ChatGPT

Autonomous Coding Agent – Technical Product Requirements

Introduction

- **Vision:** Develop an *autonomous coding agent* that assists developers in coding tasks efficiently. Instead of a simple chatbot, this agent should mimic the agentic capabilities of tools like Claude Code and Cursor by reading code, running commands, making changes, and verifying results. It will function inside a desktop application rather than a terminal interface.

- **Goals:** Create a user-friendly **Electron/React** desktop application that delivers AI-powered coding assistance. The product should leverage advanced language models, semantic search and recursion to explore codebases, generate and refactor code, ask clarifying questions and validate its work autonomously. Ralph Wiggum serves as the playful mascot. The system must also **minimize token consumption** by employing compressed communication formats (TOON) and isolating context per agent.

- **Scope:** This document describes the high-level requirements, architecture, development phases and technology choices for the autonomous coding agent. It prioritizes *phasing and tooling* over specific timelines. All references are drawn from current documentation of coding agents (e.g., Claude Code and Cursor) to ground the requirements in proven concepts.

Functional Requirements

Core Features

1. **Intelligent Code Editing and Refactoring** – The agent must be able to read files, understand code semantics and make coordinated edits across multiple files to implement features or fix bugs. Claude Code demonstrates that an agentic assistant can "read your files, run commands, make changes, and autonomously work through problems" [1]; our agent must provide similar capabilities within a desktop UI.

2. **Context-Aware Code Search and Navigation** – Efficiently search files by pattern, explore codebases, and navigate between definitions using a semantic index. Claude's built-in tools include search that can "find files by pattern" [2], while Cursor uses a "custom embedding model" for codebase indexing [3]. The agent should integrate vector-based search for accurate recall across large repositories.

3. **Automated Planning, Coding and Verification Loops** – Adopt an agentic loop similar to Claude Code: gather context, take action, and verify results. Claude's workflow repeatedly reads errors, searches relevant files, edits code and re-runs tests [4]. The agent must implement planning

(exploration and plan creation), execution and verification, with options for the user to interrupt and steer the process.

4. **Multi-Agent/Subagent Support** – Enable the spawning of specialized sub-agents for tasks like security review or documentation. Claude Code allows subagents to work in separate contexts, preventing the main session from being overwhelmed [5]. Our agent should support parallel subagents to perform long-running research or testing tasks.

5. **Token-Efficient Communication (TOON Protocol)** – All communication between subagents and the orchestrator must occur using **TOON**, a compact representation designed to minimize token usage. Unlike Markdown or JSON, TOON encodes messages in a compressed, schema-based format. This reduces the overhead of transferring large context data and helps control costs. Agents should never transmit raw conversation history; instead they send succinct instructions or tool invocation commands encoded in TOON.

6. **Natural Language Interface** – Provide a chat interface where users describe tasks in plain English. The system asks clarifying questions and can "interview you" to uncover requirements [6], then produces a complete implementation plan.

7. **Memory and Context Management** – Persist learnings across sessions using summaries and embeddings. Claude Code uses session files and auto-memory to restore context [7], while Cursor indexes entire codebases. Our agent should summarize conversation history and maintain a vector memory to recall prior decisions while keeping the active context manageable.

The memory subsystem **must isolate context per agent**. Each subagent operates on its own context pool (held in a VM environment), and the orchestrator enforces that subagents **do not exchange context**. Only high-level instructions encoded in TOON are passed between agents; context data, code snippets or embeddings are never shared. This design minimizes token overhead and preserves privacy while still enabling collaborative workflows.

1. **Tool Execution and Integration** – Allow the agent to run shell commands, tests, linters and other tools. Claude's built-in tools support execution of shell commands and tests [4]; Cursor's interface includes an integrated terminal [8]. The agent should execute commands in a sandboxed environment and report outputs.

Non-Functional Requirements

- **Desktop UI:** A modern **Electron/React** interface replaces the command-line interface. This requirement prioritizes user experience with panels for file navigation, chat, subagent management and test results. Cursor is built on VS Code (Electron) and shows how a familiar UI enhances adoption [9].

- **Performance and Responsiveness:** The application must perform semantic search and code modifications quickly. Use efficient embeddings and caching to minimize latency.

