
KS Assistant: A Simple General-Purpose AI Agent for Software Engineering

Anonymous Author(s)

Affiliation

Address

email

Abstract

Large language models can generate code and call tools with remarkable fluency, yet deploying them as practical software engineering assistants still exposes stubborn gaps: finite context windows, a single mistake that can derail entire sessions, agents that get stuck in dead ends, AI slop, and generated changes that are difficult to review or revert.

We present **KS Assistant**, a general-purpose assistant and integrated development environment (IDE) built on top of the **KS Agent Framework**, a simple AI agent framework of roughly 1,900 lines of code. The framework addresses these gaps through a robust system prompt and a five-layer agent hierarchy in which each layer adds exactly one concern: budget-tracked ReAct execution, automatic continuation across sub-sessions via summarization, coding and browser tools with parallel sub-agents, persistent multi-turn chat with history recall, and git worktree isolation so every task runs on its own branch. Both KS Assistant and the KS Agent Framework are grounded in disciplined software engineering practice; these principles are encoded directly into the agent’s system prompt, enabling KS Assistant to write code that is simple, elegant, maintainable, and bug-free.

We implemented KS Assistant as a free, open-source Visual Studio Code extension that runs locally, is effective for long-horizon tasks, and supports browser automation, multimodal input, and Docker containers. We deliberately prioritize output quality over speed: giving a frontier model adequate time to validate its own output—running linters, type checkers, and tests—dramatically reduces the low-quality code that plagues faster but less thorough agents. The entire system was built using itself over four months, providing a continuous stress test in which any bug was patched immediately after its manifestation. On Terminal-Bench 2.0, KS Assistant achieves a 62.2% overall pass rate with Claude Opus 4.6 [Anthropic, 2026a], comparing favorably to Claude Code (58%) and Cursor Composer 2 (61.7%). These results are achieved without benchmark-specific prompt tuning or model fine-tuning.

1 Introduction

Modern large language models (LLMs), such as Claude Opus 4.7 [Anthropic, 2026b], GPT 5.5 [OpenAI, 2026], and Gemini 3.1 [Google DeepMind, 2026], have demonstrated a remarkable ability to generate code, reason about software architecture, and use developer tools [Chen et al., 2021, Rozière et al., 2023]. A growing body of work has explored how to harness these capabilities for autonomous software engineering, from single-session agents that resolve GitHub issues [Yang et al., 2024, Wang et al., 2024] to industrial products marketed as AI software and general assistants [GitHub, 2021, Cursor, 2024, Cognition Labs, 2024, Anthropic, 2025, OpenAI, 2025a, OpenClaw AI, 2025]. Yet using an LLM as a practical software engineering assistant still exposes several stubborn gaps:

context windows are finite, a single mistake can derail an entire session, agents get stuck in dead ends, models generate AI slop, and generated changes are difficult to review or revert once applied to a live codebase.

We propose the *KS Agent Framework*, a simple AI agent framework containing around 1,900 lines of code for the core agent implementation. The name “KS” reflects the *Keep Simple* design philosophy: each layer is small, each concern is isolated, and the overall system avoids unnecessary abstraction. We address the above-mentioned gaps through a robust system prompt (Section 4 and Appendix A) and a five-layer agent hierarchy in which each layer solves exactly one concern:

1. **KS Agent** — budget-tracked ReAct [Yao et al., 2023b] loop with native function calling.
2. **Relentless Agent** — automatic summarization and continuation across sub-sessions.
3. **Tool Agent** — coding tools, browser automation, and parallel sub-agent execution.
4. **Chat Agent** — persistent multi-turn chat sessions with history recall.
5. **Worktree Agent** — git worktree isolation so every task runs on its own branch.

To demonstrate the power of the KS Agent Framework, we implemented KS Assistant as a Visual Studio Code extension that runs locally. It has full browser support (using open-source Chromium and Playwright), multimodal support, Docker container support, and a mobile/web app. KS Assistant is completely free and open-source; all one needs is a model API key from a major LLM provider. The open-source repository link is withheld to respect double-blind review.

