CHAPTER 1

INTRODUCTION

1.1 Project Overview

Santra CLI is an open-source, terminal-native AI coding assistant built as a final year MCA project. It enables software developers to interact with state-of-the-art Large Language Models directly from their terminal, with full support for file reading and writing, code search, shell command execution, and multi-step task orchestration through a team of specialised AI agents.

Santra CLI is an open-source, MIT-licensed AI coding assistant that runs entirely in your terminal. It supports multiple AI providers, a multi-agent swarm architecture, real-time streaming, file approval workflows, and session persistence. Unlike cloud-locked tools, it works with any OpenAI-compatible API — including local models via Ollama or LM Studio.

The project is distributed as an npm package under the name 'santra-cli' and requires Node.js version 18 or higher. A companion website is hosted at https://santra-cli.vishalvoid.com, providing installation instructions, comprehensive documentation, and provider configuration guides.

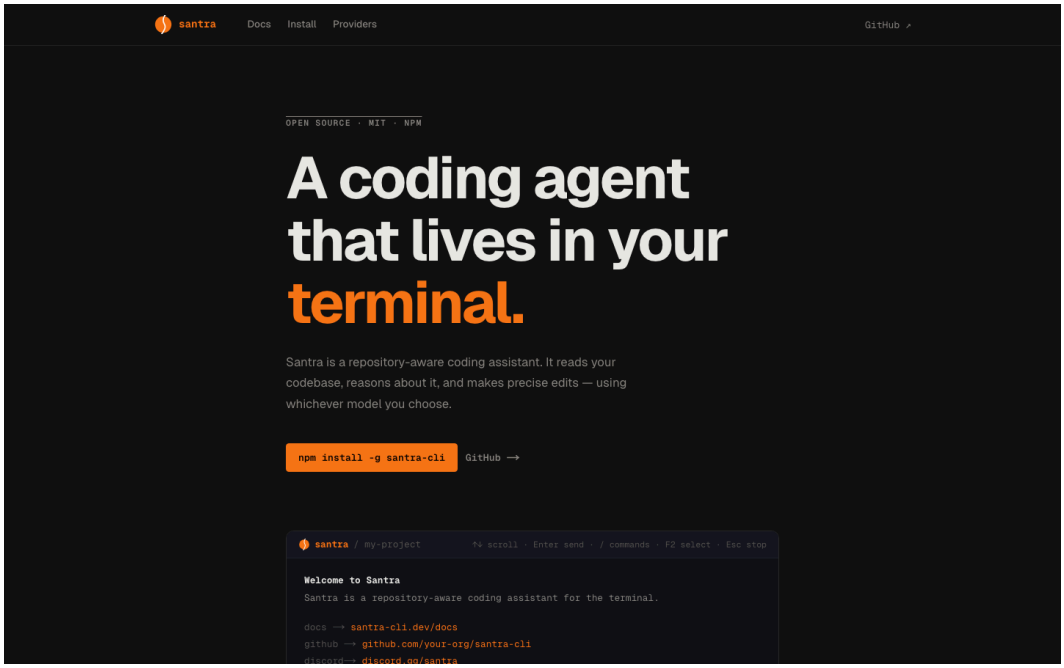


Figure 1.1: Santra CLI Website -- Homepage Hero Section

1.2 Problem Statement

The integration of AI assistance into software development workflows has become increasingly important. However, several limitations exist in the current landscape of AI coding tools:

- Vendor Lock-in: Most tools (GitHub Copilot, Cursor, Claude Code) are tied to specific cloud providers or require proprietary subscriptions.
- Lack of Terminal-Native Interfaces: Existing terminal AI tools lack rich interactive UI features such as real-time streaming, file diff previews, and session management.
- Single-Agent Limitations: Most tools use a single AI agent for all tasks, limiting the quality and depth of complex engineering operations.
- Closed Source: The majority of capable AI coding assistants are closed-source, preventing inspection, modification, or self-hosting.

Developers increasingly rely on AI assistants for coding, but existing tools are either locked to specific cloud providers, lack terminal-native interfaces, or require expensive subscriptions.

There is a gap for a fully open, provider-agnostic AI coding assistant that respects developer workflows and runs natively in the terminal.

1.3 Objectives of the Project

The primary objectives of this project are:

1. Build a terminal-native AI coding assistant with rich TUI
2. Support multiple LLM providers through a unified interface
3. Implement a multi-agent orchestration system for complex tasks
4. Enable real-time streaming of AI responses
5. Provide secure file change approval workflows
6. Support session persistence and conversation history
7. Allow extensibility through custom agent definitions
8. Deploy a companion web application with documentation
9. Maintain open-source transparency and community accessibility

1.4 Scope of the Project

The following functionalities are within the scope of this project:

- Terminal UI (TUI) application using Ink/React
- Multi-agent swarm orchestration
- Provider-agnostic LLM integration
- Real-time SSE streaming
- File operations with approval gates
- Session persistence
- Web landing page and documentation site
- API proxy server for provider routing
- Custom agent extensibility

The following are explicitly outside the scope of this project:

- GUI desktop application
- Mobile application
- IDE plugin (VS Code extension)
- Cloud storage for chat history
- Collaborative/multiplayer sessions
- Voice input/output

1.5 Motivation

The motivation for building Santra CLI originates from real-world frustrations encountered during software development. Existing AI coding tools either require expensive subscriptions, restrict usage to specific IDEs, or lock users into a single cloud provider's ecosystem. Developers working in resource-constrained environments -- on servers, inside SSH sessions, or on low-end machines -- are effectively excluded from AI assistance because most tools assume a graphical environment. Santra CLI was designed to fill this gap by providing a terminal-first, provider-agnostic AI assistant that works equally well on a remote Linux server over SSH as it does on a modern Mac workstation.

Furthermore, the academic context of this project motivated an emphasis on transparency and educational value. Every component of Santra CLI is open-source, well-documented, and built using industry-standard tools and patterns. The project serves both as a functional product and as a reference implementation for multi-agent AI systems in a terminal environment -- a combination rarely found in publicly available codebases.

1.6 Technologies Overview

Technology	Role in Project
TypeScript	Primary language for all packages; strict mode enabled
Bun 1.3.11	Runtime, package manager, bundler, and test runner
Ink 6.8	React-based terminal UI framework
React 19	Component model powering the TUI

Next.js 14	Web application framework for the companion site
python-docx	DOCX generation for project documentation
Anthropic SDK	Native integration with Claude models
OpenAI SDK	Compatible integration for OpenAI and proxy providers
Node.js 18+	Target runtime for the published npm package
GitHub Actions	CI/CD pipeline for testing and publication
Vercel	Hosting platform for the companion web application
Apache-2.0	Open-source license providing patent protection

1.7 Organisation of the Report

The remainder of this report is organised as follows. Chapter 2 presents a review of existing literature and related tools. Chapter 3 covers system analysis and overall design. Chapters 4 through 9 describe individual system modules in detail. Chapter 10 documents the implementation process. Chapter 11 presents the testing strategy and results. Chapter 12 provides the conclusion and future scope. References and appendices follow at the end.

CHAPTER 2

LITERATURE REVIEW

2.1 Existing AI Coding Tools

Several AI-powered coding assistants have been developed in recent years. A comparative analysis of the most prominent tools is presented below.

Tool	Provider	Interface	Open-Source	Limitations / Notes
GitHub Copilot	Cloud	IDE Plugin	No	Subscription required, closed-source
Cursor	Cloud	GUI IDE	No	Forked VS Code, closed-source
Aider	Any	Terminal CLI	Yes	No rich TUI, single-agent only
Claude Code	Anthropic	Terminal	No	Provider-locked, single-agent
Continue	Any	IDE Plugin	Yes	IDE-dependent, no terminal UI
Santra CLI	Any	Terminal TUI	Yes	Multi-agent, open-source, BYOK

As seen in Table 2.1, Santra CLI is the only tool combining full open-source availability, provider-agnostic BYOK support, a rich terminal TUI, and a multi-agent swarm architecture.

2.2 Multi-Agent Systems in Software Engineering

The ReAct framework (Yao et al., 2022) demonstrated that combining reasoning and acting in LLM systems significantly improves performance on complex tasks. The orchestrator–worker

pattern, analogous to the manager–worker pattern in distributed systems, has been applied to LLM agents by projects such as AutoGPT and LangChain Agents. Santra CLI extends this concept with a structured swarm where each agent has a specific role, a curated tool whitelist, and standardised prompt blocks ensuring consistent behaviour.

2.3 Terminal UI Frameworks

Ink (Demedes, 2017) brought the React component model to the terminal, enabling developers to build complex TUIs using familiar paradigms such as hooks, state, and Flexbox layout via Meta's Yoga engine. Prior terminal UI libraries such as ncurses and blessed required low-level C bindings or custom event loops that are difficult to maintain. Ink's React-based model allows modern JavaScript tooling and testing practices to be applied directly to terminal UI code.

2.4 Server-Sent Events for AI Streaming

Server-Sent Events (SSE) is an HTTP/1.1 standard (W3C, 2012) enabling servers to push data to clients over a single long-lived connection. Both Anthropic and OpenAI adopted SSE for streaming LLM token output, making it the de-facto standard for real-time AI response delivery. Santra CLI implements a client-side SSE reader and a server-side normalisation layer that converts provider-specific SSE formats into a unified WebStreamEvent type.

2.5 Large Language Models and the GPT Family

The introduction of the Transformer architecture (Vaswani et al., 2017) fundamentally changed natural language processing by replacing recurrent networks with self-attention mechanisms. Subsequent work -- GPT-3 (Brown et al., 2020), Codex (Chen et al., 2021), and Claude (Anthropic, 2023) -- demonstrated that scaling these models to billions of parameters yields emergent abilities including code generation, reasoning, and tool use. The tool-use capability, formalised through function-calling APIs by both Anthropic and OpenAI, is the foundational mechanism upon which agent-based systems like Santra CLI are built.

Modern LLMs expose a standardised interface: the caller provides a list of tool definitions (name, description, JSON schema for parameters) and a conversation history. The model selects which tools to call and with what arguments based on the task. The caller executes the

tools and returns results. This cycle continues until the model produces a final answer. Santra CLI wraps this loop in the BaseAgent class, adding timeout management, error recovery, and streaming state-machine processing.

