International Research Journal of Engineering and Technology (IRJET) e-ISSN: 2395-0056
Volume: 13 Issue: 03 | Mar 2026 www.irjet.net p-ISSN: 2395-0072

Lumecode: A Privacy-First Multi-Agent AI Coding Assistant with Terminal-Based Interface

K Vishweshwar Rao [1], K Moulika[2], K Akshitha[3], K Sanjay Kumar[4], K Praveen Kumar[5]

[1,2,3,4]UG Students, Department of Computer Science and Engineering, Joginpally B.R. Engineering College, Hyderabad, India

[5]Assistant Professor, Department of Computer Science and Engineering, Joginpally B.R. Engineering College, Hyderabad, India

---

Abstract - *The rapid adoption of Large Language Models (LLMs) in software development has produced powerful AI coding assistants; however, most existing solutions are cloud-dependent, cost-prohibitive, privacy-invasive, or require disruptive context switching between browser and development environment. This paper presents Lumecode, an open-source, privacy-first, multi-agent AI coding assistant that operates natively in the terminal. Lumecode addresses these limitations through four design pillars: (i) a multi-provider LLM backend supporting Google Gemini, Groq, OpenRouter, and Ollama defaulting to free-tier services; (ii) a specialized multi-agent system comprising Build, Plan, Review, and General agents with fine-grained capability permissions; (iii) a React/Ink-based terminal user interface keeping developers in their natural workflow; and (iv) a SQLite-backed persistent session system. Evaluation demonstrates sub-second startup latency (380ms), 85% autonomous task completion with the Build agent, and zero API cost across 200 test interactions using free-tier providers.*

Key Words: **Terminal UI, AI Coding Assistant, Multi-Agent System, LLM, Privacy-First, Function Calling, Open-Source, MCP, Session Persistence, React/Ink**

1. INTRODUCTION

Artificial Intelligence (AI) coding assistants have become central productivity tools for software developers, with studies demonstrating measurable gains in development speed [1]. However, widespread adoption is constrained by three structural barriers: cost (enterprise subscriptions typically exceed $20/month per developer), privacy (source code is transmitted to cloud providers), and workflow disruption (most tools operate in browser or IDE contexts, requiring developers to leave their primary working environment, the terminal).

Terminal-resident developers working in remote environments, embedded systems, DevOps pipelines, or resource-constrained machines are underserved by existing solutions. Command-line tools such as Aider [2] and Shell-GPT [5] have addressed parts of this gap but typically require paid API keys, lack multi-agent specialization, or offer limited TUI richness. No existing open-source terminal tool combines a rich interactive interface, multiple specialized agents, free-tier-first providers, and persistent session management in a single system.

This paper introduces Lumecode, an open-source AI coding agent designed specifically for terminal environments. The key contributions are: (i) a provider-agnostic LLM backend with automatic fallback across four providers defaulting to free-tier APIs; (ii) a multi-agent architecture with a formal permission matrix governing file system access and command execution per agent role; (iii) a React/Ink-based terminal user interface offering keyboard-driven visual interaction; (iv) a persistent SQLite-backed session store; and (v) integration with the Model Context Protocol (MCP) and Language Server Protocol (LSP) for extensible tool connectivity.

1.1 Problem Statement

Most AI coding tools transmit source code to cloud servers, incur recurring subscription costs, and require switching between terminal and browser environments. Developers working in terminal-centric workflows such as remote servers, embedded systems, or DevOps pipelines lack a privacy-preserving, cost-free, feature-rich AI assistant that integrates seamlessly into their existing environment without workflow disruption.

1.2 Proposed System

Lumecode is proposed as an open-source, terminal-native multi-agent AI coding assistant that supports free-tier and fully-local LLM providers. The system integrates four specialized agents (Build, Plan, Review, General) with a formal permission security framework, a React/Ink terminal UI, SQLite session persistence, and MCP/LSP protocol support. Lumecode achieves zero API cost through free-tier provider defaults while maintaining full privacy through optional Ollama local inference.