- **Security and Privacy:** Run code edits and commands in a sandbox. Encrypt user data and memory. Require explicit permission for any network or file operations.

- **Extensibility:** Provide plugin APIs to add new tools, models or UI panels. Claude Code supports skills, subagents, hooks and plugins [10] ; our agent should allow similar extensibility.

- **Token Efficiency & Communication Protocol:** To minimize token usage and control costs, the agent ecosystem must use a custom **TOON** protocol for inter-agent communication. TOON is a binary or highly compressed format that encodes instructions and tool invocations with minimal overhead. Agents **must not** exchange Markdown or JSON, and they **must not** share conversation history or code context. Only high-level instructions or tool commands should be transmitted.

System Architecture

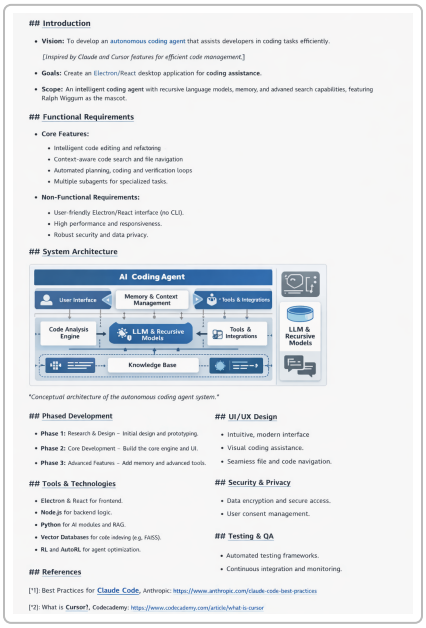

Figure: Conceptual architecture of the autonomous coding agent system.

1. **Electron/React User Interface** – The front-end consists of a React application packaged with Electron to provide a native-like desktop experience. Panels include a code editor, file explorer, chat panel, subagent viewer and results console.

2. **Agentic Orchestrator** – A back-end service orchestrates the agentic loop. It uses a large language model (LLM) to parse user requests, decompose tasks, choose tools and generate code. It controls subagents and tracks progress.

3. **LLM & Recursive Models** – Integration with one or more large language models (e.g., GPT-4, Claude, open-source LLMs) for reasoning. Recursive reasoning allows the agent to call itself to refine answers.

4. **Memory & Context Management** – A vector database (e.g., FAISS) stores embeddings of code files, conversation summaries and knowledge. Summarization ensures context windows remain usable by condensing prior conversation, similar to Claude's context management [11] .

5. **Code Analysis & Indexing Engine** – Indexes the entire codebase to allow semantic search, definition lookup and reference finding. Cursor uses a "custom embedding model" for best-in-class recall [3]; our engine should replicate this using embeddings and AST parsing.

6. **Tools & Integrations Layer** – Exposes operations like file read/write, regex search, command execution, test running and web search to the orchestrator. It also provides plugin hooks for adding new tools. Claude's built-in tool categories (file operations, search, execution and web [12]) inspire the design.

7. **Knowledge Base** – Stores domain documentation, user-provided instructions (equivalent to CLAUDE.md) and persistent rules. It is loaded into memory when needed but remains separate from active context.

8. **Context Isolation & VM Environments** – Each agent or subagent executes within its own lightweight virtual machine (VM) or sandboxed container. These environments hold their own **context pool** (code files, memory embeddings and conversation summaries) and are destroyed or reset after tasks complete. Agents **never share context between VMs**; they only exchange high-level instructions encoded in TOON. This strict isolation prevents cross-contamination of context and reduces token usage.

Phased Development

To structure development without strict timelines, work will proceed in **phases**, each delivering a usable increment.

1. **Phase 1 – Research & Design**
2. Study existing agentic coding environments (Claude Code and Cursor) to extract best practices. Identify core algorithms for file search, planning and verification.
3. Design the system architecture, define data schemas (e.g., for tasks, memory, code indices) and choose tech stacks.

4. Prototype UI layouts and gather feedback on ergonomics and workflows.

5. **Phase 2 – Core Engine & UI**

6. Implement the Electron/React desktop application with chat interface, code editor and file explorer.
7. Develop the agentic orchestrator capable of running the gather–plan–act–verify loop. Integrate a base LLM and implement tool wrappers for file reading/writing, semantic search and command execution.