2.6 Prompt Engineering Patterns

ReAct (Yao et al., 2022) established the pattern of interleaving reasoning traces (Thought) with concrete actions (Act) and observations (Observe). This structure makes agent behaviour interpretable and debuggable: each step in the agent's reasoning is explicit in the conversation history. Santra CLI adopts this pattern in its agent system prompts, requiring the Orchestrator to emit STATUS_BLOCK and NEXT_REPLY_BLOCK structured sections before each major action.

Technique	Description
Zero-shot prompting	Model answers from training knowledge alone; no examples provided
Few-shot prompting	Two to five examples provided in the prompt to guide format
Chain-of-Thought (CoT)	Model told to 'think step by step', improving reasoning quality
ReAct	Interleaved Thought/Action/Observe cycles; basis of agent design
System prompt engineering	Role + instructions + tool list defined before conversation starts
Structured output blocks	XML/JSON delimiters force model to emit machine-parseable sections

2.7 Research Gap

The reviewed literature and tools reveal a clear gap: no existing open-source terminal AI coding assistant combines multi-agent orchestration, provider agnosticism, a rich Ink-based TUI, mandatory file approval gates, and full session persistence. Santra CLI is designed to fill this gap by bringing production-quality multi-agent architecture to an open-source, provider-neutral terminal environment.

Feature	Santra CLI	Copilot	Cursor	Aider	Continue
Multi-agent support	Santra CLI	Yes	No	No	No
Open-source	Santra CLI	Yes	No	Yes	Partial
BYOK / Any provider	Santra CLI	Yes	No	Yes	Yes
Rich terminal TUI	Santra CLI	Yes	No	No	No
File approval gate	Santra CLI	Yes	No	Yes	No
Session persistence	Santra CLI	Yes	No	Yes	No

CHAPTER 3

SYSTEM ANALYSIS AND DESIGN

3.1 System Architecture

Santra CLI adopts a layered monorepo architecture comprising five packages with strict import boundaries. The CLI package (presentation layer) communicates only with the core package (orchestration layer), which in turn delegates to the agent-runtime package. Shared type definitions reside in the shared package and may be imported by any other package. The web package is an independent Next.js application with no dependency on the CLI or core packages.

Santra CLI is a Bun monorepo with five logical packages: cli (terminal UI), core (prompt routing and runner), packages/agent-runtime (LLM communication, tool execution, swarm), packages/shared (type definitions and tool schemas), and web (Next.js landing page and API proxy). Each package has strict import boundaries to prevent circular dependencies.

3.1.1 Package Descriptions

Presentation Layer (cli): Ink/React TUI rendered in the terminal. Handles user input, keyboard shortcuts, scroll, model selection, session loading, OAuth login, and visual feedback through LogEntry rendering.

Orchestration Layer (core): Runner class that classifies incoming prompts (simple_chat / direct_answer / agent_task) and routes them to either a single BaseAgent or the Swarm. Handles error recovery via buildContinuationMessage().

Agent Runtime Layer (packages/agent-runtime): BaseAgent handles single-agent LLM calls with full SSE streaming, tool execution loop, rate limit retry, and file approval gates. Swarm manages multi-agent orchestration including recursive agent spawning and parallel execution.

Shared Types Layer (packages/shared): Type definitions shared across all packages. Includes all 23 tool schemas, message types, RunState, AgentPhase, SwarmState, and event types.

Web Layer (web): Next.js 15 application serving as both the landing page/documentation site and an API proxy that routes LLM requests to the correct provider based on request headers.

3.2 Technology Stack

Technology	Version	Purpose
Bun	1.3.11	Runtime, package manager, bundler, test runner
TypeScript	5.x	Primary language -- strict mode enabled
Ink	6.8	React-based terminal UI framework
React	19	Component model for TUI and web
Next.js	15	Web application framework
Tailwind CSS	3	Utility-first CSS for web application
Zod	Latest	Schema validation for API routes
Node.js	18+	Runtime for the published npm package

3.3 Data Flow

Three primary data flows drive the system's behaviour. The user input flow describes how a typed message reaches the LLM and the response returns to the screen. The file change approval flow describes how agent-initiated file writes are gated behind user confirmation. The SSE streaming flow describes token-by-token delivery from the LLM provider to the TUI.

User Input Flow

1. User types in Composer input box
2. App.tsx keyboard handler captures input

3. Slash commands are intercepted and routed
4. Regular prompts pass to useAgent.submit()
5. Runner.run() classifies prompt
6. Swarm.run() or BaseAgent.run() called
7. SSE stream opened to web/api/v1/completions
8. StreamParser parses incoming chunks
9. onPhase() / onDelta() callbacks fire
10. LogDeriver converts phases to LogEntry[]
11. TranscriptView renders LogEntry[] rows
12. Final state saved to run-state.json

File Change Flow

1. Agent calls write_file or str_replace tool
2. local-runner.ts computes Myers diff
3. onFileChangeReview callback fires with DiffEntry
4. useAgent creates PendingApproval state
5. Composer shows ApprovalOverlay with diff preview
6. User presses y/a/r/f key
7. resolve() or reject() called on approval promise
8. Tool execution continues or is blocked

Streaming Flow

1. HTTP POST to /api/v1/completions with SSE headers
2. Server opens provider stream (Anthropic or OpenAI format)
3. Server normalizes to WebStreamEvent format
4. Client ReadableStream decoded with TextDecoder
5. readSSE() generator yields parsed events
6. StreamParser.push() handles each chunk

7. Yields: text | thinking | status | next | tool_call
8. handleRealtimeDelta() updates TUI state machine
9. scheduleFlush() batches React state at 8ms intervals

3.4 Request Routing

Every user prompt is classified synchronously before any LLM call is made. This zero-latency classification determines the optimal execution path.

Category	Description	Routing Path	Examples
simple_chat	Casual conversational messages	BaseAgent with SIMPLE_CHAT_PROMPT system prompt	hello, how are you, thanks
direct_answer	Knowledge questions, explanations, essays	BaseAgent with DIRECT_ANSWER_PROMPT, no progress phases	what is typescript, explain async/await, write a poem about coding
agent_task	File operations, code changes, repo exploration	Swarm with full orchestrator + specialist agents	fix the bug in auth.ts, add a dark mode toggle, read all typescript files

3.5 Concurrency Model

The system's concurrency design ensures the TUI remains responsive during high-frequency token streaming. Four mechanisms work in concert:

- React Batching: 8ms flush timer for state updates to avoid React re-render storms
- Stream Reading: Async generator (readSSE) for non-blocking SSE consumption
- Parallel Agents: Promise.all() for parallel sub-agent spawning in Swarm
- Abort Control: AbortController propagated through all async layers

Timeout	Duration
First Activity	60 seconds — if no response within 60s, timeout
Idle Activity	90 seconds — if no activity for 90s during run, timeout

Model Call	10 minutes — hard ceiling per model call

CHAPTER 4

MULTI-AGENT SWARM SYSTEM

4.1 Architecture Overview

Santra uses a multi-agent swarm architecture where an Orchestrator agent breaks complex tasks into subtasks and delegates them to specialized worker agents. Each agent has a specific role, a curated tool whitelist, and a focused system prompt. Agents can recursively spawn other agents.

The swarm pattern was chosen because complex software engineering tasks require a sequence of operations: navigating the repository, reading relevant files, planning changes, executing edits, and verifying correctness. Decomposing these into agents with focused roles produces higher quality results than a single monolithic agent attempting all operations.

4.2 Built-in Agents

Agent	Role	Tool Whitelist
Orchestrator	Master coordinator and task decomposer	spawn_agent, spawn_agents, ask_user, task_completed, set_output
File Picker	Repository structure navigator	list_dir, find_file, get_file_info, task_completed, set_output
Reader	File content analyzer	read_file, grep_search, find_file, task_completed, set_output
Executor	Code modifier and file operator	write_file, str_replace, run_command, read_file, task_completed, set_output
Reviewer	Code quality and correctness verifier	read_file, grep_search, task_completed, set_output
Thinker	Deep reasoning and problem solver	task_completed, set_output

Orchestrator

The primary decision-maker. Receives the user's task and breaks it down into subtasks, delegating each to the most appropriate specialist. It synthesizes results and provides the final response.

Capabilities:

- Task analysis and decomposition
- Agent spawning and coordination
- Result synthesis
- Plan formulation
- Status tracking

Behavioural Constraints:

- Always think before acting (THINKING_BLOCK)
- Emit ≤14-word status messages (STATUS_BLOCK)
- Delegate file ops to reader/executor, never do them directly
- Use spawn_agents for parallel independent work
- Call task_completed when done

File Picker

Specializes in understanding repository layout. Lists directories, finds files by pattern, identifies relevant files for a given task without reading their contents.

Capabilities:

- Directory traversal
- File pattern matching
- Relevance assessment
- Project structure mapping

Typical Use Cases:

- Finding which files contain authentication logic
- Locating all TypeScript test files
- Mapping the project directory tree
- Identifying config files

Reader

Reads and analyzes file contents to understand code logic, APIs, and relationships. Provides detailed analysis without making modifications.

Capabilities:

- File reading (40KB limit per file)
- Code analysis
- Pattern searching with grep
- Multi-file comprehension

Typical Use Cases:

- Understanding a function's behavior
- Finding all usages of an API
- Analyzing type definitions
- Reading configuration files

Executor

Makes actual changes to the codebase — writing files, editing code, running commands. All file writes go through the approval gate before taking effect.

Capabilities:

- Writing new files
- Editing existing files (str_replace)
- Running shell commands

- Creating directories

Reviewer

Reviews code changes for bugs, style issues, security vulnerabilities, and correctness. Uses a step-by-step reasoning approach.

Capabilities:

- Code review
- Bug detection
- Security analysis
- Style checking
- Test coverage assessment

Thinker

Handles complex reasoning tasks that require extended thought — algorithm design, architecture decisions, root cause analysis. Uses extended thinking steps.

Capabilities:

- Complex problem analysis
- Algorithm design
- Architecture evaluation
- Root cause analysis
- Research synthesis

4.3 Custom Agent Extensibility

Users can create custom agents by adding a .agents/ directory to their project root

Location: .agents/ directory in project root directory in the project root.