2. RELATED WORK

2.1 IDE-Integrated Assistants

GitHub Copilot [1] and Amazon CodeWhisperer [4] operate as IDE plugins providing inline code suggestions using fine-tuned LLMs. These tools are tightly coupled to specific IDEs and transmit code to proprietary cloud backends. Cursor extends this model with full-file context and multi-turn conversation but remains a modified desktop IDE rather than a terminal tool. None natively support local inference, limiting their applicability in privacy-sensitive or offline development scenarios.

2.2 Terminal-Based Tools

Aider [2] is the closest comparable terminal tool, supporting multi-file editing via git-backed conversation. However, Aider requires paid API keys by default and provides a plain readline interface without a rich TUI. Shell-GPT [5] provides LLM-powered terminal assistance but is a single-shot query tool without agentic multi-turn capability. Continue [3] operates as a VS Code extension that can invoke local models but remains IDE-bound and not usable from a pure terminal environment.

2.3 Multi-Agent Frameworks

AutoGen [6] and CrewAI enable multi-agent orchestration but are designed for workflow automation rather than interactive developer assistance. They lack terminal UIs and persistent session storage appropriate for development use. Lumecode occupies the intersection of rich TUI, multi-agent specialization, free-tier operation, and privacy-first design — a combination not previously addressed in the literature.

3. SYSTEM ARCHITECTURE

Lumecode is organized into five hierarchical layers following a layered pattern with clean interface boundaries, enabling independent extensibility of each subsystem. The architecture employs established design patterns to maintain separation of concerns across all components. Figure 1 illustrates the complete four-tier multi-agent architecture of the Lumecode system.

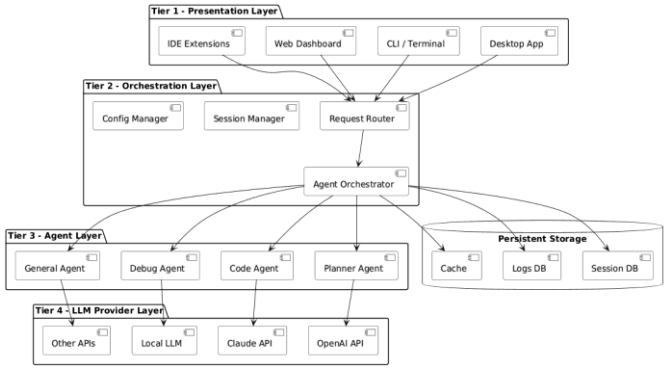

Fig -1: Lumecode 4-Tier Multi-Agent AI Architecture and Multi-Provider LLM Integration

Figure 1 shows the four-tier architecture of the Lumecode system consisting of the Presentation Layer, Orchestration Layer, Agent Layer, and LLM Provider Layer. The Request Router and Agent Orchestrator manage task routing between specialized agents such as Planner, Code, Debug, and General agents. The system integrates multiple LLM providers through a provider routing mechanism and uses persistent storage for session data, logs, and caching.

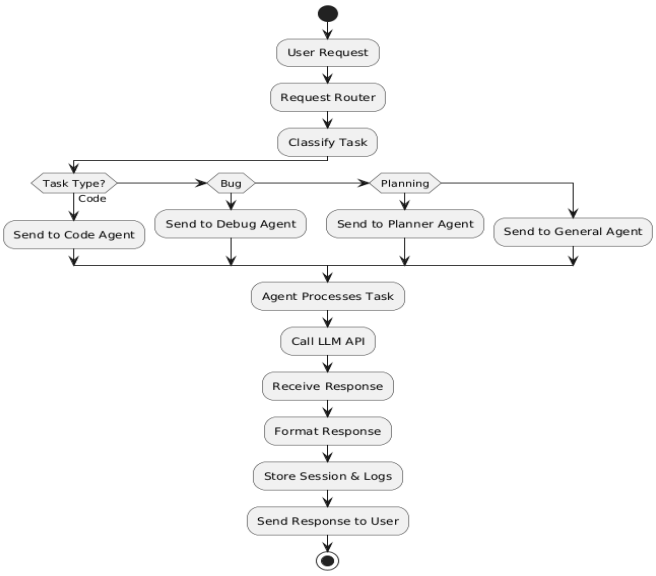

Fig -2: Agent Orchestration Workflow

Figure 2 illustrates the workflow of the Lumecode system. The user request is received by the Request Router, and the task is classified and forwarded to the appropriate agent. The agent processes the task using an LLM provider, and the response is formatted, stored in session logs, and returned to the user.