8. Build the code indexing engine using embeddings and AST parsing. Add simple memory management and session persistence.

9. **Phase 3 – Memory & Advanced Tools**

10. Expand memory management with vector storage, auto-summarization and persistent instructions. Implement subagent support to run specialized tasks in separate contexts [5].

11. Introduce reinforcement learning (RL) and auto-RL frameworks to optimize agent behaviour over time.

12. Develop plugin APIs for custom tools, integrate external services (e.g., CI/CD, GitHub) and implement robust security (sandboxing, permission prompts).

13. **Phase 4 – Polishing & Scaling**

14. Optimize performance across large repositories and long sessions. Add advanced UI features such as visual diffing, inline annotations and multi-pane layouts.
15. Conduct user testing, gather feedback and iterate on ergonomics. Prepare documentation, onboarding tutorials and integrate the Ralph Wiggum mascot into the UI for branding.

Tools & Technologies

- **Electron & React** – Provide the cross-platform desktop front-end with a familiar coding environment.
- **Node.js** – Runs the agentic orchestrator and coordinates between the UI and Python modules.
- **Python** – Hosts AI modules, such as model inference, recursive reasoning, semantic search and RL algorithms.
- **Vector Databases (e.g., FAISS)** – Store embeddings for code files, conversations and memory.
- **Large Language Models** – Use state-of-the-art models (GPT-4, Claude, or open-source LLMs) for reasoning, code generation and chat.
- **RL and Auto-RL** – Optimize agent behaviour and tool selection using reinforcement learning frameworks.

UI/UX Design

- Provide an **intuitive, modern interface** with dark/light themes and accessibility options. Use a code editor component with syntax highlighting, diffing, and inline error annotations.
- Integrate **chat and plan panels** where users can describe tasks and view agent plans.
- Offer **visual coding assistance** – for example, highlight where code will be modified, preview changes before applying them, and allow step-wise execution or full automation.
- Implement **seamless file and code navigation**. Use fuzzy search and semantic indexing so that users can jump to definitions or search within code quickly, echoing Cursor's ability to understand entire codebases [9].

Security & Privacy

- **Sandboxed Execution:** All commands and code generated by the agent run in isolated environments to prevent unintended side effects.
- **Data Encryption:** User data, memory snapshots and embeddings are encrypted at rest and in transit.
- **User Consent:** The agent asks for explicit permission before network requests, file writes outside the project or integration with external services.

Testing & Quality Assurance

- **Automated Testing Framework:** Build unit, integration and end-to-end tests to verify agent actions. Provide test harnesses that the agent can run itself to validate changes, mirroring Claude's emphasis on verifying its work [13].
- **Continuous Integration:** Integrate with CI pipelines to ensure that code modifications pass tests and linters. Use RL to optimize patch acceptance rates.
- **Monitoring & Analytics:** Collect metrics on agent performance, latency and memory usage. Use feedback to improve models and heuristics.

References

1. **Best Practices for Claude Code** – Anthropic's official documentation notes that Claude Code is an agentic environment that can read files, run commands and make changes autonomously [1]. It highlights context-aware workflows, subagents and the importance of verifying work [5].
2. **How Claude Code Works** – The guide explains the agentic loop (gather context, take action, verify results) and lists built-in tools such as file operations, search and execution [4]. It also details context window management and auto-compaction [11].
3. **Cursor AI Guide (Codecademy)** – Cursor is a fork of VS Code augmented with large language models that can understand entire codebases, generate functions from plain English, explain complex code and debug errors across multiple languages [9]. The platform uses embedding-based indexing for fast recall and integrates a familiar UI.

---

[1] [5] [6] [10] [13] Best Practices for Claude Code - Claude Code Docs
https://code.claude.com/docs/en/best-practices

[2] [4] [7] [11] [12] How Claude Code works - Claude Code Docs
https://code.claude.com/docs/en/how-claude-code-works

[3] [8] Cursor · Agent
https://cursor.com/product

[9] How To Use Cursor AI: A Complete Guide With Practical Example | Codecademy
https://www.codecademy.com/article/how-to-use-cursor-ai-a-complete-guide-with-practical-examples