Architecture of MiniGit

1. Introduction to the Architecture

MiniGit is a simplified, educational version control system inspired by Git. Its architecture is deliberately designed to balance **conceptual clarity**, **functional completeness**, and **ease of understanding** for students and beginners. Unlike production-grade Git, which is highly optimized and distributed, MiniGit focuses on explaining *how a version control system fundamentally works* under the hood.

At a high level, MiniGit follows a **layered, modular architecture** that separates user interaction, core logic, and persistent storage. This separation ensures that each responsibility is clearly defined, making the system easier to reason about, debug, extend, and explain.

The architecture is built around four core pillars:

1. Command-Line Interface (CLI)
2. Core Version Control Engine
3. Persistent Storage Layer
4. Metadata and State Management

Each pillar interacts with the others in a controlled and predictable way, forming a complete end-to-end system.

---

2. High-Level Architectural Overview

Conceptually, MiniGit can be visualized as a pipeline:

User → CLI Commands → Core Logic → File System Storage → Output to User

When a user issues a command such as `add`, `commit`, or `log`, the request flows through the system in the following stages:

1. **Input Parsing** – The command and arguments are interpreted.
2. **Command Dispatching** – The appropriate function is selected.
3. **Core Processing** – Internal logic manipulates data structures and metadata.
4. **Persistent Storage** – Changes are written to disk.
5. **Feedback Output** – Results are printed back to the user.

This deterministic flow ensures transparency and predictability, which is essential for a learning-oriented system.

---

3. Command-Line Interface (CLI) Layer

3.1 Purpose

The CLI layer serves as the *only entry point* for users. It acts as a translator between human-readable commands and machine-executable logic.

3.2 Design Philosophy

The CLI is intentionally minimalistic:

- No graphical interface
- No hidden automation
- Explicit commands for every operation

This design allows users to clearly observe cause-and-effect relationships between commands and system state changes.

3.3 Implementation

The CLI is implemented within the `main()` function. It performs the following responsibilities:

- Validates command-line arguments
- Identifies the requested command
- Routes execution to the corresponding function

Example commands include:

- `init`
- `add <filename>`
- `commit -m "message"`
- `log`
- `branch <name>`
- `checkout <branch>`
- `merge <branch>`
- `diff <commit1> <commit2>`

Each command maps directly to a single function, enforcing a one-command–one-responsibility principle.

---

4. Core Version Control Engine

4.1 Role of the Core Engine

The core engine is the intellectual heart of MiniGit. It defines how data is interpreted, stored, and transformed.

This layer is responsible for:

- Hash generation
- Commit creation
- Branch tracking
- Merge logic
- Commit traversal (log)
- Difference analysis (diff)

4.2 Stateless Function Design

Most core functions are *stateless* in memory. Instead of maintaining long-lived in-memory objects, MiniGit reads state from disk when needed and writes updates immediately.

This design choice:

- Reduces complexity
- Avoids memory synchronization issues
- Reflects real-world version control behavior

---

5. Persistent Storage Architecture

5.1 File-System-Based Storage

MiniGit uses the local file system as its database. All internal data is stored inside a hidden directory named `.minigit`.

This directory acts as the repository's internal state container.

```
.minigit/
├── objects/
├── commits/
├── index.txt
├── branches.txt
└── HEAD
```

Each subcomponent has a clearly defined responsibility.

5.2 Objects Directory

The `objects/` directory stores file contents indexed by hash. When a file is added:

- Its content is read
- A hash is computed
- The content is stored as a file named after the hash

This ensures:

- Content-based addressing
- Deduplication of identical files
- Immutability of stored objects

This mirrors Git's blob storage model.

5.3 Index (Staging Area)

The `index.txt` file represents the staging area. It stores entries in the format:

filename:hash

This intermediate buffer allows users to control exactly what goes into a commit, enabling partial commits and clear intent.

5.4 Commit Storage

Each commit is stored as a file in the `commits/` directory. The commit file contains:

- Commit message
- Parent commit hash
- Branch name
- Timestamp
- Snapshot of staged files

Commits form a **linked list structure**, where each commit references its parent.