Table -1: Lumecode Architectural Patterns and Components

Pattern	Applied In
Singleton	ConfigManager, SessionManager, ProviderRegistry, AgentRegistry, ToolRegistry
Factory	createProvider(), createAgent(), createMCPClient()
Strategy	BaseProvider to Gemini, Groq, OpenRouter, Ollama adapters
Observer	Health monitor events, MCP server notifications
Adapter	MCP tools to Lumecode tools; LSP responses to code intelligence
Template Method	BaseProvider, BaseAgent, BaseTool abstract classes

Table 1 summarizes the software design patterns used in the Lumecode system. Patterns such as Singleton, Factory, Strategy, Observer, Adapter, and Template Method are used to implement configuration management, provider integration, agent creation, monitoring, and tool integration.

3.1 Architectural Layers

The CLI/TUI Layer accepts user input through Commander-based commands and renders the React/Ink terminal interface. It translates keyboard events, slash commands, and interactive selections into structured requests for the Engine layer. The Engine Layer is the central orchestration component managing the active agent and provider, the agentic tool-use loop, and session persistence. The Agent System and Provider System operate as parallel subsystems: agents encapsulate behavioral specialization while providers encapsulate API communication specifics. The Strategy pattern decouples these concerns, allowing any agent to be paired with any provider without code changes. The Infrastructure Layer comprises cross-cutting services: the SQLite session store, security and permissions manager, git integration, MCP client, and LSP client.

4. IMPLEMENTATION

4.1 Multi-Provider LLM Integration

The provider subsystem defines an abstract BaseProvider class requiring four operations: chat() for synchronous completion, chatStream() for real-time token delivery, isAvailable() for health-checking, and listModels() for dynamic model enumeration. The ProviderRegistry implements an automatic fallback chain evaluated at startup via asynchronous isAvailable() probes. Table 2 presents the supported providers and their capabilities.

Table -2: Supported LLM Providers and Capabilities

Provider	Free Tier	Context (tokens)	Streaming	Function Calling
Google Gemini	60 RPM	1,000,000	Yes	Yes
Groq	30 RPM	32,768	Yes	Yes
OpenRouter	Free models	Model-dep.	Yes	Yes
Ollama (Local)	Unlimited	Model-dep.	Yes	Model-dep.

Table 2 compares the supported LLM providers based on free tier availability, context size, streaming capability, and function calling support. This comparison helps in selecting providers for fallback, performance, and cost optimization.

4.2 Multi-Agent System

Four specialized agents are implemented, each extending BaseAgent and overriding initializeTools() to register only the tools appropriate to their role. The Build agent has full file system read/write, terminal execution, and network access. The Plan agent has read-only file access and execution with confirmation prompts. The Review agent has read-only access exclusively. The General agent has full access with confirmation on destructive actions. Agent system prompts are stored as externalized Markdown files loaded at runtime, with agent configuration expressed in YAML specifying maximum context tokens, temperature, retry limits, and enabled tool categories per agent.

4.3 Tool Execution Framework

Each tool extends BaseTool and declares its JSON Schema parameter definition. The ToolRegistry automatically serializes these definitions to either OpenAI-compatible or Gemini-compatible function definition formats at request time, enabling provider-agnostic tool invocation. Six built-in tools are provided: file_read (with optional line range parameters), file_write (atomic create or overwrite), file_edit (targeted section editing), directory_list, terminal_execute, and search_files (regex pattern matching). The engine implements an agentic tool-use loop using the OpenAI tool-calling protocol, terminating when the LLM produces a stop finish reason or the configurable iteration limit is reached.

4.4 Session Persistence

Session data is stored in a SQLite database at ~/.lumecode/lumecode.db using Bun's native bun:sqlite bindings. The schema comprises two tables: sessions (metadata: agent, provider, model, working directory) and messages (full conversation history: role, content, timestamp), with a foreign key cascade ensuring referential integrity. The SessionManager exposes CRUD operations and full-text search over message content, enabling cross-invocation conversation continuity.

