	VersionGraphService
AuthenticationService	ICredentialStore	AuthController, RemoteController
FileWatcher	Operating System APIs	ChangeDetector, Infrastructure
CredentialStore	Windows Credential Manager	AuthenticationService

Table 3: Component Dependencies and Information Flow

8.3 Design Patterns Used

- **Strategy Pattern:** IDiffStrategy with multiple implementations
- **Repository Pattern:** MetadataStore, CredentialStore
- **Observer Pattern:** FileWatcher notifying ChangeDetector
- **Factory Pattern:** Credential creation based on type
- **Command Pattern:** All version control operations
- **Template Method:** Credential.AuthenticateAsync()

9 Deployment Architecture

9.1 System Deployment

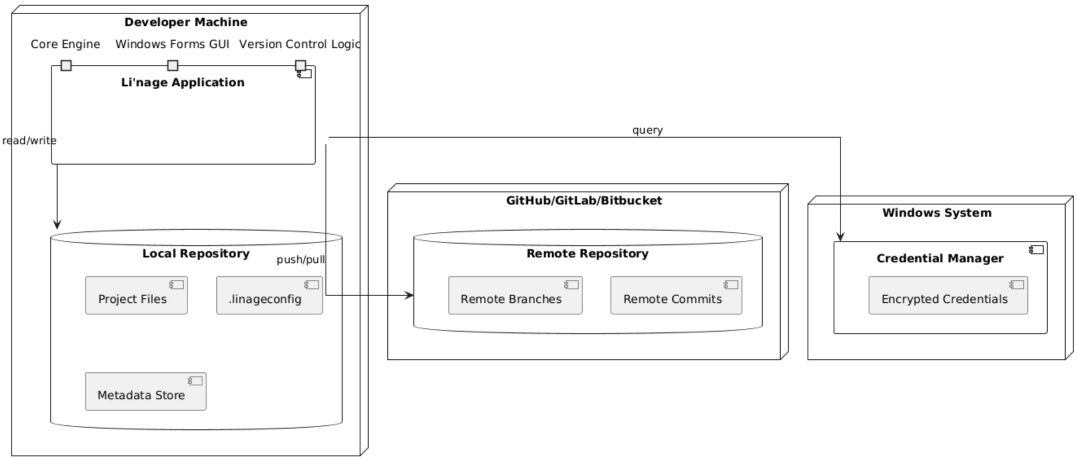

Figure 8: Deployment Diagram - Developer Machine, Local Repository, Remote Repository, and Windows System Integration

9.2 Deployment Topology

9.2.1 Developer Machine (Primary Environment)

The Li'nage application runs entirely on the developer's local machine:

Li'nage Application Components:

- **Core Engine:** Business logic execution
- **Windows Forms GUI:** User interface rendering
- **Version Control Logic:** All VCS operations

Local Repository Structure:

- **Project Files:** Source code and assets under version control
- **.linageconfig:** Configuration file with repository settings
- **Metadata Store:** Database containing:
  - Commit history and DAG
  - Snapshots and file metadata
  - Line change records
  - Branch references
  - Remote configurations

9.2.2 Remote Repository (GitHub/GitLab/Bitbucket)

External hosting for distributed collaboration:

- **Remote Branches:** Branch references synchronized via push/pull
- **Remote Commits:** Full commit history with snapshots
- **Remote Tags:** Release markers and version tags

9.2.3 Windows System Integration

Operating system services utilized:

- **Credential Manager:** Secure encrypted credential storage
- **File System:** NTFS monitoring and file operations
- **Network Stack:** HTTP/HTTPS and SSH communication

9.3 Data Flow Patterns

- **read/write:** Li'nage Application ↔ Local Repository
- **query:** Li'nage Application → Windows Credential Manager
- **push/pull:** Li'nage Application ↔ Remote Repository (via Network)
- **monitor:** File System → FileWatcher → ChangeDetector

10 Performance and Security

10.1 Performance Characteristics

10.1.1 Diff Algorithm Performance

Algorithm	Time Complexity	Space Complexity	Best Case
Myers	$O(N + D^2)$	$O(N + D)$	$O(N)$ for identical
Patient	$O(N^2)$ worst	$O(N)$	$O(N)$ for unique lines
Minimal	$O(N \log N)$	$O(N)$	$O(N)$ approximate

Table 4: Diff Algorithm Performance Characteristics

Where N = number of lines and D = edit distance (number of changes).