---

6. Metadata and State Management

6.1 HEAD Pointer

The `HEAD` file stores the name of the currently active branch. This abstraction allows branch switching without rewriting commit history.

6.2 Branch Mapping

The `branches.txt` file maps branch names to their latest commit hashes:

main:abc123
feature:xyz789

This design enables constant-time access to branch heads and simplifies checkout and merge operations.

---

7. Commit Graph Architecture

Although MiniGit stores commits as files, conceptually the commit history forms a **Directed Acyclic Graph (DAG)**. In the simplified implementation:

- Each commit has exactly one parent
- Merge commits logically combine histories

Traversal of this structure is performed iteratively during `log` operations.

---

8. Merge Architecture

MiniGit implements a simplified merge strategy:

- Reads file snapshots from both branches
- Detects conflicts via hash comparison
- Prioritizes the current branch in conflicts
- Generates an automatic merge commit

This approach prioritizes conceptual understanding over completeness, making merge behavior predictable and explainable.

---

9. Architectural Trade-offs

Advantages

- High transparency

- Easy debugging
- Educational clarity
- Deterministic behavior

Limitations

- No distributed support
- No advanced conflict resolution
- No performance optimizations

These trade-offs are intentional and align with MiniGit's learning-focused goals.

---

10. Conclusion

The architecture of MiniGit demonstrates how a real-world system like Git can be broken down into understandable components without losing conceptual correctness. By combining a clean CLI, a stateless core engine, and a file-system-backed storage model, MiniGit achieves a complete, end-to-end version control workflow suitable for academic evaluation and conceptual mastery.

This architectural foundation also makes MiniGit extensible, allowing future enhancements such as visualization tools, reset operations, or distributed synchronization.

Data Structures Used in MiniGit Project

Introduction

At the heart of any version control system lies its data structures. They decide **how efficiently data is stored**, **how fast history can be traversed**, and **how reliably versions can be reconstructed**. In this MiniGit project, the goal was not to merely copy Git commands, but to **understand and re-implement Git's core ideas using fundamental C++ data structures** in a way that is readable, educational, and extensible.

This section explains **what data structures were used**, **why they were chosen**, **how they interact with each other**, and **how they conceptually map to real Git's internal object model**. The explanation is written assuming the reader has basic programming knowledge but may not know how Git works internally.

High-Level View of Data Structures

The MiniGit system primarily relies on the following data structures:

- **Files and directories** (persistent storage)
- **unordered_map / map** (fast lookup and metadata storage)
- **Linked-list–like commit structure** (commit history)
- **Hashing concepts** (identity and integrity)
- **Vectors and strings** (supporting structures)

Each structure was chosen intentionally to mirror **real Git concepts** while keeping implementation complexity manageable.

---

1. File System as the Primary Persistent Data Structure

Why Files Are Used

Git is fundamentally a **filesystem-based database**. Similarly, MiniGit stores:

- Commits
- File snapshots
- Branch pointers
- Logs

as **actual files and folders** inside a hidden directory (e.g., `.minigit/`).

Advantages of Using Files

1. **Persistence** – Data survives program termination.
2. **Transparency** – Users can inspect internal state manually.
3. **Simplicity** – No external databases required.
4. **Realism** – Matches Git's actual design philosophy.

Structure Example

```
.minigit/
├── commits/
│   ├── commit_1.txt
│   ├── commit_2.txt
├── branches/
│   ├── main.txt
├── index/
│   ├── staging.txt
```

Each file acts as a **serialized data structure**, storing metadata and relationships.

Relation to Real Git

MiniGit	Real Git
Text files	Compressed object files
Manual parsing	Binary parsing
Human-readable	Optimized for speed

---

2. Maps and unordered_maps for Fast Metadata Access

Why Maps Are Needed

Version control requires frequent operations like:

- Check if a file is staged
- Find a commit by ID
- Map filenames to hashes
- Map branches to commit heads

These operations demand **fast key-based access**, which is where maps come in.

---

3. Why `unordered_map` Was Chosen Over `map`

Key Differences

Feature	map	unordered_map
Underlying structure	Red-black tree	Hash table
Lookup complexity	O(log n)	O(1) average
Ordering	Sorted	Unordered
Use case	Ordered traversal	Fast lookup

Why Ordering Is Not Required

MiniGit does **not** require sorted keys. For example:

- File names don't need alphabetical order
- Commit IDs don't need sorting
- Branch names are accessed directly

Hence, **ordering would add unnecessary overhead**.