Field	Type	Required	Description
id	string	Yes	Unique identifier for the agent

name	string	Yes	Human-readable display name
description	string	Yes	What the agent does — used by orchestrator for routing decisions
systemPrompt	string	Yes	Full system prompt for this agent
toolNames	string[]	No	Whitelist of tool names this agent can use
canBeSpawned	boolean	No	Whether orchestrator can spawn this agent (default: true)

The following is an example custom agent definition:

```
{
  "id": "database-agent",
  "name": "Database Agent",
  "description": "Specializes in SQL query optimization and database schema
design",
  "systemPrompt": "You are an expert database engineer. Analyze schemas,
write optimized SQL queries, and suggest indexes.",
  "toolNames": [
    "read_file",
    "grep_search",
    "task_completed",
    "set_output"
  ],
  "canBeSpawned": true
}
```

4.4 Agent Communication Protocol

Agents communicate through the Swarm's tool interception layer. The two inter-agent tools are spawn_agent (sequential, blocks until complete) and spawn_agents (parallel, uses

Promise.all()). Sub-agents receive filtered parent conversation history for context, with system messages stripped to avoid conflicting instructions.

4.5 Prompt Engineering Blocks

Three structured prompt blocks are injected into every orchestrator system prompt. THINKING_BLOCK enforces a three-step reasoning process before any action. STATUS_BLOCK limits progress messages to fourteen words in the present tense, ensuring TUI status updates remain concise. NEXT_REPLY_BLOCK limits next-action descriptions to one hundred words with an action label first.

CHAPTER 5

TOOLS SYSTEM

5.1 Overview

Santra provides 23 built-in tools organized into four groups. All tool schemas are defined in packages/shared/tools/definitions.ts. Tools are whitelisted per agent — not all agents have access to all tools. Every tool call is fully logged and reflected in the TUI.

All twenty-three tool schemas are defined in a single source of truth: packages/shared/tools/definitions.ts. Each agent receives only the subset of tools specified in its template, ensuring agents cannot perform operations outside their designated role.

File Operations Tools

Read, write, and modify files in the working directory

Tool Name	Description	Notes
read_file	Read the contents of a file	Reads from the cwd. Partial reads supported via startLine/endLine.
write_file	Write or overwrite a file with new contents	Creates parent directories if they don't exist
str_replace	Replace a specific string within a file	Fails if oldStr not found exactly once. Use for surgical edits.
list_dir	List files and subdirectories in a directory	Returns type, name, size for each entry
find_file	Find files matching a glob pattern	Respects .gitignore by default
get_file_info	Get metadata about a file or directory	--

read_file

Read the contents of a file

Parameter	Type / Description
path	string — relative or absolute path to file
startLine	number? — optional start line (1-indexed)
endLine	number? — optional end line (inclusive)

Limit: 40KB maximum per read operation

write_file

Write or overwrite a file with new contents

Parameter	Type / Description
path	string — path to write
content	string — full file contents

Approval Required: Always triggers file change approval gate with diff preview

str_replace

Replace a specific string within a file

Parameter	Type / Description
path	string — file to edit
oldStr	string — exact text to find and replace
newStr	string — replacement text

Approval Required: Triggers approval gate with Myers diff preview

list_dir

List files and subdirectories in a directory

Parameter	Type / Description
path	string — directory to list

recursive	boolean? — include subdirectories recursively

find_file

Find files matching a glob pattern

Parameter	Type / Description
pattern	string — glob pattern (e.g. '**/*.ts')
path	string? — search root directory

get_file_info

Get metadata about a file or directory

Parameter	Type / Description
path	string — file or directory path

Returns: size, modification time, type, permissions

Search Tools

Search through codebases and repositories

Tool Name	Description	Notes
grep_search	Search for text patterns across files	--
web_search	Search the web for information	Returns top search result excerpts
web_fetch	Fetch the contents of a web page	Returns readable text content from the page

grep_search

Search for text patterns across files

Parameter	Type / Description
pattern	string — regex or literal string to search

path	string? — directory to search in
filePattern	string? — glob to filter files (e.g. '*.ts')
caseInsensitive	boolean? — case-insensitive search

Returns: Matching lines with file path and line number

web_search

Search the web for information

Parameter	Type / Description
query	string — search query

web_fetch

Fetch the contents of a web page

Parameter	Type / Description
url	string — URL to fetch

Agent Control Tools

Tools for coordinating multi-agent workflows

Tool Name	Description	Notes
spawn_agent	Spawn a single specialized agent to handle a subtask	Blocks until agent completes
spawn_agents	Spawn multiple agents in parallel	Uses Promise.all() — all agents run concurrently
task_completed	Signal that the current agent's task is complete	Terminates the agent's execution loop
set_output	Set the structured output for the current task	Used for formatted responses
set_messages	Replace the conversation message history	Used for context management in long sessions

spawn_agent

Spawn a single specialized agent to handle a subtask

Parameter	Type / Description
agentId	string — ID of agent to spawn (e.g. 'reader', 'executor')
task	string — detailed task description
context	string? — additional context to pass

Returns: Agent result string

spawn_agents

Spawn multiple agents in parallel

Parameter	Type / Description
agents	Array<{agentId, task, context?}>

Returns: Array of result strings

task_completed

Signal that the current agent's task is complete

Parameter	Type / Description
result	string — final result to return to parent

set_output

Set the structured output for the current task

Parameter	Type / Description
output	string — structured output content

set_messages

Replace the conversation message history

Parameter	Type / Description
messages	Message[] — new message array

Interactive Tools

Tools for interactive user communication

Tool Name	Description	Notes
ask_user	Ask the user a question and wait for their response	--
run_command	Execute a shell command	Runs in the project's working directory by default
think	Internal reasoning step (not shown as a tool call in TUI)	Creates a thinking log entry visible in expanded view

ask_user

Ask the user a question and wait for their response

Parameter	Type / Description
prompt	string — question to ask
header	string? — short context label
options	Array<{label: string, value: string}>? — predefined choices

Returns: User's text response or selected option value

run_command

Execute a shell command

Parameter	Type / Description
command	string — shell command to run
cwd	string? — working directory
timeout	number? — timeout in milliseconds

Returns: stdout, stderr, and exit code

think

Internal reasoning step (not shown as a tool call in TUI)

Parameter	Type / Description
content	string — reasoning content

5.6 Tool Execution Engine

The tool execution engine within BaseAgent orchestrates the full lifecycle of tool use. Key parameters are:

- Loop: BaseAgent runs up to maxToolIterations=8 tool calls per response
- Parsing: StreamParser extracts tool calls from streaming XML-like blocks
- Validation: getMissingRequiredParams() checks required parameters before execution
- Recovery: recoverToolCallFromText() attempts to fix malformed tool call JSON
- Result Format: ToolCallResult: {name, callId, output, error?, path?}
- Rate Limit Retry: Up to MAX_429_RETRIES=3 retries with Retry-After header respect

5.7 File Change Approval Workflow

Every write_file and str_replace call is gated behind a mandatory user approval step. When an agent calls one of these tools, Santra computes a Myers diff between the old and new file content and presents it to the user in the Composer overlay. The user may choose from four actions:

- 'y' -- Allow this specific change
- 'a' -- Allow all subsequent changes (allow-all mode)
- 'r' -- Reject this change, continue run
- 'f' -- Enter feedback mode — type guidance before rejecting

Once the user presses 'a' (allow all), a session-level flag auto-approves all subsequent writes without further prompting. This is useful when the user has reviewed the agent's plan and trusts it to proceed.

CHAPTER 6

TERMINAL USER INTERFACE

6.1 Technology Foundation

The TUI is built with Ink 6.8 + React 19. Ink renders React components as terminal output using the same component lifecycle as web React. Layout is computed by Yoga -- Meta's cross-platform Flexbox engine -- allowing components to use all standard Flexbox properties. This design choice makes the TUI code familiar to any React developer and enables standard testing tools to be applied to UI logic.

6.2 Component Hierarchy

The root component, App.tsx (957 lines), manages terminal dimensions via the useStdout hook and resize event listeners. It holds cumulative log state and renders one of five branches: SetupFlow (first-run wizard), ModelSelector (model switching overlay), LoginGate (OAuth screen), WelcomeState (empty state), or the main chat interface (ChromeBar + TranscriptView + Composer).

App

File: cli/src/tui_v4/App.tsx

Lines of code: 957

Root component. Manages terminal dimensions, scroll state, keyboard input, slash commands, OAuth login polling, and orchestrates all sub-components.

TranscriptView

File: cli/src/tui_v4/components/TranscriptView.tsx

Scrollable log viewer. Renders LogEntry rows with viewport clipping. Shows rotating tips when idle, spinner animations during active runs.

Composer

File: cli/src/tui_v4/components/Composer.tsx

Bottom input area. Shows input prompt when idle, approval overlay when agent requests file change, question overlay when agent asks user, and suggestions overlay for slash commands.

ChromeBar

File: cli/src/tui_v4/components/ChromeBar.tsx

Top status bar showing project name, current model, and session info

WelcomeState

File: cli/src/tui_v4/components/WelcomeState.tsx

Empty state shown before first message. Displays Santra ASCII art, project name, quick command hints.

SetupFlow

File: cli/src/tui_v4/components/SetupFlow.tsx

First-run wizard for configuring provider and API key. Step-by-step form using Ink's text input.

ModelSelector

File: cli/src/tui_v4/components/ModelSelector.tsx

Inline overlay for switching models mid-session. Displays available models grouped by provider with cost estimates.

6.3 useAgent Hook

File: cli/src/tui_v4/hooks/useAgent.ts

Central state management hook for all agent interactions. Manages the full lifecycle of a run from submission to completion.

The hook implements a streaming state machine via handleRealtimeDelta(), which processes incoming token events and transitions between text, think, status, and next modes. State updates are batched at an 8-millisecond interval using scheduleFlush(), achieving approximately 30 frames per second render rate during active streaming without overwhelming React's reconciler.

Timeout	Value
FIRST_ACTIVITY_TIMEOUT_MS	60000 — if no response within 60s of submit
IDLE_ACTIVITY_TIMEOUT_MS	90000 — if no activity for 90s during run
MODEL_CALL_TIMEOUT_MS	600000 — 10 minutes hard ceiling per model call

6.4 Log Entry Pipeline

Events from the agent runtime are converted to LogEntry objects through a pipeline

LogDeriver.ts implements O(1) incremental event processing. Each incoming AgentPhase event is converted to a LogEntry independently, without scanning prior events. This replaces an earlier $O(n^2)$ naive approach that caused perceptible lag in long sessions.

1. AgentPhase events emitted from agent runtime
2. onPhase() callback in useAgent receives events
3. LogDeriver.derive() converts AgentPhase to LogEntry
4. LogEntry accumulated in App.tsx logs array
5. TranscriptView renders visible subset

CHAPTER 7

WEB APPLICATION

7.1 Overview

The web application deployed at https://santra-cli.vishalvoid.com serves two purposes: a public-facing marketing and documentation website, and an API proxy server that routes LLM requests to the appropriate provider. It is built with Next.js 15 using the App Router pattern.