10.1.2 Caching Strategy

- **Commit Graph Cache:** In-memory DAG for fast branch operations
- **File Hash Cache:** Computed hashes stored per snapshot
- **Credential Cache:** Active credentials cached with expiration
- **Line Change Cache:** Recent diffs cached per file

10.1.3 Performance Targets

Operation	Target Time
Commit creation (typical files)	$<$ 500 ms
Line tracking per file	$<$ 10 ms
Diff computation (1000 lines)	$<$ 100 ms
Branch switch	$<$ 200 ms
Remote push	Network-dependent

Table 5: Performance Targets

10.2 Security Architecture

10.2.1 Credential Storage Security

- Windows Credential Manager integration with OS-level encryption
- Private SSH keys protected with user-specific encryption
- Encrypted at-rest storage using AES-256
- Memory clearing after authentication (secure disposal)
- No plaintext credential storage in configuration files

10.2.2 Authentication Flow

1. Credential retrieval from secure storage
2. Automatic token refresh before expiration
3. Validation before each remote operation
4. Audit logging of all authentication events
5. Failed authentication rate limiting

10.2.3 Data Protection

- **File Integrity:** SHA-256 hash verification
- **Commit Immutability:** Cryptographic commit hashes prevent tampering
- **Snapshot Consistency:** Integrity checks on snapshot creation and retrieval
- **Rollback Safety:** Backup snapshots before destructive operations

11 Implementation Roadmap

11.1 20-Week Development Plan

11.1.1 Phase 1: Foundation (Weeks 1-5)

Week 1-2: Data Model

- Implement core domain classes (Commit, Snapshot, FileMetadata, LineChange, Branch)
- Define database schema and migrations
- Create MetadataStore with SQLite backend
- Unit tests for all entities ($> 90\%$ coverage)

Week 3-4: Version Graph

- Implement DAG data structure for commit graph
- VersionGraphService core operations
- Branch management (create, switch, delete)
- Parent-child relationship management

Week 5: Snapshot System

- Snapshot creation and validation
- File metadata extraction
- Hash computation service
- Integration tests

11.1.2 Phase 2: Core Features (Weeks 6-10)

Week 6-7: Diff Algorithms

- IDiffStrategy interface definition
- Myers diff algorithm implementation
- Unit tests with various edge cases
- Performance benchmarking

Week 8: LineTracker Service

- LineTracker implementation with strategy pattern
- Line change generation

- Line evolution tracking
- Blame functionality

Week 9: ChangeDetector

- FileWatcher integration
- Change detection logic
- Conflict identification
- Merge scenario handling

Week 10: Integration Testing

- End-to-end commit workflow tests
- Branch and merge testing
- Performance profiling
- Bug fixes and optimization

11.1.3 Phase 3: Authentication (Weeks 11-13)

Week 11: Credential System

- Credential class hierarchy
- HttpCredential implementation
- Windows Credential Manager integration
- Encryption/decryption utilities

Week 12: SSH and OAuth

- SshCredential with key management
- OAuthCredential with token refresh
- AuthenticationService implementation
- Credential validation and testing

Week 13: Remote Integration

- Remote class implementation
- HttpTransport and SshTransport
- Push/pull basic operations
- Authentication flow testing

11.1.4 Phase 4: UI Implementation (Weeks 14-16)

Week 14: Main Window

- Windows Forms project setup
- MainWindow layout and controls
- Commit interface implementation
- File selection and staging