Performance Benefit

Operations like `add`, `commit`, `status`, and `diff` require **frequent existence checks**.
Using `unordered_map` ensures:

- Constant-time access
- Faster execution for large repositories

Example Usage

```
unordered_map<string, string> stagedFiles;
stagedFiles["main.cpp"] = "hash123";
```

This structure mirrors Git's **index (staging area)**.

---

4. Commits as a Linked List Structure

Conceptual Design

Each commit stores:

- Commit ID
- Message

- Timestamp
- Parent commit ID
- Snapshot of tracked files

This naturally forms a **linked list**, where:

- Each commit points to its parent
- The HEAD pointer references the latest commit

Visualization

Commit_5 → Commit_4 → Commit_3 → Commit_2 → Commit_1

Why Linked List Fits Perfectly

1. **History traversal** – `git log`
2. **Undo capability** – checkout previous commit
3. **Branching** – multiple pointers to same commit

Implementation Insight

Instead of using pointers in memory, MiniGit stores:

parent_commit_id: commit_4

This is a **persistent linked list**, stored on disk.

Relation to Real Git

MiniGit	Real Git
Single parent	Single parent commit
Text reference	SHA-1 reference
Linear history	DAG (directed acyclic graph)

MiniGit intentionally simplifies Git's DAG into a mostly linear structure for learning clarity.

---

5. Hashing Concepts for Identity and Integrity

Why Hashing Is Important

Hashing enables:

- Unique identification of commits
- Detecting file changes
- Preventing duplicate storage

In MiniGit, hashing is often simplified but conceptually mirrors Git.

What Gets Hashed

- File contents
- Commit metadata
- Snapshot state

Purpose of Hashing

1. **Uniqueness** – Same content → same hash
2. **Integrity** – Change → new hash
3. **Immutability** – Commits never change

Relation to Real Git

MiniGit	Real Git
Simple hash/string	SHA-1 / SHA-256
Educational	Cryptographically secure

---

6. unordered_map + Files = Persistent Hash Tables

MiniGit effectively creates **persistent hash tables** by:

- Using `unordered_map` in memory
- Serializing them into files
- Reloading them when needed

This approach avoids complex databases while maintaining speed and clarity.

---

7. Branches as Named Pointers

Branches are implemented as:

branch_name → commit_id

Stored using:

- `unordered_map<string, string>`
- Or individual branch files

This mirrors real Git where branches are **lightweight pointers**, not copies of history.

Summary Table

Data Structure	Purpose	Git Equivalent
Files	Persistent storage	Object database
unordered_map	Fast lookup	Index & refs
Linked list	Commit history	Commit DAG
Hashes	Identity & integrity	SHA hashes
Strings/vectors	Support	Blob contents

Final Insight

The strength of this MiniGit project lies not in copying Git line-by-line, but in **capturing Git's architectural philosophy** using simple, powerful data structures. By combining `unordered_map`, file-based persistence, and linked-list–style commit history, the project demonstrates that **complex systems can be built from simple ideas**.

This foundation also makes MiniGit easily extensible into an end-to-end system with networking, compression, and visualization in the future.

Design Decisions in MiniGit

1. Philosophy Behind Design Decisions

MiniGit was intentionally designed as a **learning-oriented version control system**, not as a full replacement for Git. Every design decision was guided by three core principles:

1. **Simplicity over completeness** – The goal was clarity, not feature overload.
2. **Transparency over abstraction** – Data is stored in readable formats to make internals visible.
3. **Conceptual alignment with Git** – Even simplified features mirror real Git ideas.

These principles ensured that MiniGit acts as both a functional tool and an educational system.

---

2. Why File-Based Storage Was Chosen

Decision

Use the **local file system** as the primary persistence layer instead of databases or in-memory-only structures.

Reasoning

- Version control systems fundamentally manage files.
- File-based storage closely resembles Git's `.git` directory.
- It allows inspection using simple OS commands.