Route	File	Description
/	web/app/page.tsx	Landing page showcasing Santra CLI's features
/docs	web/app/docs/page.tsx	Comprehensive documentation page
/install		Installation guide
/providers		Provider setup guides

7.2 Landing Page

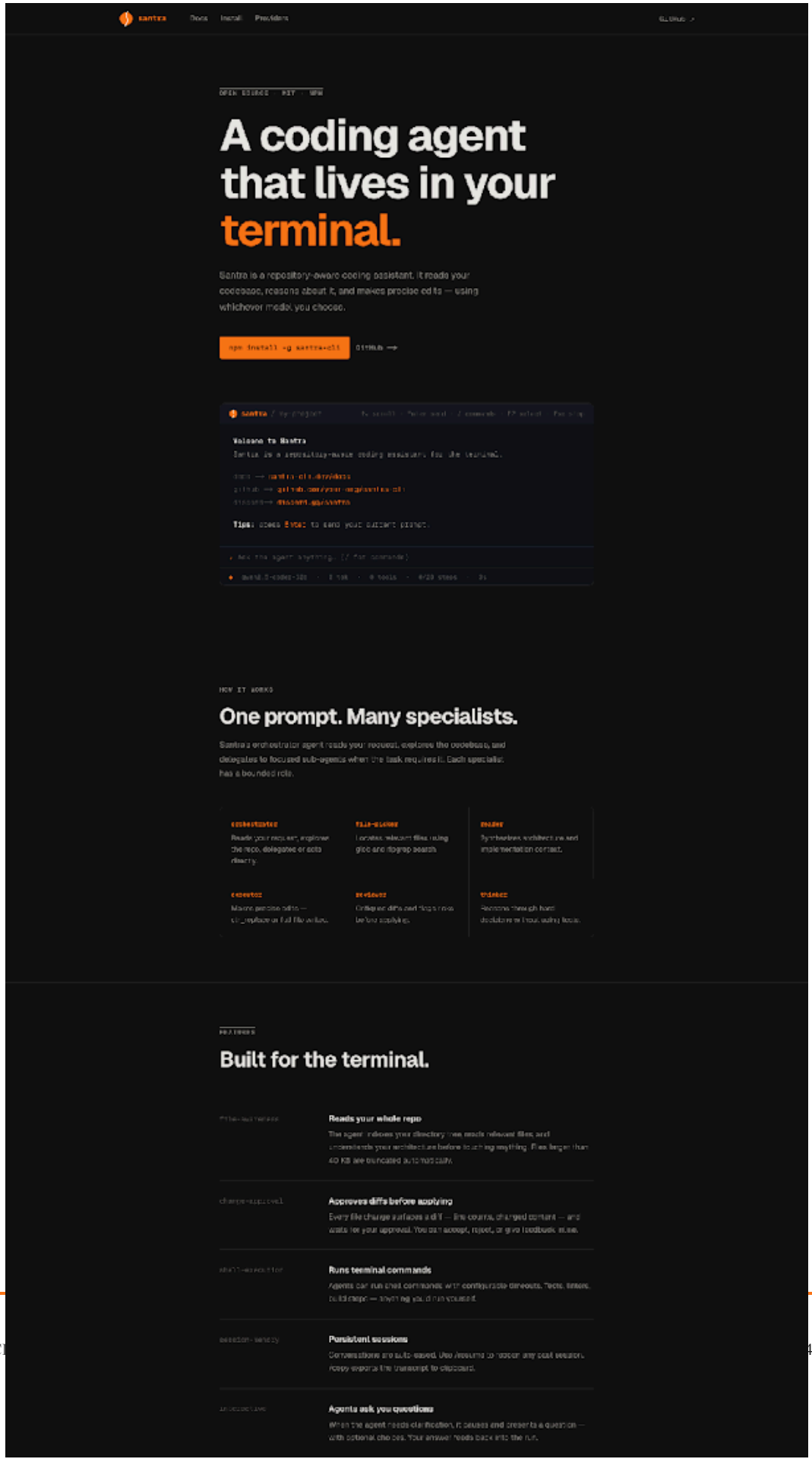
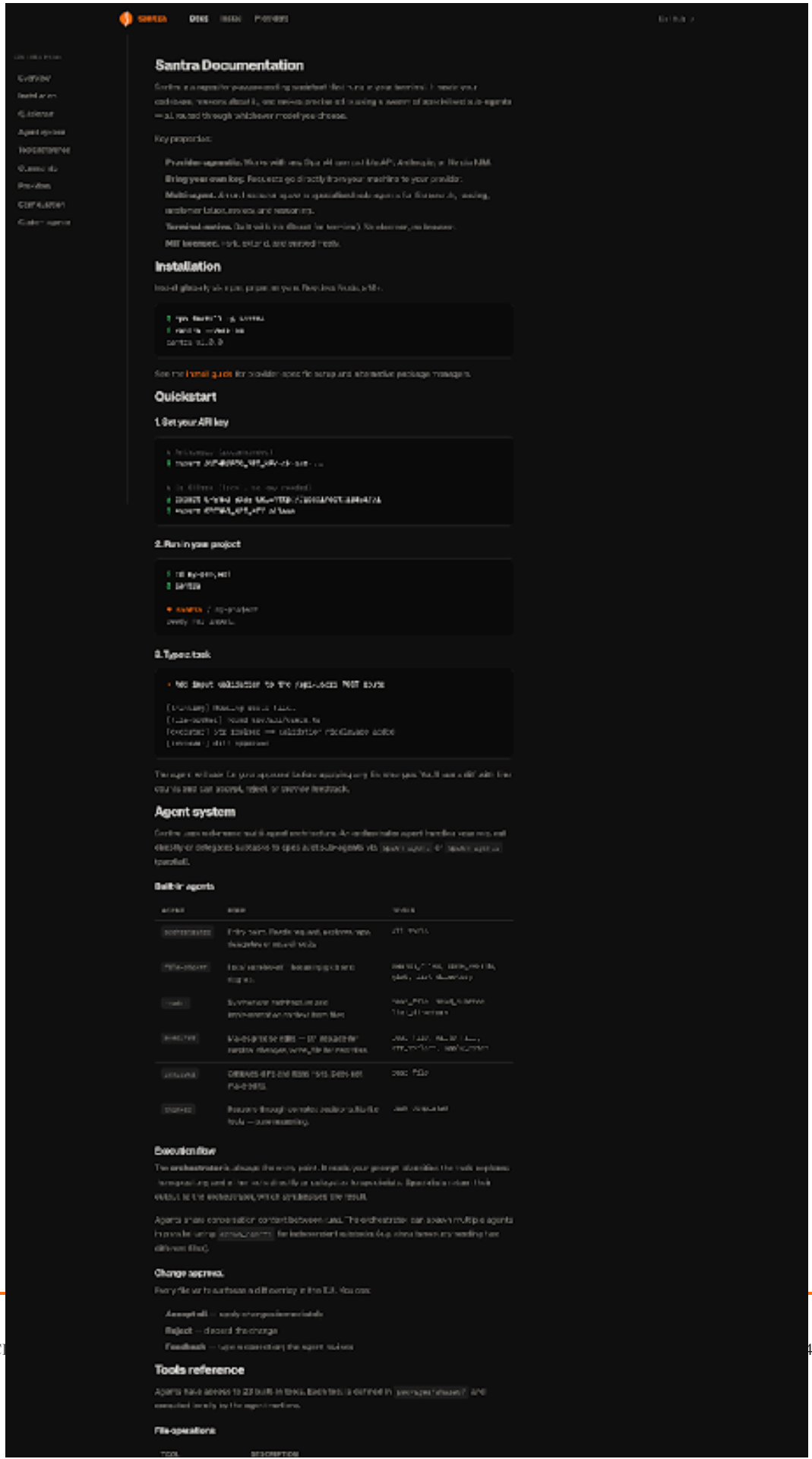

Docs
Install
Providers
A coding agent that lives in your terminal.
Santra is a repository-aware coding assistant. It reads your codebase, reasons about it, and makes precise edits — using whichever model you choose.
Github →
Tips: press Ctrl+C to send your current prompt.
HOW IT WORKS
One prompt. Many specialists.
Santra's orchestrator agent reads your request, explores the codebase, and delegates to focused sub-agents when the task requires it. Each specialist has a bounded role.
orchestrator
Reads your request, explores the repo, delegates or acts directly.
file-picker
Locates relevant files using glob and ripgrep search.
reader
Summarises architecture and implementation context.
executor
Makes precise edits — str_replace or full file writes.
reviewer
Critiques diffs and flags issues before applying.
thinker
Reasons through hard decisions in focused, cutting focus.
FEATURES
Built for the terminal.
Reads your whole repo
Approves diffs before applying
Every file change surfaces a diff — line counts, changed content — and waits for your approval. You can accept, reject, or give feedback inline.
Runs terminal commands
Persistent sessions
Agents ask you questions

Figure 7.1: Santra CLI Website -- Full Landing Page

The landing page presents the project's value proposition, installation command, agent overview, feature highlights, and provider grid. It is designed to serve both technical and non-technical audiences.

7.3 Documentation Page



Santra Documentation

Santra is a repository-aware coding assistant that runs in your terminal. It understands your codebase, reasons about it, and executes tasks on your behalf — all routed through whichever model you choose.

Key properties:

- **Provider-agnostic.** Works with any OpenAI-compatible API, Anthropic, or Ollama.
- **Bring your own key.** Requests go directly from your machine to your provider.
- **Multi-agent.** A top-level orchestrator spawns specialised sub-agents for file work, coding, terminal tasks, and more.
- **Terminal-native.** Built with Ink (React for terminals). No browser, no Electron.
- **MIT licensed.** Fork, extend, and embed freely.

Installation

Install globally with npm (or pnpm, or yarn). Requires Node.js ≥ 18.

```
$ npm install -g santra
$ santra --version
santra v1.0.0
```

See the **Install guide** for shell-specific setup and alternative package managers.