KS Assistant has been built using itself. The entire codebase—the KS Agent framework, the agent layers, the VS Code extension, and the system prompt—was developed by KS Assistant operating on its own repository. This self-hosting discipline provides a continuous-integration-style stress test: if the agent introduces a bug that impairs its ability to function, the developers immediately ask the agent to fix it by analyzing the trajectory and code. The simplicity of the framework enabled the agent to implement features confidently without bugs and AI slop. The five core agent classes are remarkably compact: the KS Agent comprises 413 lines, the Relentless Agent 321 lines, the Tool Agent 322 lines, the Chat Agent 125 lines, and the Worktree Agent 704 lines—a total of roughly 1,885 lines of code (excluding empty lines and comments).

We deliberately prioritize output quality over speed. Using a weaker or cheaper model often forces the developer to discard the agent’s work and retry, ultimately increasing the total cost. Conversely, giving a frontier model adequate time to validate its own output—running linters, type checkers, and tests before declaring success—dramatically reduces slop. We expect token costs and inference latencies to continue to fall [Gao et al., 2025], making this quality-first posture increasingly practical.

We evaluate on Terminal-Bench 2.0 and achieve a 62.2% overall pass rate using Claude Opus 4.6 [Anthropic, 2026a], comparing favorably to Claude Code (58%) and Cursor Composer 2 (61.7%) [Cursor Research, 2026] on the same benchmark (Section 3). These results are achieved without benchmark-specific prompt tuning or model fine-tuning. We show that a simple agent framework, without sophisticated agent technologies such as trajectory compaction and asynchronous multi-agent orchestration, is sufficient to build a competitive software engineering assistant. By building KS Assistant using itself and matching or beating both the Cursor and Claude Code agents, we demonstrate that time-tested software engineering techniques and principles are invaluable for building reliable, practical agent systems.

Outline. Section 2 presents the five-layer agent architecture. Section 3 reports evaluation results on Terminal-Bench 2.0. Section 4 describes the two most distinctive aspects of the system prompt: the execution mindset and the web research protocol (the full prompt is detailed in Appendix A). Section 5 covers unique features of the VS Code extension. Section 6 discusses related work, and Section 7 concludes.

2 Agent Architecture

The KS Agent Framework was originally built to rapidly prototype and experiment with prompt optimization techniques [Agrawal et al., 2026] and evolutionary algorithms for algorithmic optimization [Novikov et al., 2025, Algorithmic Superintelligence, 2025]. The emphasis on simplicity enabled coding agents to write simple, bug-free code. No prompt optimization techniques were used when creating the system prompt for KS Assistant; it was hand-tuned based on long-term experience

90 with the agent and its behavior. The framework uses five agent layers combining composition and
91 inheritance. Each layer delegates upward for concerns it does not own.

92 2.1 KS Agent

93 The KS Agent is the innermost execution unit implementing a standard ReAct loop [Yao et al.,
94 2023b].

```
95 from ks.core.ks_agent import KSAgent
96
97
98 def calculate(expression: str) -> str:
99     """Evaluate a math expression."""
100     return str(eval(expression))
101
102 agent = KSAgent(name="Math Buddy")
103 result = agent.run(
104     model_name="gemini-2.5-flash",
105     prompt_template="Calculate: {question}",
106     arguments={"question": "What is 15% of 847?"},
107     tools=[calculate]
108 )
109 print(result) # 127.05
110
```

Listing 1: A complete KS agent with a single tool.

111 **Native function calling.** Tools are registered as ordinary Python callables. The agent builds an
112 OpenAI-compatible tool schema once at setup time and caches it. A special `finish` tool signals task
113 completion and returns the result.

114 **Step, token, and budget tracking.** At every step, the agent extracts input and output token counts
115 from the API response, computes dollar cost using a per-model pricing table, and updates both local
116 and global budget counters protected by a class-level lock. Three limits are checked before each step:
117 per-agent budget, global budget, and maximum step count.