Week 15: Visualization

- GitGraphView with DAG rendering
- EditorView with syntax highlighting
- Line tracking visualization
- Branch switching UI

Week 16: Additional Views

- TerminalView for command execution
- DebugView for diagnostics
- UI refinement and polish
- User experience testing

11.1.5 Phase 5: Advanced Features (Weeks 17-19)

Week 17: Alternative Diff Strategies

- Patient diff algorithm
- Minimal diff algorithm
- Strategy selection UI
- Comparative performance testing

Week 18: Advanced Operations

- Conflict resolution UI
- AI activity tracking
- RecoveryManager implementation
- Rollback functionality

Week 19: Performance Optimization

- Caching implementation
- Index optimization
- Memory profiling and optimization
- Large repository testing

11.1.6 Phase 6: Polish and Deployment (Week 20)

- Documentation completion (API docs, user guide)
- Security audit and penetration testing
- Final performance tuning
- Installer creation
- Beta user acceptance testing
- Production release preparation

11.2 Milestones and Deliverables

Milestone	**Week**	**Deliverable**
M1: Data Model Complete	5	Core domain classes with tests
M2: Core VCS Operations	10	Commit, branch, merge working
M3: Authentication Ready	13	Multi-credential support
M4: UI Complete	16	Full graphical interface
M5: Feature Complete	19	All advanced features
M6: Production Ready	20	Tested, documented, deployed

Table 6: Project Milestones

12 Risk Analysis and Mitigation

12.1 Technical Risks

Risk	Impact	Mitigation Strategy
Performance with large files	Slow diff computation, poor user experience	Implement streaming diff algorithms, file size limits, chunked processing
Merge conflict complexity	Difficult conflict resolution, data loss risk	Develop 3-way merge with visualization, automatic conflict detection
Data corruption	Lost version history, repository damage	Implement checksums, automatic backups, corruption detection
Credential security leaks	Unauthorized access, security breach	Encrypt all credentials, use OS credential manager, audit logging
Network failures during sync	Interrupted operations, inconsistent state	Retry logic with exponential backoff, transaction rollback
Memory exhaustion	Application crashes, data loss	Streaming operations, memory profiling, garbage collection optimization
DAG corruption	Invalid commit graph, broken history	Integrity validation, parent verification, repair utilities
Concurrent access issues	Race conditions, data inconsistency	File locking, transaction serialization, atomic operations

Table 7: Technical Risk Analysis and Mitigation

12.2 Project Risks

- **Schedule Risk:** 20-week timeline is aggressive
  - Mitigation: Bi-weekly milestone reviews, scope adjustment if needed
- **Resource Risk:** Team availability and expertise
  - Mitigation: Cross-training, documentation, pair programming
- **Integration Risk:** Windows system dependencies
  - Mitigation: Early integration testing, abstraction layers
- **User Adoption Risk:** Learning curve for new system
  - Mitigation: Comprehensive documentation, tutorials, migration guides

13 Quality Attributes

13.1 Reliability

- **Target Uptime:** 99.9% for local operations
- **Recovery Time:** < 1 minute for failures
- **Data Loss Prevention:** Full backup on every commit
- **Error Handling:** Graceful degradation with user notification
- **Rollback Capability:** Complete state recovery to any previous commit

13.2 Performance

- **Commit Creation:** < 500ms for typical files (up to 10,000 lines)
- **Line Tracking:** < 10ms per file
- **Diff Computation:** < 100ms for 1,000-line files
- **Branch Operations:** < 200ms for switch/create/delete
- **Graph Rendering:** < 2 seconds for 10,000 commits

13.3 Scalability

- **Commit Capacity:** Support 100,000+ commits per repository
- **File Support:** Unlimited file count with performance degradation considerations
- **Line Change Records:** Millions of line changes with indexed retrieval
- **Branch Count:** Support hundreds of concurrent branches
- **Repository Size:** Handle multi-gigabyte repositories