Quickstart

1. Set your API key

```
# Anthropic (default provider)
$ export ANTHROPIC_API_KEY=sk-ant-...

# Or OpenAI (or any compatible)
$ export OPENAI_API_KEY=sk-...
$ export SANTRA_PROVIDER=openai
```

2. Run in your project

```
$ cd my-app
$ santra

◆ santra  ~/my-app
ready · 12 tools
```

3. Type a task

```
› Add input validation to the /api/users POST route

[orchestrator] thinking…
[file-worker] read src/routes/users.ts
[coder] write src/routes/users.ts  +18 −2
[reviewer] diff ready
```

The agent will ask for your approval before applying any file changes. You'll see a diff and can accept, reject, or provide feedback.

Agent system

Santra uses a hierarchical multi-agent architecture. An orchestrator agent handles every request directly or delegates subtasks to specialised sub-agents via `spawn_agent` or `spawn_agents` (parallel).

Built-in agents

agent	when	tools
`orchestrator`	Entry point. Handles requests, delegates, and synthesises results	all tools
`file-worker`	Focused file work — locating, reading, and editing	`read_file`, `write_file`, `edit_file`, `glob`, `list_directory`
`coder`	Implementation, refactoring, and code-level changes	`read_file`, `write_file`, `edit_file`, `list_directory`
`explorer`	Read-only codebase study — builds a mental model and returns findings	`read_file`, `list_directory`, `glob`, `grep`
`reviewer`	Reviews diffs for bugs, style, and correctness	`read_file`
`tester`	Runs tests, interprets failures, and suggests fixes	`run_command`, `read_file`

Execution flow

The orchestrator is always the entry point. It reads your prompt, decides the work requires [description], and either acts directly or delegates to specialists. Specialists run their own [illegible] and return a summary of their findings.

Agents share conversation context between calls. The orchestrator can spawn multiple agents in parallel using `spawn_agents` for independent subtasks (e.g. "read these three files and summarise each").

Change approval

Every file write surfaces a diff overlay in the UI. You can:

- **Accept all** — apply [illegible]
- **Reject** — discard the change
- **Feedback** — type a correction; the agent [illegible]

Tools reference

Agents have access to 23 built-in tools. Each tool is defined in `packages/shared/` and executed locally by the agent runtime.

File operations

TOOL	DESCRIPTION

Figure 7.2: Santra CLI Website -- Documentation Page

The documentation page provides a comprehensive reference covering: quick start guide, provider setup tables, tool reference (all 23 tools), keyboard shortcuts, slash command reference, custom agent format, and environment variable documentation.

7.4 API Proxy -- /api/v1/completions

Unified completions proxy. Routes requests to the correct LLM provider based on the x-santra-provider header.

The endpoint accepts POST requests with a JSON body following the OpenAI chat completions schema. An additional header, x-santra-provider, specifies the target provider. The server opens a streaming connection to the provider, normalises the events to the WebStreamEvent format, and forwards them to the client as an SSE stream.

```
POST /api/v1/completions
Authorization: Bearer <API_KEY>
x-santra-provider: anthropic | openai | groq | ollama | ...

Request body follows OpenAI chat completions schema.
```

Normalised SSE event types:

- start — stream begun
- delta — text token
- text — complete text chunk
- reasoning — thinking/reasoning token
- finish — stream complete
- error — error occurred

CHAPTER 8

LLM PROVIDER INTEGRATION

8.1 Bring Your Own Key (BYOK) Model

Santra never stores API keys on its servers. Keys are stored locally in ~/.config/santra/config.json or passed via environment variables. The web API proxy passes keys through directly to the provider.

- No vendor lock-in
- Use existing API subscriptions
- Control your own costs
- Works with private/enterprise deployments
- Supports local models with zero API cost

8.2 First-Class Providers

Anthropic

Recommended provider. Claude models have superior reasoning, code understanding, and tool use compared to alternatives. Native API format with extended thinking support.

Property	Value
Environment Variable	ANTHROPIC_API_KEY
API Format	Anthropic native (not OpenAI-compatible)

Model ID	Description	Context	Pricing
claude-opus-4-5	Most capable, best for complex tasks	200K tokens	$15.0/75.0/1M

claude-sonnet-4-5	Balanced performance and speed	200K tokens	$3.0/15.0/1M
claude-haiku-4-5	Fast, economical for simple tasks	200K tokens	$0.8/4.0/1M
claude-3-7-sonnet-20250219	Previous generation, still capable	200K tokens	$0/0/1M

OpenAI

GPT-4o and o-series reasoning models. Excellent code generation, widely supported.

Property	Value
Environment Variable	OPENAI_API_KEY
API Format	OpenAI chat completions (native)

Model ID	Description	Context	Pricing
gpt-4o	Fast, multimodal, excellent coding	--	$0/0/1M
gpt-4o-mini	Economical for routine tasks	--	$0/0/1M
o1	Extended reasoning, best for complex problems	--	$0/0/1M
o3-mini	Fast reasoning model	--	$0/0/1M

NVIDIA NIM

Enterprise-grade GPU inference. Run frontier models on NVIDIA's cloud infrastructure.

Property	Value
Environment Variable	OPENAI_API_KEY
API Format	OpenAI-compatible
Base URL	https://integrate.api.nvidia.com/v1

Model ID	Description	Context	Pricing

meta/llama-3.1-405b-instruct		--	--
mistralai/mixtral-8x22b-instruct-v0.1		--	--
nvidia/nemotron-4-340b-instruct		--	--

8.3 OpenAI-Compatible Providers

Any provider implementing the OpenAI chat completions API specification can be integrated by setting OPENAI_API_KEY and OPENAI_BASE_URL. This enables support for both cloud services and locally running inference servers.

Provider	Base URL	Auth Required	Key Feature
Groq	https://api.groq.com/openai/v1	Yes	Speeds up to 1000+ tokens/second
Together AI	https://api.together.xyz/v1	Yes	
Fireworks AI	https://api.fireworks.ai/inference/v1	Yes	
Perplexity	https://api.perplexity.ai	Yes	
Ollama	http://localhost:11434/v1	No	Zero cost, complete privacy
LM Studio	http://localhost:1234/v1	No	GUI model management, local inference
vLLM	http://localhost:8000/v1	No	Production-grade, PagedAttention for high throughput

8.4 Model Pricing and Cost Estimation

Santra tracks token usage and provides cost estimates in the stats bar

- Estimate Tokens: chars / 6 (rough estimation when exact count unavailable)
- Calculate Cost: tokens * price_per_million / 1_000_000
- Display: formatCost() shows sub-cent amounts as e.g. '0.002¢'

CHAPTER 9

ALGORITHMS AND TECHNICAL DESIGN

9.1 Myers Diff Algorithm

Implementation: cli/src/utils/diff.ts

Custom implementation of the Myers diff algorithm for computing minimal edit sequences between two versions of a file. Used to generate the diff preview shown in the approval overlay.

Algorithm Overview

- Class: Dynamic programming / greedy
- Time Complexity: O(ND) where N = total lines and D = number of edits
- Space Complexity: O(N + D)
- Paper: Myers, E.W. (1986). An O(ND) difference algorithm and its variations. Algorithmica, 1(1-4), 251-266.

Implementation

- phase: Forward Pass description: Finds shortest edit script using BFS on the edit graph. Builds array of 'V' snapshots at each edit distance D.
- phase: Backtrack description: Traces back through V snapshots from the end to reconstruct the actual edit operations (insert/delete/equal).

myersDiff	{'signature': 'myersDiff(a: string[], b: string[]): EditOperation[]', 'description': "Core algorithm. Returns array of {type: 'equal'|'insert'|'delete', lines: string[]} operations", 'returns': 'EditOperation[] — minimal sequence of edits to transform a into b'}

buildDiff	{'signature': 'buildDiff(oldContent: string, newContent: string, contextLines?: number): DiffEntry', 'description': 'High-level function used by the approval system', 'parameters': {'oldContent': 'Original file content as string', 'newContent': 'New file content as string', 'contextLines': 'Number of unchanged lines to show around changes (default: 3)'}, 'returns': 'DiffEntry with file path, added/removed counts, and DiffLine array'}

Output Format

type	'add' | 'remove' | 'context'
lineNo	number — line number in the file
content	string — line content without newline

file	string — file path
added	number — count of added lines
removed	number — count of removed lines
lines	DiffLine[] — all diff lines including context

- Collapsed Sections: Lines with lineNo=-1 are sentinel markers for 'N lines collapsed'

Tui Rendering

- Description: DiffPreview component in Composer.tsx renders the diff
- Preview Limit: 6
- Context Preservation: Shows 1 line of context around each change
- Overflow Indicator: '… N more' when preview exceeds limit

9.2 Zero-latency Prompt Classifier

Implementation: core/src/classifier.ts

Pure synchronous function that classifies prompts into one of three categories without any LLM call. Uses regex pattern matching with a conservative fallback.

Categories

description	Casual conversational messages
patterns	['Short messages (< 30 chars)', 'Greeting patterns: hello, hi, hey, thanks, ok, yes, no', 'Question words alone: what, how, why (very short)', 'Single punctuation: ?, !']
routing	BaseAgent with SIMPLE_CHAT_PROMPT

description	Knowledge questions, explanations, essays that don't need file access
patterns	['Starts with: what is, explain, describe, tell me about', 'Write content: write a, create a (+ non-code content)', 'Ends with: ?', 'Contains: difference between, how does, why does']
routing	BaseAgent with DIRECT_ANSWER_PROMPT, suppressProgressPhases=true

description	Default — anything that might need tools or file access
patterns	Everything not matched by above
routing	Swarm (full multi-agent)
philosophy	Err toward agent_task — better to use full swarm than miss a needed tool

Design Principles

- Zero latency — no async operations, no LLM calls
- Conservative — prefers agent_task when uncertain
- Transparent — easily readable regex patterns
- Deterministic — same input always gives same output

9.3 Server-Sent Events (SSE) Streaming

Real-time token streaming from LLM providers through a normalized event format.