118 **Error resilience.** The agent retries transient API errors (rate limits, server errors) up to a configurable
119 threshold. Non-retryable errors (authentication failures, permission denials) are raised immediately.

120 **Non-agentic mode.** When tools are not needed, the agent runs a single generation without the ReAct
121 loop, useful for summarization sub-tasks.

122 Listing 1 shows a complete, working agent in under ten lines of code. The developer defines an
123 ordinary Python function (`calculate`), instantiates a `KSAgent`, and calls its `run` method with a
124 model name, a prompt template, template arguments, and a list of tools. The framework automatically
125 handles tool-schema generation, the ReAct loop, and budget tracking.

126 The KS Agent is stateless across runs: each call resets the conversation, token counters, and tool
127 registry, making it safe to reuse a single instance for multiple sequential tasks.

128 2.2 Relentless Agent

129 The Relentless Agent wraps a KS Agent in a continuation loop. Its core contribution is executing
130 tasks that exceed a single context window by breaking them into sub-sessions.

131 Rather than investing in context-compaction techniques, we adopt a simple continuation protocol:
132 when a sub-session exhausts its context window or step budget, the agent produces a *structured*
133 *summary* of every action taken so far—chronologically ordered, with explanations and relevant
134 code snippets—and a fresh sub-session resumes from that summary. This approach is related to
135 Reflexion [Shinn et al., 2023], which feeds verbal self-critiques back into subsequent trials, but
136 uses a Reflexion-like technique to *continue* a task rather than retry it. We found that a naïve
137 instruction to “summarize the current context” produced poor continuations; requiring a *step-by-step*
138 *chronological account with code snippets* dramatically improved coherence across sub-sessions. A
139 potential limitation is that summaries may grow unwieldy for multi-day tasks; in practice, we have
140 not encountered this problem even for tasks spanning several hours, but a thorough evaluation of
141 summary scaling remains future work.

142 **Continuation protocol.** The `finish` tool accepts three fields: a success flag, a continue flag, and
143 a summary. When `is_continue=True`, the Relentless Agent starts a new sub-session with a fresh

144 context window. The prompt for the new session includes a chronologically ordered list of all prior
145 attempt summaries and instructs the agent not to redo completed work:

```
146 # Task Progress (Continuation {continuation_number})
147
148 {progress_text}
149
150 # Continue
151 - Complete the rest of the task.
152 - **DON'T** redo completed work.
153 - If you have been retrying the same approach without
154 progress, step back and rethink the strategy from
155 scratch.
```

158 **Forced continuation on failure.** If a sub-session raises an exception (e.g., the step limit is hit before
159 calling finish), the Relentless Agent saves the full trajectory to a temporary file, spawns a separate
160 summarizer agent to produce a concise summary, and uses that summary as the progress text for the
161 next sub-session. This ensures that even crashed sessions contribute useful context. The summarizer
162 receives the following prompt:

```
163 # Summarizer
164
165 The trajectory of the agent is stored in the file:
166 {trajectory_file}
167
168 # Instructions
169 - Read the trajectory file and analyze it. The trajectory
170 file could be large.
171 - Return a precise chronologically-ordered list of things
172 the agent did, with the reason for doing that, along
173 with relevant code snippets
```

176 **Pre-emptive continuation.** To force the agent to self-continue before exhausting its step budget, the
177 system prompt is augmented with a step-threshold instruction:

```
178 # MOST IMPORTANT INSTRUCTIONS
179 - **At step {step_threshold}: you MUST call
180 finish(success=False, is_continue=True,
181 summary="precise chronologically-ordered list of things
182 the agent did with the reason for doing that along with
183 relevant code snippets")** or if the task is not complete
184 and you are at risk of running out of steps or context
185 length.
```

188 This instruction is injected near the end of the budget window. Without it, the agent exhibits a
189 “rush to finish” behavior when steps are running low—skipping verification, making hasty edits, and
190 calling finish with an incomplete result. The explicit step threshold redirects that urgency into the
191 continuation protocol, ensuring that a clean handoff to a new sub-session produces better results than
192 a frantic attempt to squeeze everything into the remaining steps.