13.4 Maintainability

- **Code Coverage:** > 85% unit test coverage
- **Documentation:** 100% of public APIs documented
- **Architecture:** Clean layers with well-defined interfaces
- **Code Quality:** Static analysis passing with no critical issues
- **Technical Debt:** Tracked and addressed regularly

13.5 Security

- **Credential Protection:** OS-level encryption for all credentials
- **Data Integrity:** Cryptographic hashing for all content
- **Access Control:** File system permissions respected
- **Audit Trail:** Complete logging of authentication and operations
- **Vulnerability Management:** Regular security audits

13.6 Usability

- **Learning Curve:** Familiar Git-like interface
- **Visual Feedback:** Real-time visualization of operations
- **Error Messages:** Clear, actionable error descriptions
- **Help System:** Context-sensitive help and tooltips
- **Accessibility:** Keyboard shortcuts and screen reader support

14 Technology Stack

14.1 Development Platform

- **Language:** C# 11.0 or later
- **Framework:** .NET 7.0 or .NET 8.0
- **IDE:** Visual Studio 2022 or JetBrains Rider
- **UI Framework:** Windows Forms (.NET)
- **Target OS:** Windows 10/11 (x64)

14.2 Data Storage

- **Metadata Store:** SQLite 3.x for local database
- **ORM:** Entity Framework Core 7.0+
- **Serialization:** JSON (System.Text.Json) for configuration
- **File System:** NTFS with change notification APIs

14.3 Security and Cryptography

- **Hashing:** SHA-256 (System.Security.Cryptography)
- **Encryption:** AES-256 for credential storage
- **SSH:** SSH.NET library for SSH operations
- **HTTPS:** System.Net.Http.HttpClient
- **Credential Manager:** Windows Credential Manager API

14.4 Testing

- **Unit Testing:** xUnit or NUnit
- **Mocking:** Moq framework
- **Code Coverage:** Coverlet + ReportGenerator
- **Integration Testing:** xUnit with TestContainers
- **Performance Testing:** BenchmarkDotNet

14.5 Build and Deployment

- **Build System:** MSBuild with SDK-style projects
- **CI/CD:** GitHub Actions or Azure DevOps
- **Package Management:** NuGet
- **Installer:** WiX Toolset or Advanced Installer
- **Version Control:** Git (for Li'nage development itself)

14.6 Third-Party Libraries

Library	**Purpose**	**License**
SSH.NET	SSH protocol implementation	MIT
LibGit2Sharp	Git protocol reference	MIT
Newtonsoft.Json	JSON serialization (alternative)	MIT
Serilog	Structured logging	Apache 2.0
Autofac	Dependency injection	MIT

Table 8: Third-Party Library Dependencies

15 Appendices

15.1 Glossary

Commit
Immutable snapshot of code changes with metadata (author, timestamp, message, parents)

Snapshot
Complete filesystem state at a specific point in time, containing all file metadata

LineChange
Individual line-level modification record (addition, deletion, or modification)

Diff Computation of differences between two versions of a file or set of files

Branch
Named reference to a sequence of commits, representing a development line

Merge
Combining changes from multiple branches into a single unified history

Rebase
Re-applying commits from one branch on top of another base commit

Remote
External repository reference with URL and authentication credentials

Credential
Authentication material for accessing remote repositories (token, key, or OAuth)

DAG
Directed Acyclic Graph representing commit relationships without cycles

Conflict
Incompatible changes in the same location requiring manual resolution

Rollback
Reverting repository state to a previous commit or snapshot

Hash
Cryptographic fingerprint (SHA-1 or SHA-256) uniquely identifying content

HEAD
Reference to the current commit in the active branch

Ancestor
A commit that precedes another commit in the version history

Opcode
Operation code describing a specific diff operation (add/delete/modify)

Strategy

Pluggable algorithm implementation (e.g., diff strategies)