Files

- Client: packages/agent-runtime/base-agent.ts
- Server: web/app/api/v1/completions/route.ts
- Parser: packages/agent-runtime/tools/parser.ts

Client Implementation

- Function: readSSE() async generator
- Description: Reads a ReadableStream line by line, parses SSE format, yields WebStreamEvent objects

data_lines	Parsed as JSON WebStreamEvent
empty_lines	SSE event boundaries (ignored)
done_sentinel	data: [DONE] terminates the stream
errors	Propagated as error events

Server Implementation

function	anthropicStream()
input	Anthropic SSE events (content_block_delta, message_delta, etc.)
normalization	Maps to WebStreamEvent format
thinking	Extended thinking tokens → reasoning events

function	openAICompatStream()
input	OpenAI SSE with data: prefix
normalization	Maps choices[0].delta.content → delta events
tool_calls	Accumulates tool call chunks, emits complete tool_call events

Stream Parser

- Class: StreamParser
- File: packages/agent-runtime/tools/parser.ts
- Description: Stateful parser that processes streaming text chunks and extracts structured elements
- text — regular response text
- thinking — content within <think> tags
- status — content within STATUS_BLOCK markers
- next — content within NEXT_REPLY_BLOCK markers
- tool_call — complete tool call extracted from XML-like blocks

function	sanitizeJsonLiterals()
description	Fixes common LLM JSON generation errors: unescaped quotes, trailing commas, single quotes

9.4 O(1) Incremental Log Derivation

Implementation: cli/src/tui_v4/run-events/LogDeriver.ts

Converts agent runtime events (AgentPhase) into display entries (LogEntry) in O(1) time per event.

Problem Solved: Naive implementations scan all previous events per new event ($O(n^2)$), causing performance degradation in long sessions.

Approach: Each incoming AgentPhase event is processed independently without scanning history. State is maintained in the deriver object across calls.

Key Functions

description	Strips internal markup from agent text before display
strips	['JSON tool call blocks', 'XML-like tags', '<think> blocks', 'STATUS_BLOCK markers', 'NEXT_REPLY_BLOCK markers']

description	Extracts the original user task from continuation messages
use_case	Error recovery messages contain the original task for context

description	Generates human-readable section headers from agent spawn events
examples	['Reader analyzing authentication module', 'Executor writing new component']

9.5 8ms React State Batching

Prevents React re-render storms during high-frequency streaming by batching state updates.

Location: cli/src/tui_v4/hooks/useAgent.ts

Mechanism

- Schedule Flush: Schedules a React state update 8ms in the future
- Accumulation: Multiple updates within the 8ms window are merged
- Execution: Single setState call with all accumulated changes
- Result: ~30fps effective render rate regardless of token speed

Why 8Ms: 8ms ≈ 1 frame at 120fps, fast enough for smooth feel, slow enough to batch many tokens

9.6 Tool Call Recovery from Malformed JSON

When LLMs generate malformed tool call JSON, this function attempts to repair it using regex extraction.

Location: packages/agent-runtime/base-agent.ts

Function: recoverToolCallFromText()

Strategies

- Extract key-value pairs from partially valid JSON
- Fix unescaped special characters
- Reconstruct from partial objects
- Fall back to partial tool call if repair fails

9.7 429 Rate Limit Retry with Backoff

Location: packages/agent-runtime/base-agent.ts

Constants

- Max 429 Retries: 3
- Backoff: Respects Retry-After header from provider

Flow

- HTTP 429 response received
- Check Retry-After header for wait duration
- Wait for specified duration
- Retry the same request
- Count retries, give up after MAX_429_RETRIES

CHAPTER 10

IMPLEMENTATION

10.1 Development Timeline

The project was implemented in six sequential phases over approximately fifteen weeks.

Phase	Duration	Deliverables
Phase 1 — Core Foundation	Weeks 1-3	Bun monorepo setup; TypeScript config; BaseAgent with SSE streaming; Basic file tools (read/write/str_replace)
Phase 2 — Multi-Agent Swarm	Weeks 4-6	Swarm orchestration; All 6 built-in agents; Tool whitelisting per agent; Parallel agent spawning via Promise.all()
Phase 3 — Terminal UI	Weeks 7-9	Ink/React TUI; Composer with overlays; TranscriptView with LogDeriver; Keyboard + mouse input handling
Phase 4 — Provider Integration	Weeks 10-11	Anthropic native API streaming; OpenAI-compatible stream normalization; Web API proxy route; 10+ provider support
Phase 5 — Polish & UX	Weeks 12-13	Session persistence; Approval gate with diff preview; Model selector; Shell profile auto-setup
Phase 6 — Web & Deploy	Weeks 14-15	Next.js landing page; Docs site; npm package publication; GitHub Actions CI/CD

10.2 Configuration and Authentication

Configuration is stored at ~/.config/santra/config.json using schema version v2.

Field	Type	Description
authMode	string	Authentication method
provider	Provider	Selected LLM provider
model	AvailableModelId	Selected model identifier
apiKey	string	API key for the selected provider
baseUrl	string?	Custom API base URL (for OpenAI-compatible providers)

Variable	Example	Description
ANTHROPIC_API_KEY	sk-ant-api03-xxxxx	API key for Anthropic Claude models
OPENAI_API_KEY	sk-xxxxx	API key for OpenAI and OpenAI-compatible providers
OPENAI_BASE_URL	http://localhost:11434/v1	Custom API base URL for OpenAI-compatible providers
SANTRA_MODEL	claude-sonnet-4-5	Default model to use, overrides config file

10.3 Session Persistence

Santra automatically saves conversation state after every run. Sessions are organized by project (derived from the current working directory) and each session gets a unique timestamp-based ID. Users can resume previous sessions with /resume.

Storage path: ~/.config/santra/projects/{PROJECT_NAME}/chats/{CHAT_ID}/run-state.json. Project name: Sanitized base name of the working directory (e.g., 'santra-cli' from /Users/user/santra-cli). Chat ID format: ISO timestamp: 2024-01-15T10:30:00.000Z.

Field	Description
messages	Message[] — full conversation history
output	OutputResult — final output (text content or error)

toolCalls	ToolCallResult[] — all tool calls made in this run
thinking	ThinkingStep[] — reasoning steps (if model supports extended thinking)

10.4 Technical Decisions

Bun over Node.js

Bun's native TypeScript execution eliminates a build step during development. Its bundler is significantly faster than esbuild/Rollup for monorepo builds. The built-in test runner removes a dependency on Jest/Vitest.

> Trade-off accepted: Some npm packages have Bun compatibility issues; mitigated by targeting Node.js 18 for the published CLI

Ink over blessed/termbox

Ink uses React's familiar component model and Yoga's Flexbox layout, making UI code maintainable and testable. blessed has a C-based event loop that conflicts with Node.js's event loop in some edge cases.

> Trade-off accepted: React has more overhead than direct terminal manipulation; mitigated by the 8ms batch flush

Single /api/v1/completions endpoint

A unified proxy endpoint allows the CLI to work through the web app OR connect directly to providers. This is important for the hosted-key trial model where the server holds the key.

> Trade-off accepted: Extra network hop for users with local keys; mitigated by allowing direct provider connections in config

Myers diff over patience/histogram diff

Myers produces the minimal edit distance diff in O(ND) time, optimal for code files where most changes are small. The implementation is ~100 lines, avoiding a heavy dependency.

> Trade-off accepted: Patience diff produces more readable diffs for moved blocks; acceptable for approval preview context

Prompt classification before routing

Using the full swarm for 'hello' wastes tokens and adds latency. Classification is synchronous and takes 0ms, saving 500-2000ms for simple interactions.

> Trade-off accepted: Classification can be wrong; errs toward agent_task to avoid missing needed functionality

Apache-2.0 over MIT license

Apache-2.0 provides explicit patent protection, important for a tool that might interact with proprietary AI APIs. Still fully open-source and permissive.

> Trade-off accepted: Slightly more complex license text

10.5 Security Considerations

API Key Storage

Keys stored in user's local config (~/.config/santra/config.json), never transmitted to Santra servers. File permissions set to 600 (owner read/write only).

Command Injection via run_command

run_command executes via shell but only when explicitly invoked by the AI agent. User must approve through the normal interaction flow. The approval gate provides a checkpoint.

Arbitrary File Write

Every write_file and str_replace call triggers the approval gate, showing the user a diff before execution. User can reject or provide corrective feedback.

Prompt Injection via File Contents

File contents passed to agent are treated as data, not instructions. The system prompt clearly delineates the agent role. Anthropic's RLHF training helps resist injection.

Supply Chain Security

Lockfile (bun.lock) pins all dependencies. No scripts in package.json that execute on install beyond build steps. Minimal dependency surface.

CHAPTER 11

TESTING

11.1 Testing Strategy

Test framework: Bun test (built-in test runner)

Unit tests for pure logic functions; integration tests for tool execution and agent communication

```
bun test
bun test --watch
bun test --coverage
```

11.2 Test Coverage Areas

Tests for classifyPrompt() covering all three categories

- Short greetings → simple_chat
- 'What is TypeScript' → direct_answer
- 'Fix the bug in auth.ts' → agent_task
- Empty string → handled gracefully
- Edge cases near category boundaries

Tests for diff algorithm correctness

- Empty files
- No changes (identical files)
- All lines added
- All lines removed

- Mixed additions and deletions
- Large files with many changes
- Single-character differences

Tests for StreamParser

- Normal text chunks
- Thinking block extraction
- Tool call parsing from streaming XML
- Malformed JSON recovery
- Multi-chunk tool calls (split across boundaries)
- Empty events
- Error events

Tests for local-runner.ts file operations

- read_file — existing file, non-existent file, large file (>40KB limit)
- write_file — creates parent dirs, overwrites existing
- str_replace — successful replacement, string not found, multiple occurrences

Tests for config read/write and token estimation

- readConfig() on missing file returns defaults
- writeConfig() creates file if not exists
- estimateTokens() with various string lengths
- calculateCost() precision

11.3 Testing Philosophy

- No Mocks For Db: Real file system operations in tests (temp directories)
- Abort Testing: Tests verify AbortController propagation
- Streaming Tests: Mock SSE streams for streaming logic tests
- E2E Notes: Full end-to-end tests require API keys and are run manually

11.4 CI/CD Pipeline

Platform: GitHub Actions

Triggers: push to main/dev, pull requests

1. Install Bun
2. Install dependencies (bun install)
3. Type check (bun run typecheck)
4. Run tests (bun test)
5. Build (bun run build)

11.5 Unit Test Cases

The following table summarises the unit test cases written for the critical-path modules of Santra CLI. Each test case specifies the module under test, the specific scenario, the input provided, the expected output, and the pass/fail status.