193 2.3 Tool Agent

194 The Tool Agent adds the tools that make the system useful for software development and general-
195 purpose automation.

196 **Coding tools.** Four core tools: a shell command executor with streaming output, a file reader, a
197 precise string-based file editor, and a file writer. The shell executor supports configurable timeouts,
198 streams output in real time, and respects a stop event for user cancellation.

199 **Browser automation.** A web-use tool provides programmatic browser control: navigating to URLs,
200 reading page accessibility trees, clicking elements, typing text, pressing keys, scrolling, and taking
201 screenshots. It uses the open-source Chromium browser via the Playwright library.

202 **Parallel sub-agents.** An optional parallel execution tool spawns independent Tool Agent instances
203 in a thread pool. Each sub-agent gets its own LLM context and tool set, useful for embarrassingly
204 parallel tasks such as summarizing multiple files or researching independent topics. Results are

collected and returned in input order. This tool is not activated by default because the IDE cannot stream multiple agent outputs coherently in the chat window.

User interaction. An ask-user-question tool pauses execution and requests clarification from the user.

Docker isolation. When a Docker image is specified, coding tools are replaced with Docker-aware variants that execute commands inside a container.

2.4 Chat Agent

The Chat Agent adds multi-turn conversation persistence.

Chat sessions. Each task is assigned to a chat session identified by a stable chat ID. Tasks and results are persisted to a local database. New tasks within the same session include prior tasks and results as numbered context entries, allowing the LLM to reference earlier work.

Bounded chat context. The agent caps in-context history entries at $K = 10$. When the cap is exceeded, it preserves the first two entries (which establish the user’s intent) and the most recent entries, dropping the middle entries least likely to be referenced.

Session management. Three operations are supported: starting a new chat, resuming by task description, and resuming by explicit chat ID.

Frequent task tracking. Each time a task is executed, the agent records the task description in a frequency table. The IDE sidebar surfaces the most frequent tasks so users can re-issue them with one click, turning recurring requests (“run the test suite,” “regenerate the changelog”) into a click-to-replay experience.

Metadata persistence. After each task, the agent records metadata including the model used, working directory, software version, token counts, cost, and whether the task used parallel execution or worktree isolation. This audit trail supports cost analysis and debugging.

2.5 Worktree Agent

The Worktree Agent is the outermost layer. Its defining feature is git-worktree isolation.

Branch-per-task. When a task starts, the agent creates a new git branch and corresponding worktree directory. All agent modifications happen inside the worktree; the user’s main working tree remains untouched.

Dirty-state preservation. Uncommitted changes in the main working tree are copied into the worktree with a baseline commit. During merge, cherry-pick from the baseline replays only the agent’s changes.

Concurrency safety. A per-repository file lock serializes checkout, stash, merge, and pop operations. Thread-local storage isolates per-task state.

Crash recovery. All worktree state is stored in git itself (branch names, git config entries) rather than sidecar files. On process restart, the agent queries git and reconstructs instance attributes, enabling seamless recovery.

Graceful fallback. If the working directory is not inside a git repository, has no commits, or has a detached HEAD, the agent falls back to direct execution without worktree isolation.

3 Evaluation on Terminal-Bench 2.0

We evaluate on Terminal-Bench 2.0,¹ a benchmark comprising 89 diverse terminal-based programming tasks, ranging from building legacy compilers and configuring servers to solving cryptanalysis challenges and training machine-learning models. Each task runs in an isolated Docker container; a separate verifier automatically judges the result. We use the Harbor² framework to orchestrate execution and Claude Opus 4.6 [Anthropic, 2026a] as the underlying LLM. We do not modify the

¹<https://www.tbench.ai/>

²<https://github.com/harbor-framework/harbor>

249 general system prompt or inject benchmark-specific instructions. Evaluation was carried out on a
 250 2025 MacBook Air 15” with an M4 processor and 24 GB RAM.

