Software Architecture Specification: AI-Powered Natural Language Linux Terminal

February 16, 2026

1 Software Architecture Style: Layered Architecture

The software architecture which we selected for the AI-Powered Natural Language Linux Terminal is the **Layered Architecture** style. This arhcitecture organizes the system into distinct horizontal layers, where each layer has a specific responsibility and communicates only with the layers directly above or below it.

1.1 Understanding Layered Architecture

Layered architecture, often referred to as n-tier architecture, is a de facto standard for enterprise applications. It is built on the principle of **Separation of Concerns**, where the software is decomposed into units that handle specific logic types.

In this model, components within a specific layer only deal with logic pertinent to that layer. For instance, the UI layer never interacts with the database or the OS shell directly; instead, it sends a request to the Application layer. This "closed layer" strategy ensures that a change in one layer (such as updating the UI theme) does not impact the logic in the execution or infrastructure layers.

1.2 A. Justification: Categorization and Granularity

The system follows this architecture to isolate user interaction from high-risk system execution. The granularity of these components is defined as follows:

- **Presentation Layer (UI)**:
  - **Component:** `TerminalUI` Class.
  - **Granularity:** Coarse-grained, user-facing components.
  - **Role:** This layer handles all user interactions, simulates the terminal environment, captures keystrokes, and renders text-based output. It acts as a pure view, containing no business logic regarding command interpretation.
- **Business Logic Layer (Application Core)**:
  - **Component:** `CommandController`, Safety Validation, and LLM Logic.
  - **Granularity:** Medium-grained, orchestrating components.
  - **Role:** Functioning as the "brain," it coordinates with external AI services to interpret intent and validates commands for safety to prevent destructive operations.

- **Infrastructure Layer (Data/Execution)**:
  - **Component:** `CommandExecutor` and OS Shell Interface.
  - **Granularity:** Fine-grained, atomic execution units.
  - **Role:** This layer handles low-level execution, such as spawning C++ subprocesses via `QProcess` to execute Bash commands directly on the host operating system.

1.3 B. Strategic Suitability for the Project

Layered Architecture is the optimal choice for this project based on several non-functional requirements:

- **Security**: The architecture explicitly places a **Safety Validation** layer between the non-deterministic output of the LLM and the system shell. This ensures that no AI-generated command reaches the machine without being vetted.
- **Maintainability**: The modular approach allows for independent updates. The UI can be swapped for a web interface, or the LLM provider can be changed, without necessitating a rewrite of the core execution logic.
- **Performance**: Execution is kept local and low-latency. By utilizing `QProcess` in the infrastructure layer, validated commands are executed immediately by the machine.
- **Parallel Development and Mocking:** This architecture enables different team members to work on the UI, the Logic, and the Execution layers simultaneously. For example, the Infrastructure layer can be unit-tested with hardcoded shell commands while the AI integration is still in development, accelerating the overall project timeline.
- **Simplicity**: For a small development team, a layered application is significantly less complex to manage and debug than distributed microservices.
- **Fault Isolation and Error Resilience:** By isolating the `LLM API Connector` within the Business Logic layer, the system ensures that external failures (such as network timeouts or malformed API responses) do not crash the terminal interface. The `CommandController` can intercept these exceptions and gracefully revert to standard shell mode without compromising the user session.

2 Key Application Components

The following components constitute the functional building blocks of the software:

1. **User Interface (Terminal UI):** Manages the visual appearance of the Linux terminal. It captures input strings and renders output to the user.
2. **Command Controller:** Serves as the central coordinator. It accepts raw input from the UI and routes it through the validation and execution pipeline.
3. **LLM API Connector:** Formats natural language input into prompts, communicates with AI models, and parses responses into executable shell commands.
4. **Safety Validator:** Analyzes commands generated by the AI against predefined rules to block high-risk or destructive operations.

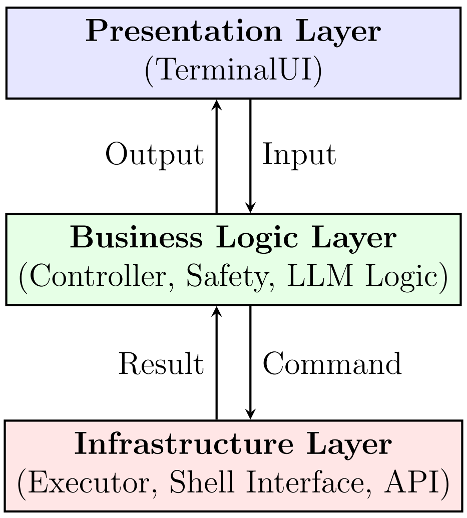

Figure 1: Layered Architecture Model for SE Terminal

5. **Command Executor:** Interacts directly with the OS. It spawns child processes and returns the exit status to the controller.

6. **Session History Manager:** Stores a log of all interactions during the session to provide context for sequential natural language requests.