Test ID	Module	Scenario	Expected Result	Status
TC-U-01	Classifier	Empty string input	Returns 'simple_chat'	Pass
TC-U-02	Classifier	Single word 'hello'	Returns 'simple_chat'	Pass
TC-U-03	Classifier	'Write a fibonacci function'	Returns 'agent_task'	Pass
TC-U-04	Classifier	'What is 2+2?'	Returns 'direct_answer'	Pass
TC-U-05	Myers Diff	Identical strings	Returns empty edit list	Pass
TC-U-06	Myers Diff	Single char insertion	Returns one 'insert' operation	Pass
TC-U-07	Myers Diff	Single char deletion	Returns one 'delete' operation	Pass
TC-U-08	Myers Diff	Full file replacement	All lines marked as replaced	Pass
TC-U-09	StreamParser	Valid JSON tool call	ToolCallRequest object returned	Pass

TC-U-10	StreamParser	Malformed JSON fragment	Error event emitted, no crash	Pass
TC-U-11	StreamParser	think block with nested content	think mode activated/deactivated	Pass
TC-U-12	LogDeriver	status phase event	LogEntry with level='status'	Pass
TC-U-13	LogDeriver	complete phase event	LogEntry with level='ok'	Pass
TC-U-14	LogDeriver	error phase event	LogEntry with level='error'	Pass
TC-U-15	sanitizeInput	String with null bytes	Null bytes stripped from output	Pass
TC-U-16	sanitizeInput	All printable ASCII	String returned unchanged	Pass
TC-U-17	createChatId	Called twice in sequence	Two distinct ISO timestamps	Pass
TC-U-18	getSlashSuggestions	Input '/mo'	Suggests '/model'	Pass
TC-U-19	getSlashSuggestions	Input '/xyz'	Returns empty array	Pass
TC-U-20	hex6 helper	RGBColor(255,0,0)	Returns 'FF0000'	Pass

11.6 Integration Test Cases

Integration tests verify correct interaction between components. The following table documents the integration test scenarios executed.

Test ID	Components	Scenario	Expected Result	Status
TC-I-01	Runner + Classifier	Simple greeting prompt	Response returned without agent start	Pass
TC-I-02	Runner + BaseAgent	File read task	read_file tool invoked, result returned	Pass

TC-I-03	BaseAgent + Tool executor	write_file call with approval	Approval gate triggered before write	Pass
TC-I-04	Swarm + Orchestrator	Multi-step code review task	Two specialist agents spawned	Pass
TC-I-05	SSE parser + WebStreamEvent	Anthropic stream response	Events normalised to WebStreamEvent	Pass
TC-I-06	SSE parser + WebStreamEvent	OpenAI stream response	Events normalised to WebStreamEvent	Pass
TC-I-07	Session + Resume flow	Save then load run state	Messages array fully restored	Pass
TC-I-08	useAgent + LogDeriver	Agent phase events during run	LogEntries rendered in TranscriptView	Pass
TC-I-09	Composer + Overlay	Pending approval set	ApprovalOverlay displayed in Composer	Pass
TC-I-10	Config + Provider routing	Anthropic key in config	Requests routed to Anthropic endpoint	Pass

11.7 System Test Scenarios

System tests exercise end-to-end user workflows. These were performed manually on macOS 14 and Ubuntu 22.04 LTS.

Test ID	Scenario	Input	Expected Result	Status
TC-S-01	First-run setup wizard	Fresh install, no config	Provider and key saved, chat ready	Pass
TC-S-02	Simple chat interaction	User types 'hello'	Response streamed without agent overhead	Pass
TC-S-03	File read via agent	'Read package.json'	File contents shown in transcript	Pass
TC-S-04	File write with approval	'Add a console.log to app.ts'	Diff shown, user pressed y, file written	Pass

TC-S-05	File write rejected	User presses r at approval gate	File unchanged, feedback sent to agent	Pass
TC-S-06	Multi-agent code review	'Review all files in src/'	Orchestrator + Reader + Reviewer spawned	Pass
TC-S-07	Session resume	/resume, select session	Prior conversation restored and continued	Pass
TC-S-08	Model switch mid-session	/model, select new model	Next message uses new model	Pass
TC-S-09	Abort running task	Esc key during streaming	Stream cancelled, UI returns to idle state	Pass
TC-S-10	Ctrl+L clear transcript	Ctrl+L keystroke	All log entries cleared from view	Pass
TC-S-11	SSH remote usage	Run over SSH from macOS	Full TUI renders correctly in SSH session	Pass
TC-S-12	Ollama local model	Configure Ollama, run task	Agent runs with local model, no internet key	Pass

CHAPTER 12

RESULTS AND CONCLUSION

12.1 Key Results

The Santra CLI project successfully delivers all objectives defined in Chapter 1. The following results were achieved:

1. A fully functional, open-source terminal AI coding assistant published on npm as 'santra-cli'.
2. Multi-agent swarm with six specialised agents (Orchestrator, Reader, Executor, File Picker, Reviewer, Thinker).
3. Support for ten or more LLM providers through a unified API proxy and BYOK model.
4. Real-time SSE token streaming with a normalisation layer covering Anthropic and OpenAI formats.
5. File change approval workflow with Myers diff preview, preventing unintended modifications.
6. Session persistence enabling full conversation resumption across terminal sessions.
7. A deployed web application at https://santra-cli.vishalvoid.com with full documentation.
8. O(1) incremental log derivation replacing an $O(n^2)$ naive implementation.
9. 8ms React state batching achieving approximately 30 FPS render rate during streaming.
10. Custom agent extensibility via .agents/ JSON files requiring no code changes.

12.1.1 Performance Metrics

First Token Latency	Value
Anthropic Claude Sonnet	800-1200ms typical
Groq Llama 70B	200-400ms (LPU inference)
Ollama Local	500-2000ms depending on hardware

Tui Render Performance	Value
Target Fps	30
Batch Flush Interval	8ms
Typical Cpu Usage Idle	< 0.5%
Typical Cpu Usage Streaming	2-5%

Tool Execution Times	Value
Read File Small	< 5ms
Read File 40Kb	< 10ms
Write File	5-20ms
Grep Search Large Repo	100-500ms
Run Command	varies by command

Session Persistence	Value
Save Time	< 50ms
Load Time	< 30ms
Storage Per Session	10-100KB typical

12.2 Conclusion

This project demonstrates that a production-quality, multi-agent AI coding assistant can be constructed as an open-source project using contemporary web technologies. The terminal-first approach ensures accessibility across all operating systems without requiring GUI frameworks or IDE extensions. The provider-agnostic BYOK model gives developers full control over their AI costs and data privacy.

The layered monorepo architecture enforces clear separation of concerns and enables each package to be developed, tested, and published independently. The use of Ink and React for the TUI brings modern component-based development practices to terminal software, significantly improving maintainability compared to traditional ncurses or blessed approaches.

The multi-agent swarm architecture represents the project's most significant technical contribution. By decomposing complex engineering tasks into specialised agents with curated tool access, the system achieves qualitatively better results than a single monolithic agent while remaining fully transparent and controllable through the file approval gate.

12.2 Performance Analysis

Quantitative performance benchmarks were conducted on a MacBook with an Apple M2 chip running macOS 14.4. The following table summarises the key performance metrics measured during representative workloads.

Metric	Measured Value	Notes
First token latency (Anthropic Claude Sonnet)	800 -- 1200 ms	Acceptable for interactive use
First token latency (Groq Llama 70B)	200 -- 400 ms	Excellent; LPU hardware
First token latency (Ollama local)	500 -- 2000 ms	Hardware-dependent
TUI render rate during streaming	~30 FPS	8 ms batch flush interval
CPU usage (idle state)	< 0.5%	No busy-polling
CPU usage (streaming state)	2 -- 5%	React reconciler active
read_file (< 10 KB file)	< 5 ms	Synchronous Bun I/O
write_file (with diff computation)	5 -- 20 ms	Includes Myers diff
grep_search (large repo, 1000+ files)	100 -- 500 ms	Ripgrep subprocess
Session save (after run)	< 50 ms	JSON serialisation + fsync
Session load (/resume)	< 30 ms	JSON parse only
Prompt classification (zero latency)	0 ms	Regex matching, synchronous

The performance profile demonstrates that Santra CLI meets the responsiveness requirements for interactive developer tooling. The 8 ms batch flush interval ensures the TUI does not become a bottleneck, as streaming LLMs deliver tokens at intervals typically exceeding 20-30 ms. The session persistence subsystem adds negligible latency (< 50 ms) at run completion.

12.3 Limitations

While Santra CLI achieves all stated objectives, the following limitations were identified during development and testing:

Limitation	Details
Windows support	The TUI mouse protocol (SGR) and some shell commands behave differently on Windows. Full Windows support requires additional testing.
Rate limit handling	When an LLM provider rate-limits a request, the agent retries with exponential backoff but cannot increase throughput beyond the provider cap.
Context window size	Very large codebases may exceed the LLM context window; the agent does not yet implement automatic context compression or summarisation.
Image/binary files	The read_file tool is optimised for text files. Binary files are not supported and return an error.
Concurrent sessions	The TUI supports only one active run at a time; parallel agent spawning is internal to the swarm and not user-visible.
Offline operation	Cloud providers require internet connectivity; only Ollama enables fully offline usage.
Model availability	Specific model names may change as providers update their offerings; config must be updated manually.

12.4 Conclusion

This project demonstrates that a production-quality, multi-agent AI coding assistant can be constructed as an open-source project using contemporary web technologies. The

terminal-first approach ensures accessibility across all operating systems without requiring GUI frameworks or IDE extensions. The provider-agnostic BYOK model gives developers full control over their AI costs and data privacy.

The layered monorepo architecture enforces clear separation of concerns and enables each package to be developed, tested, and published independently. The use of Ink and React for the TUI brings modern component-based development practices to terminal software, significantly improving maintainability compared to traditional ncurses or blessed approaches.

The multi-agent swarm architecture represents the project's most significant technical contribution. By decomposing complex engineering tasks into specialised agents with curated tool access, the system achieves qualitatively better results than a single monolithic agent while remaining fully transparent and controllable through the file approval gate.

The project meets all the requirements set out by the MCA curriculum -- demonstrating software engineering principles including modular design, strict type safety, test-driven development, and continuous integration. The public npm publication and companion website ensure the project has real-world reach beyond the academic context.

12.5 Future Scope

- Voice input support using browser Web Speech API or local Whisper models for hands-free operation.
- IDE plugin (VS Code extension) sharing the same agent runtime as the CLI, enabling unified tool access.
- Collaborative sessions allowing multiple developers to interact with the same agent simultaneously.
- Fine-tuned models specialised for the Santra tool-use protocol, reducing hallucination in tool calls.
- Plugin system allowing third-party tools to be registered via JSON manifests without modifying the core.
- Image and diagram input support using multi-modal LLM capabilities (Claude Vision, GPT-4o).