251 3.1 Setup

252 We run 5 independent trials per task. The agent is a thin Harbor adapter that installs and invokes the
 253 KS Assistant CLI inside each container. We hard-skip 9 tasks verified to be infeasible for Opus 4.6
 254 across 6+ prior attempts (e.g., CompCert compilation, Windows 3.11 GUI installation, video OCR)
 255 to save time and token cost. Skipped tasks still count as failures.

256 3.2 Aggregate Results

257 Table 1 summarizes the aggregate statistics.

Table 1: Terminal-Bench 2.0 aggregate results (89 tasks, 5 trials each, Claude Opus 4.6).

Metric	Value
Total tasks	89
Overall pass rate	62.2% (277/445)
pass@any (at least 1/5 passes)	78.7% (70/89)
pass@all (all 5 pass)	43.8% (39/89)
Always-fail tasks	19
Always-pass tasks	39
Mixed-result tasks	31
Median cost per trial	\$0.45
Mean cost per trial	\$0.90
Median duration per trial	202 s
Mean duration per trial	446 s

258 The 62.2% overall pass rate compares favorably to other agents using the same underlying model:
 259 at the time of writing, Claude Code (also Opus 4.6) scores approximately 58% on the Terminal-
 260 Bench 2.0 leaderboard, and Cursor’s Composer 2—a custom fine-tuned model trained with large-scale
 261 reinforcement learning [Cursor Research, 2026]—achieves 61.7%. Our result suggests that the layered
 262 architecture and the structured system prompt contribute meaningfully beyond what the base model
 263 provides.

264 3.3 Task-Level Breakdown

265 **Consistently solved tasks (39 of 89).** These include cryptanalysis (FEAL differential), game-playing
 266 (chess best move), git operations (leak recovery), server configuration (gRPC key-value store, PyPI
 267 server, NGINX logging), data processing (resharding), formal verification (Coq plus_comm), ML
 268 inference (HuggingFace model serving, LLM batching scheduler), and system emulation (QEMU
 269 startup). The breadth of this set demonstrates that the agent’s generality is not limited to a single
 270 domain.

271 **Consistently failed tasks (19 of 89).** The failures cluster into three categories: (1) tasks requiring
 272 graphical or multimedia capabilities unavailable in the container (video processing, Windows 3.11
 273 GUI installation, MTEB leaderboard scraping), (2) tasks demanding very long or resource-intensive
 274 builds that exceed time or memory limits (CompCert compilation, Doom for MIPS, Caffe CIFAR-
 275 10, training fastText on Yelp data), and (3) tasks with niche domain-specific requirements that the
 276 model struggles to satisfy (DNA insertion, OCaml GC patching, polyglot C/Python binaries, protein
 277 assembly, cell segmentation).

278 **Mixed-result tasks (31 of 89).** Tasks such as write-compressor (3/5), crack-7z-hash (4/5),
 279 and feal-linear-cryptanalysis (4/5) succeed in most trials but occasionally fail due to non-
 280 determinism in the model’s reasoning or timing-sensitive environment interactions. Conversely,
 281 cancel-async-tasks (1/5) and dna-assembly (1/5) succeed rarely, suggesting they are at the
 282 boundary of the model’s capability.

283 **Leaderboard context.** KS Assistant does not score as high as some coding agents on the Terminal-
284 Bench 2.0 leaderboard. However, we note that recent analysis has found widespread cheating
285 on popular agent benchmarks, including Terminal-Bench 2.0: the top three submissions commit
286 harness-level cheating (e.g., leaking verifier code or answer keys into the agent’s environment), and
287 task-level cheating (e.g., Googling answers, mining git history, hardcoding test outputs) affects 28+
288 submissions across 9 benchmarks [Stein et al., 2026a,b]. Separately, an automated benchmark audit
289 found 45 confirmed hacking solutions across 13 widely used benchmarks exhibiting process-isolation
290 failures, answer leakage, and weak test assertions that allow perfect scores without solving a single
291 problem [Wang et al., 2026].

