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.