15.2 References

1. Myers, E. W. (1986). *An O(ND) difference algorithm and its variations.* Algorithmica, 1(1), 251-266. DOI: 10.1007/BF01840446

2. Bram, C. (1990). *Patient diff algorithm.* GNU Diffutils documentation. Available at: https://www.gnu.org/software/diffutils/

3. Hunt, J. W., & Szymanski, T. G. (1977). *A fast algorithm for computing longest common subsequences.* Communications of the ACM, 20(5), 350-353. DOI: 10.1145/359581.359603

4. Microsoft. (2024). *Windows Credential Manager API Reference.* Retrieved from https://docs.microsoft.com/en-us/windows/win32/api/wincred/

5. Git Project. (2024). *Git Internals - Git Objects.* Pro Git Book, Chapter 10. Retrieved from https://git-scm.com/book/en/v2/Git-Internals

6. Hardt, D. (Ed.). (2012). *The OAuth 2.0 Authorization Framework.* RFC 6749. Retrieved from https://tools.ietf.org/html/rfc6749

7. Torvalds, L. & Hamano, J. (2024). *Git Source Code Repository.* GitHub. Retrieved from https://github.com/git/git

8. Martin, R. C. (2017). *Clean Architecture: A Craftsman's Guide to Software Structure and Design.* Prentice Hall.

9. Gamma, E., Helm, R., Johnson, R., & Vlissides, J. (1994). *Design Patterns: Elements of Reusable Object-Oriented Software.* Addison-Wesley.

10. Fowler, M. (2018). *Refactoring: Improving the Design of Existing Code (2nd ed.).* Addison-Wesley Professional.

15.3 Acronyms and Abbreviations

Acronym	**Meaning**
API	Application Programming Interface
DAG	Directed Acyclic Graph
HTTP	Hypertext Transfer Protocol
HTTPS	HTTP Secure
SSH	Secure Shell
OAuth	Open Authorization
GUID	Globally Unique Identifier
SHA	Secure Hash Algorithm
AES	Advanced Encryption Standard
VCS	Version Control System
UI	User Interface
IDE	Integrated Development Environment
ORM	Object-Relational Mapping
JSON	JavaScript Object Notation
NTFS	New Technology File System
CI/CD	Continuous Integration/Continuous Deployment

Table 9: Acronyms and Abbreviations

16 Conclusion

Li'nage represents a comprehensive solution for modern version control with unprecedented line-level tracking capabilities. This technical proposal has detailed the complete system architecture, from low-level data models to high-level user interfaces, demonstrating a well-designed, maintainable, and scalable system.

16.1 Key Achievements

- **Clean Architecture:** Four-layer design with clear separation of concerns
- **Flexible Diff Algorithms:** Pluggable strategy pattern supporting multiple algorithms
- **Comprehensive Authentication:** Support for HTTP, SSH, and OAuth credentials
- **Advanced Visualization:** Interactive commit graph and line tracking
- **Enterprise Security:** Windows Credential Manager integration with encryption
- **Performance Optimized:** Caching, indexing, and algorithmic efficiency

16.2 Next Steps

1. **Approval and Funding:** Secure stakeholder approval and resource allocation
2. **Team Formation:** Assemble development team with required expertise
3. **Environment Setup:** Configure development, testing, and CI/CD infrastructure
4. **Phase 1 Initiation:** Begin implementation with data model foundation
5. **Iterative Development:** Follow 20-week roadmap with bi-weekly reviews
6. **Beta Testing:** Conduct user acceptance testing in weeks 19-20
7. **Production Deployment:** Release version 1.0 with full documentation

16.3 Success Criteria

The project will be considered successful when:

- All functional requirements are implemented and tested
- Performance targets are met ($<$ 500ms commit time)
- Security audit passes with no critical vulnerabilities
- User documentation is complete and accessible

- Beta testing demonstrates positive user satisfaction
- Production deployment is stable and reliable