292 4 The System Prompt

293 The system prompt is a structured document that governs the agent’s behavior across all tasks. It is
294 not a generic instruction to “be helpful” but rather a precise specification of engineering discipline.
295 We present the two most distinctive aspects here; the complete prompt is detailed in Appendix A.

296 4.1 Execution Mindset

297 The prompt opens with two directives that set the tone for the entire session:

298 # FOCUS ON THE GIVEN TASK. ITS COMPLETION IS YOUR SOLE GOAL.
299 # BE RELENTLESS. BE CALM. BE RIGOROUS. BE ACCURATE. CHECK FACTS. NO AI SLOP.
300

302 Each directive addresses a specific failure mode observed in LLM-based agents:

303 **“FOCUS ON THE GIVEN TASK. ITS COMPLETION IS YOUR SOLE GOAL.”** LLMs have a
304 tendency to drift: they comment on the difficulty of a problem, explore tangential concerns, or ask
305 clarifying questions that the user has already answered. This directive anchors the model on the task
306 at hand and discourages meta-commentary that consumes tokens without making progress.

307 **“BE RELENTLESS.”** When an agent encounters an error—a failing test, a type violation, a command
308 that returns unexpected output—the default LLM behavior is to apologize, summarize the failure,
309 and ask the user what to do next. This directive instructs the model to treat errors as obstacles to
310 overcome, not as reasons to stop.

311 **“BE CALM.”** The complement of relentlessness. An overly aggressive agent may thrash between
312 approaches without deliberation. This directive encourages the model to analyze errors methodically
313 before reacting.

314 **“BE RIGOROUS.”** This instructs the model to follow the verification and testing disciplines encoded
315 later in the prompt, rather than taking shortcuts when a solution *looks* right.

316 **“BE ACCURATE.”** LLMs frequently generate plausible-but-wrong code, file paths, or command-line
317 flags. This directive raises the model’s threshold for asserting facts, encouraging it to verify claims
318 against the actual file system or documentation rather than relying on parametric memory.

319 **“CHECK FACTS.”** A more specific version of “be accurate,” targeting information the agent collects
320 via web tools.

321 **“NO AI SLOP”** “AI slop” refers to the low-quality, generic, hedging text that LLMs produce when
322 they lack confidence: filler phrases like “certainly!”, unnecessary caveats, and boilerplate explanations.
323 This directive encourages the model to be concise and substantive.

324 These two sentences occupy only two lines, yet they establish a behavioral contract that permeates
325 every subsequent interaction. We observe empirically that removing any single directive degrades the
326 agent’s behavior along the corresponding dimension (e.g., removing “BE RELENTLESS” causes the
327 agent to give up after the first error more frequently).

328 4.2 Web Research Protocol

329 When the agent needs external knowledge, the prompt prescribes a structured research workflow
330 rather than allowing ad-hoc browsing:

```

331 ## Use web tools when you need to:
332
333
334 - When you need to collect knowledge from the internet,
335   visit **AT LEAST 30 WEBSITES** (use a counter to keep
336   track of the number of websites you visited) and collect
337   information necessary for the task without much thinking
338   in a new file PWD/tmp/information-{unique_id}.md. Then go
339   over the information in
340   PWD/tmp/information-{unique_id}.md and think deeply about
341   how to complete the task at hand.
342 - If you need to log in to a website while browsing for
343   information, you MUST ask the user to help you with the
344   login.

```

346 The rationale is a two-phase separation between *collection* and *synthesis*. LLMs tend to anchor on
347 the first few results they encounter, biasing their solutions toward a narrow slice of the design space.
348 By forcing the agent to accumulate a broad set of information into a file *before* reasoning about it, the
349 protocol counteracts anchoring bias and encourages the model to consider diverse approaches.