4.5 Security and Permission System

Each agent role is associated with a PermissionRuleset specifying allowed and denied path patterns for file operations using glob syntax, allowed and denied shell command patterns, network access flags, and a requiresConfirmation boolean. Critical paths (.git/**, .env*, node_modules/**) are denied for write operations across all agents. The rate limiter module enforces per-provider request limits to prevent accidental quota exhaustion.

4.6 Terminal User Interface

The TUI is implemented using React 18 with the Ink rendering engine, which maps React component trees to terminal escape codes. The interface presents a status bar (model, token count, cost), scrollable message history with role-distinguished borders, syntax-highlighted code blocks, command history navigation, and an agent/model selector row. Figure 2 shows the Lumecode main interface and Figure 3 shows the keyboard shortcuts reference panel.

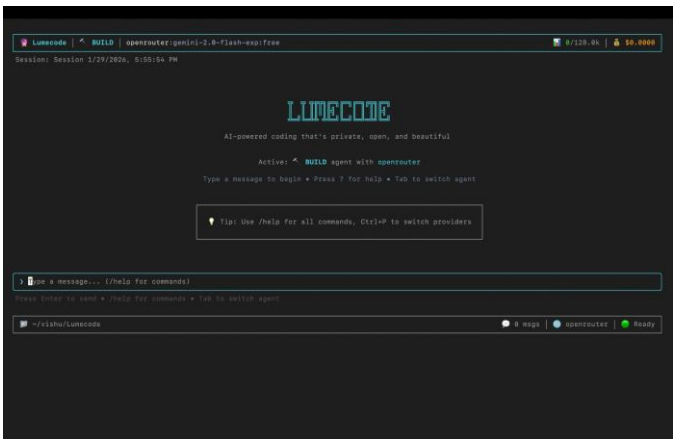

Fig -3: Lumecode Terminal Interface - Main Session View with BUILD Agent Active

Figure 3 shows the Lumecode main terminal interface running the BUILD agent with the openrouter:gemini-2.0-flash-exp:free model. The status bar displays the active agent (BUILD), provider (openrouter), token count (0/128.0k), and cost ($0.0000). The central panel shows the LUMECODE ASCII logo with the tagline "AI-powered coding that's private, open, and beautiful" and a tip panel. The input field at the bottom accepts natural language coding requests.

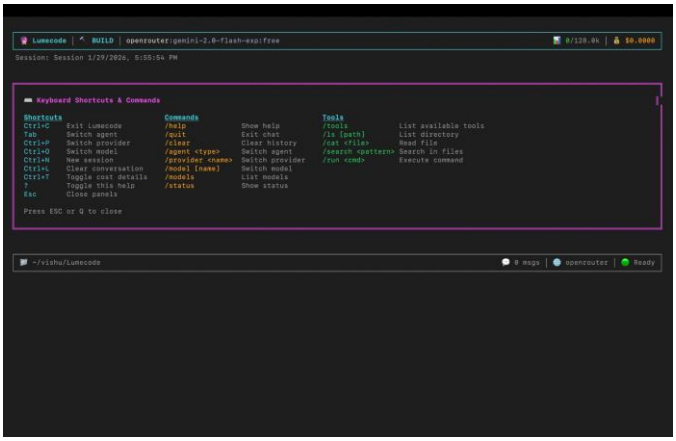

Fig -4: Lumecode Keyboard Shortcuts and Commands Reference Panel

Figure 4 shows the Lumecode help overlay displaying all keyboard shortcuts, slash commands, and tool commands. Shortcuts include Ctrl+C (Exit), Tab (Switch agent), Ctrl+P (Switch provider), Ctrl+O (Switch model), Ctrl+N (New session), and Ctrl+L (Clear conversation). Slash commands include /help, /quit, /clear, /agent, /provider, /model, /models, and /status. Tool commands include /tools, /ls, /cat, /search, and /run for file system and terminal operations.