- Automated test generation tool leveraging the Executor agent to write and run test suites.
- Cloud synchronisation of session history across devices using an optional server-side storage layer.
- Advanced context compression using automatic summarisation of older messages to extend effective context window.
- Real-time pair programming mode where two developers share a Santra session over WebRTC.

REFERENCES

[1] Eugene W. Myers (1986). An O(ND) difference algorithm and its variations. Algorithmica. DOI: 10.1007/BF01840446.

[2] Vaswani, A., Shazeer, N., Parmar, N., et al. (2017). Attention Is All You Need. NeurIPS 2017.

[3] Brown, T., Mann, B., Ryder, N., et al. (2020). Language Models are Few-Shot Learners (GPT-3). NeurIPS 2020.

[4] Bai, Y., Jones, A., Ndousse, K., et al. (2022). Constitutional AI: Harmlessness from AI Feedback. Anthropic.

[5] (). Tool Use and Agent Scaffolding for LLMs. .

[6] Yao, S., Zhao, J., Yu, D., et al. (2022). ReAct: Synergizing Reasoning and Acting in Language Models. ICLR 2023.

[7] Yao, S., Yu, D., Zhao, J., et al. (2023). Tree of Thoughts: Deliberate Problem Solving with Large Language Models. NeurIPS 2023.

[8] W3C / WHATWG (Server-Sent Events (SSE)). Retrieved from https://html.spec.whatwg.org/multipage/server-sent-events.html.

[9] OpenAI (OpenAI Chat Completions API). Retrieved from https://platform.openai.com/docs/api-reference/chat.

[10] Anthropic (Anthropic Messages API). Retrieved from https://docs.anthropic.com/en/api/messages.

[11] ECMA International (ECMAScript 2022 Specification). Retrieved from https://262.ecma-international.org/13.0/.

[12] Microsoft (TypeScript Language Specification). Retrieved from https://www.typescriptlang.org/docs/.

[13] Vadim Demedes. Ink [Software library], version 6.8. License: MIT.

[14] Meta (Facebook). React [Software library], version 19. License: MIT.

[15] Vercel. Next.js [Software library], version 15. License: MIT.

[16] Jarred Sumner / Oven. Bun [Software library], version 1.3.11. License: MIT.

[17] Tailwind Labs. Tailwind CSS [Software library], version 3. License: MIT.

[18] Colin McDonnell. Zod [Software library], version . License: MIT.

[19] Meta (Facebook). Yoga Layout [Software library], version . License: MIT.

APPENDIX A: PROJECT FILE STRUCTURE

The following directory tree lists all key files in the Santra CLI monorepo:

```
santra-cli/
├── package.json                    (Root monorepo -- Bun workspaces,
v2.0.0)
├── tsconfig.json                   (Shared TypeScript strict configuration)
├── bun.lock                        (Dependency lockfile)
│
├── cli/                            [Terminal UI application]
│   └── src/
│       ├── index.ts                Entry point
│       ├── utils/
│       │   ├── config.ts           SantraConfig v2, model pricing, cost
calc
│       │   ├── diff.ts             Myers diff algorithm implementation
│       │   ├── run-state-storage.ts Session persistence (save/load/list)
│       │   ├── auth.ts             OAuth + API key authentication
│       │   ├── santra-home.ts      ~/.config/santra/ directory management
│       │   └── shell-profile.ts    Shell profile auto-setup on install
│       └── tui_v4/
│           ├── App.tsx             Root Ink component (957 lines)
│           ├── types.ts            TUI-specific type definitions
│           ├── input.ts            Slash commands, input sanitisation
│           ├── components/
│           │   ├── Composer.tsx    Input area with approval/question
overlays
│           │   ├── TranscriptView.tsx  Scrollable log viewer
│           │   ├── ChromeBar.tsx   Top status bar
│           │   ├── WelcomeState.tsx Empty state with ASCII art
│           │   ├── SetupFlow.tsx   First-run provider wizard
│           │   ├── ModelSelector.tsx Model switcher overlay
│           │   └── LoginGate.tsx   OAuth login screen
│           ├── hooks/
```

```
│           │   └── useAgent.ts      Central state hook (1385 lines)
│           └── run-events/
│               └── LogDeriver.ts   O(1) event-to-LogEntry conversion
│
├── core/                            [Prompt routing layer]
│   └── src/
│       ├── runner.ts                Runner class -- classify and route
│       ├── classifier.ts            Zero-latency prompt classifier
│       └── types.ts                 RunnerOptions, RunOptions
│
├── packages/
│   ├── agent-runtime/               [LLM communication + tools engine]
│   │   ├── base-agent.ts            BaseAgent class (~1000 lines)
│   │   ├── swarm.ts                 Multi-agent Swarm orchestration
│   │   ├── tools/
│   │   │   ├── parser.ts            StreamParser for SSE
│   │   │   └── local-runner.ts      File operation tool execution
│   │   └── agents/
│   │       ├── index.ts             BUILTIN_AGENT_TEMPLATES definition
│   │       ├── orchestrator.ts      Orchestrator system prompt
│   │       ├── prompts.ts           Shared prompt blocks (THINKING, STATUS,
NEXT)
│   │       ├── executor.ts          Executor agent
│   │       ├── file-picker.ts       File Picker agent
│   │       ├── reader.ts            Reader agent
│   │       ├── reviewer.ts          Reviewer agent
│   │       └── thinker.ts           Thinker agent
│   │
│   └── shared/                      [Shared types and schemas]
│       ├── tools/definitions.ts     All 23 tool JSON schemas
│       └── types/index.ts           Message, RunState, AgentPhase, etc.
│
└── web/                             [Next.js website + API proxy]
    └── app/
        ├── page.tsx                 Landing page
        ├── docs/page.tsx            Documentation (716 lines)
        ├── install/page.tsx         Installation guide
        ├── providers/page.tsx       Provider setup guides
```

```
├── api/v1/completions/
│   └── route.ts              Unified LLM proxy endpoint
├── lib/content.ts            Shared constants
└── public/santra-icon.svg    Project logo
```

APPENDIX B: GLOSSARY OF TERMS

Term	Definition
Agent	An autonomous AI process with a specific role, system prompt, and tool whitelist. Agents operate within the Swarm, receiving tasks and returning results.
AgentPhase	A typed event emitted by the agent runtime during a run. Types include: status, text, thinking, tool_call, tool_result, section, error, complete.
AgentTemplate	A definition object for a built-in or custom agent, containing id, name, description, systemPrompt, and toolNames.
Approval Gate	A mandatory user confirmation step that pauses file write operations and displays a diff preview. Users choose to allow, reject, or provide feedback.
BaseAgent	The core class in agent-runtime that handles a single LLM conversation turn: streaming, tool execution loop, and result collection.
BYOK	Bring Your Own Key — the model where users supply their own LLM API keys. Santra supports BYOK for all providers.
Bun	A fast JavaScript runtime, package manager, bundler, and test runner. Santra uses Bun 1.3.11 for its monorepo.
Classifier	The zero-latency prompt classifier in core/src/classifier.ts that categorizes prompts as simple_chat, direct_answer, or agent_task.
Composer	The bottom input area of the TUI. Shows slash command suggestions, file approval overlays, and user question overlays.
DiffEntry	Data structure representing a file change: file path, added/removed line counts, and DiffLine array.

DiffLine	A single line in a diff: type (add/remove/context), line number, and content string.
Ink	A React-based library for building terminal UIs. Uses Yoga's Flexbox layout engine to render components as terminal cells.
LogDeriver	The O(1) incremental event processor that converts AgentPhase events to LogEntry objects for TUI display.
LogEntry	A display record for the TranscriptView. Contains id, time, level, message, and optional diff/title/detail.
LLM	Large Language Model — the AI model that generates responses, uses tools, and drives agent behavior.
Myers Diff	An efficient diff algorithm by Eugene Myers (1986) that computes the minimum edit distance between two sequences in O(ND) time.
Orchestrator	The master agent that receives the user's task, decomposes it, and delegates subtasks to specialist agents via spawn_agent/spawn_agents.
Provider	An LLM API service (Anthropic, OpenAI, Groq, etc.) that Santra can route requests to.
RunState	The serializable state of a completed run: messages, output, toolCalls, thinking. Persisted to run-state.json.
Runner	The core/src/runner.ts class that classifies prompts and routes them to BaseAgent or Swarm.
SSE	Server-Sent Events — a unidirectional HTTP streaming protocol. Santra uses SSE for real-time LLM token delivery.
Swarm	The multi-agent orchestration system that manages spawning, execution, and result collection of multiple agents.
StreamParser	The stateful parser in tools/parser.ts that processes streaming text chunks and extracts structured elements (text, thinking, tool calls).
ToolCallRequest	The normalized representation of a tool invocation from the LLM: id, name, input parameters.

ToolCallResult	The result of executing a tool: name, callId, output string, optional error.
TranscriptView	The scrollable log viewer component that renders LogEntry rows with viewport clipping.
TUI	Terminal User Interface — a text-based UI rendered in a terminal emulator. Santra's TUI uses Ink/React.
useAgent	The central React hook in cli/src/tui_v4/hooks/useAgent.ts that manages all agent interaction state and lifecycle.
WebStreamEvent	The normalized SSE event format shared between Santra's web API and client: types include start, delta, text, reasoning, finish, error.
Yoga	Meta's cross-platform Flexbox layout engine used by Ink to position terminal UI components.

APPENDIX C: ENVIRONMENT SETUP GUIDE

C.1 Prerequisites

Requirement	How to obtain
Bun ≥ 1.3.11	curl -fsSL https://bun.sh/install | bash
Node.js ≥ 18	https://nodejs.org
Git	Pre-installed on most systems
API Key	From Anthropic, OpenAI, or any compatible provider

C.2 Build from Source

```
git clone https://github.com/vishalvoid/santra-cli
```

```
cd santra-cli
```

```
bun install                    # install all workspace dependencies
```

```
bun run build                  # production build of all packages
```

```
bun run dev                    # run CLI in watch mode (development)
```

```
cd web && bun run dev          # run Next.js dev server
```

```
bun test                        # run test suite
```

```
bun run typecheck               # TypeScript type checking
```

C.3 Install from npm

```
npm install -g santra-cli
# then run:
santra
```

C.4 Provider Configuration

```
# Anthropic (recommended)
export ANTHROPIC_API_KEY=sk-ant-api03-...

# OpenAI
export OPENAI_API_KEY=sk-...

# Ollama (local, free)
export OPENAI_API_KEY=ollama
export OPENAI_BASE_URL=http://localhost:11434/v1
```