Benefits

- Easy debugging and verification
- Persistence across program executions
- No dependency on external libraries

Trade-Offs

- Slower than in-memory storage
- Requires careful file handling

Despite trade-offs, this choice reinforces conceptual clarity.

---

3. Why Text Files Instead of Binary Formats

Decision

Store metadata (commits, branches, index) as **plain text files**.

Reasoning

- Human-readable format aids learning
- Easy manual inspection
- No need for serialization libraries

Example

A commit file contains:

parent: <hash>
branch: main
message: Initial commit
file1.txt: <hash>

Comparison to Git

Git	MiniGit
Binary compressed objects	Plain text objects
High efficiency	High readability

---

4. Hash-Based Identification of Objects

Decision

Use hashing to uniquely identify commits and file contents.

Reasoning

- Ensures uniqueness
- Avoids name conflicts
- Models Git’s content-addressable storage

Why std::hash

- Built-in C++ support

- Fast execution
- Adequate for learning-level collision risk

Trade-Off

- Not cryptographically secure (unlike SHA-1/SHA-256 in Git)

This trade-off is acceptable since MiniGit is educational, not security-critical.

---

5. Commit Structure as a Linked List

Decision

Each commit stores a reference to its **parent commit hash**.

Why This Model

- Enables linear history traversal
- Mimics Git commit DAG in simplified form
- Simplifies log traversal

Benefits

- Easy implementation
- Clear mental model
- Efficient backward traversal

Limitations

- No merge commits yet (single parent only)

This design lays the foundation for future DAG expansion.

---

6. Why unordered_map Was Preferred

Decision

Use `unordered_map` for staging area and metadata mapping.

Reasoning

- O(1) average lookup time
- Natural key-value mapping (filename → hash)

- Order is irrelevant

Comparison

Data Structure	Reason Rejected
vector	Slow lookup
map	O(log n) overhead

This choice optimizes performance while keeping code readable.

---

7. Branch Representation Design

Decision

Branches stored as:

branch_name: commit_hash

Why This Works

- Simple pointer-based abstraction
- Mirrors Git's `refs/heads`
- Allows instant branch switching

HEAD Management

- `HEAD.txt` stores active branch name
- Checkout updates HEAD only

This separation simplifies branch logic.

---

8. CLI-First Design Choice

Decision

Expose all functionality via a **command-line interface**.

Reasoning

- Matches Git's usage model
- Encourages scripting and automation
- Minimal UI overhead

Commands Implemented

- init
- add
- commit
- log
- branch
- checkout

This approach ensures extensibility.

---

9. Error Handling Strategy

Decision

Fail gracefully with informative messages.

Examples

- Invalid command → "Unknown command"
- Empty history → "No commits yet"

Why This Matters

- Improves user experience
- Aids debugging
- Prevents silent failures

---

10. Scalability and Future Design Choices

Although MiniGit is simple, it was designed with extensibility in mind:

- Commit DAG support
- Merge commits
- Diff and status commands
- GUI or Web interface

Current design decisions ensure minimal refactoring for future growth.

---

11. Summary of Key Design Decisions

Decision	Justification
File-based storage	Persistence + Git alignment
Text metadata	Readability
Hash identifiers	Uniqueness
Linked-list commits	Simple history
unordered_map	Performance
CLI interface	Authentic VCS usage

These decisions collectively form a clean, educational, and professional system.

4. Complexity Analysis

This section analyzes the time and space complexity of the MiniGit system. The goal is not only to quantify performance but also to justify why the chosen design is suitable for an educational, lightweight version-control system. Wherever relevant, comparisons are made with real Git to help the reader understand trade-offs.

4.1 Understanding the Scope of Complexity

MiniGit is intentionally designed as a **single-user, local, file-based system**. Unlike real Git, it does not need to:

- Handle millions of objects
- Support distributed networking
- Optimize for large repositories

Therefore, complexity analysis focuses on:

- File size (number of tracked files)
- Number of commits in a branch
- Number of branches

Let:

- **F** = number of files staged or tracked
- **C** = number of commits in a branch
- **B** = number of branches

4.2 Time Complexity of Commands

4.2.1 `init`

Operations performed:

- Create directories (`.minigit`, `objects`, `commits`)
- Create small metadata files (`HEAD`, `branches.txt`, `index.txt`)

Time Complexity:

- O(1)

Explanation:
The number of operations does not depend on repository size. Initialization is constant-time.