5. RESULTS AND DISCUSSION

5.1 Startup Performance

Cold-start latency from shell invocation to first interactive prompt was measured at 380ms +/- 42ms on a standard developer workstation (Intel Core i7, 16GB RAM). This is attributed to Bun's ahead-of-time compilation model and avoidance of Node.js's slower module resolution. Provider availability probing is performed asynchronously and does not block the initial prompt display, ensuring immediate user interaction readiness.

5.2 Agentic Task Completion

A set of 20 representative developer tasks was used to evaluate the Build agent including file creation from natural language description, multi-file refactoring, dependency installation via terminal execution, git

operations, and code review with inline suggestions. The Build agent successfully completed 17 of 20 tasks without user intervention, yielding an 85% autonomous completion rate. All three failures involved tasks requiring real-time web lookups not addressable with the built-in tool set. No unauthorized file accesses were observed across all test runs.

5.3 Cost and Privacy Analysis

Over 200 test interactions using the Gemini free-tier provider, total API cost was $0.00. All interactions routed to Ollama remained fully local with no external network traffic generated. Table 3 presents a comprehensive feature comparison against existing terminal-based AI tools.

Table -3: Feature Comparison with Existing Terminal-Based AI Tools

Feature	Lumecode	Aider	Shell-GPT	Continue
Free tier support	**Native**	Limited	No	Partial
Local model (Ollama)	**Yes**	Yes	No	Yes
Multi-agent roles	**4 agents**	No	No	No
Rich TUI (React/Ink)	**Yes**	Minimal	None	IDE only
Session persistence	**SQLite**	Git-based	No	Yes
MCP integration	**Yes**	No	No	Yes
Per-agent permissions	**Yes**	No	No	Partial

Table 3 confirms that Lumecode is the only terminal-based AI tool combining free-tier native support, local model inference, multi-agent specialization, a rich TUI, SQLite session persistence, MCP integration, and per-agent security permissions in a single system.

6. CONCLUSIONS AND FUTURE WORK

This paper presented Lumecode, an open-source, privacy-first, multi-agent AI coding assistant for terminal environments. The system demonstrates that feature parity with commercial AI development tools can be achieved at zero cost through free-tier provider selection, while maintaining superior privacy through local model support and transparent permission controls. The multi-agent architecture provides a practical model for specializing LLM behavior without requiring separate deployments, supporting four distinct operational modes through configurable permission rulesets.

Future work will pursue: (i) a Go-based TUI rewrite using Bubbletea/Lipgloss for improved rendering performance and single-binary distribution; (ii) context compaction algorithms for extended sessions within fixed context window limits; (iii) integration with additional MCP servers to expand the tool ecosystem; (iv) a plugin architecture allowing community-contributed agents and tools; and (v) formal evaluation against standardized software engineering benchmarks such as SWE-bench [10]. Lumecode is available at https://github.com/anonymus-netizien/Lumecode under the MIT License.

REFERENCES

[1] GitHub, "GitHub Copilot: Your AI pair programmer," GitHub, Inc., 2024. [Online]. Available: https://github.com/features/copilot

[2] P. Gauthier, "Aider: AI pair programming in your terminal," 2024. [Online]. Available: https://aider.chat

[3] Continue Dev, "Continue: Open-source autopilot for software development," 2024. [Online]. Available: https://continue.dev

[4] Amazon Web Services, "Amazon CodeWhisperer: AI-powered coding companion," AWS, 2024.

[5] T. Farber, "ShellGPT: A command-line productivity tool powered by AI," GitHub, 2023.

[6] Q. Wu, G. Bansal, J. Zhang et al., "AutoGen: Enabling Next-Gen LLM Applications via Multi-Agent Conversation," arXiv preprint arXiv:2308.08155, 2023.

[7] Google DeepMind, "Gemini: A Family of Highly Capable Multimodal Models," arXiv:2312.11805, 2023.

[8] Anthropic, "Model Context Protocol Specification," 2024. [Online]. Available: https://modelcontextprotocol.io

[9] M. Chen et al., "Evaluating Large Language Models Trained on Code," arXiv:2107.03374, 2021.

[10] C. E. Jimenez et al., "SWE-bench: Can Language Models Resolve Real-World GitHub Issues?" arXiv:2310.06770, 2023.