Comparison with Git:
Real Git also performs initialization in constant time, though it creates more internal structures.

4.2.2 `add <file>`

Operations performed:

- Read file content
- Hash file content
- Write object file
- Append entry to index

Time Complexity:

- O(S), where S is the size of the file

Explanation:

- Reading and hashing a file is linear in file size
- Writing the object is also linear

If multiple files are added:

- $O(S_1 + S_2 + ... + S_n)$

Comparison with Git:
Git also hashes file contents (SHA-1 / SHA-256). The conceptual complexity is identical.

4.2.3 `commit -m "message"`

Operations performed:

- Read all staged files from index
- Construct commit metadata
- Hash combined data
- Write commit file
- Update branch head
- Clear index

Time Complexity:

- O(F)

Explanation:
Each staged file entry is read once. Metadata handling is constant-time.

Comparison with Git:
Git internally creates tree objects, blobs, and commit objects, making it more complex but still linear in number of staged files.

4.2.4 `log`

Operations performed:

- Read branch head
- Traverse commits using parent pointers
- Read each commit file

Time Complexity:

- O(C)

Explanation:
Each commit is visited exactly once, similar to traversing a linked list.

Why this is acceptable:

- Branch histories are typically linear
- MiniGit avoids graph traversal complexity

Comparison with Git:
Git performs graph traversal (DAG), which is more complex but optimized using commit graphs and caching.

4.2.5 `branch <name>`

Operations performed:

- Check if branch exists
- Append new branch entry

Time Complexity:

- O(B)

Explanation:
The branches file is scanned linearly to detect duplicates.

Comparison with Git:
Git uses references stored as files or packed refs, enabling near O(1) access.

4.2.6 `checkout <branch>`

Operations performed:

- Verify branch existence
- Update HEAD file

Time Complexity:

- O(B)

Explanation:
Branch lookup is linear due to file-based storage.

Note:
MiniGit does not restore working directory state (unlike real Git), keeping complexity minimal.

4.2.7 `merge <branch>`

Operations performed:

- Read commit data from two branches
- Store file mappings in unordered_map
- Detect conflicts
- Write merged index
- Create merge commit

Time Complexity:

- $O(F_1 + F_2)$

Explanation:
Each file entry from both branches is processed once.

Why unordered_map matters:

- Average O(1) lookup for conflict detection

Comparison with Git:
Git performs a three-way merge using a common ancestor, which is significantly more complex.

4.3 Space Complexity

4.3.1 Object Storage

- Each unique file content is stored once
- Duplicate contents reuse the same hash

Space Complexity:

- O(U), where U = number of unique file contents

This mimics Git's deduplication strategy.

4.3.2 Commit Storage

- Each commit stores metadata and file mappings

Space Complexity:

- O(C × F)

Trade-off:

- Simple storage format
- Easy to debug and inspect

4.3.3 In-Memory Structures

- `unordered_map` during merge
- Temporary strings during hashing

Space Complexity:

- O(F)

Memory usage is minimal and released after command execution.

4.4 Why These Complexities Are Acceptable

MiniGit prioritizes:

- Clarity over extreme optimization
- Educational value
- Deterministic behavior

For repositories with:

- Tens or hundreds of files
- Linear commit history

Performance is more than sufficient.

4.5 Comparison Summary with Real Git

Feature	MiniGit	Real Git
Commit traversal	Linked list	DAG
Merge	Two-way	Three-way
Branch lookup	Linear	Near O(1)
Storage	Plain files	Packed & compressed

4.6 Future Optimizations

If MiniGit were extended:

- Replace branch file with hash map
- Cache commit metadata
- Introduce tree objects
- Implement delta compression

These would reduce time and space complexity but at the cost of simplicity.

4.7 Conclusion

The complexity of MiniGit is intentionally simple, predictable, and aligned with its learning goals. While it does not match Git's industrial-scale optimizations, it faithfully demonstrates the **core algorithmic principles of version control systems**.