350 The “at least 30 websites” threshold is deliberately aggressive: it ensures the agent does not shortcut
351 the research phase after two or three hits, and the resulting information file serves as an auditable
352 artifact of what the agent considered. The explicit “use a counter” instruction addresses an empirically
353 observed failure mode in which the model claims to have “visited many sites” after only a handful of
354 fetches; a concrete counter forces the model to verify the actual number visited before declaring the
355 collection phase complete.

356 This two-phase collect-then-synthesize discipline is also applied to local file browsing (Appendix A.9):
357 the agent writes a structured summary of each file’s relevant information into a temporary markdown
358 file, externalizing its understanding into a compact artifact that persists across context boundaries.
359 The instruction to collect “without much thinking” is deliberate: during the collection phase, the
360 agent extracts and records facts rather than analyzes. Analysis happens in the second phase, when the
361 agent reads its own summary and reasons about the collected information as a whole.

362 5 VS Code Extension: Unique Features

363 We release our system as a VS Code extension and a web app. While the underlying agent architecture
364 already differs from existing AI coding assistants, the extension introduces several features that,
365 to our knowledge, are not present in competing assistants—including GitHub Copilot [GitHub,
366 2021], Cursor [Cursor, 2024], Windsurf [Codeium, 2024], Devin [Cognition Labs, 2024], and
367 Aider [Gauthier, 2023].

368 **Cross-session self-improvement.** The extension maintains a `USER_PREFS.md` file that the agent
369 reads at the start of each task and *updates* at the end. When the agent discovers a new preference or
370 project invariant during task execution, it writes it to the file. The agent also resolves conflicts: if a
371 new preference contradicts an existing one, the old entry is removed. Over time, the file accumulates
372 a curated set of project-specific knowledge, making the agent progressively more effective without
373 any manual configuration. This mechanism is detailed in Appendix A.7.

374 **Real-time budget accountability.** The extension displays real-time cost tracking in the sidebar:
375 input tokens, output tokens, cache hits, dollar cost, and elapsed time are updated at every agent step.
376 Both per-task and global budget ceilings are enforced.

377 **Integrated browser automation.** The extension includes a browser automation tool that allows the
378 agent to navigate URLs, read accessibility trees, click elements, type text, press keys, scroll, and take
379 screenshots—all controlled programmatically from within a VS Code task.

380 6 Related Work

381 **Code-specialized language models.** Code Llama [Rozière et al., 2023] fine-tunes Llama 2 for code
382 generation. StarCoder [Li et al., 2023] trains on permissively licensed code. DeepSeek-Coder [Guo
383 et al., 2024] trains on a 2-trillion-token corpus. Frontier models such as Claude Opus 4.7 [Anthropic,
384 2026b], GPT 5.5 [OpenAI, 2026], Kimi K2.5 [Kimi Team, 2026], and GLM-5.1 [Z.ai, 2026] target

385 agentic software engineering. Our system is model-agnostic and benefits from advances in code-
386 specialized pre-training without architectural changes.

387 **Code generation agents.** SWE-Agent [Yang et al., 2024] and OpenHands [Wang et al., 2024] provide
388 LLM-based agents for resolving software engineering tasks. Agentless [Xia et al., 2024] shows that
389 a two-phase localize-then-repair pipeline can achieve competitive results. Devin [Cognition Labs,
390 2024], Claude Code [Anthropic, 2025], Codex [OpenAI, 2025a], and Aider [Gauthier, 2023] offer
391 various approaches to autonomous coding. Cursor released Composer 2 [Cursor Research, 2026],
392 a fine-tuned coding model trained with large-scale reinforcement learning. Our Relentless Agent
393 layer addresses the single-session limitation common to most of these systems, while our worktree
394 isolation provides stronger safety guarantees.

395 **ReAct, reasoning, and planning.** ReAct [Yao et al., 2023b] interleaves reasoning and action. Chain-
396 of-thought prompting [Wei et al., 2022] elicits step-by-step reasoning. Tree of Thoughts [Yao et al.,
397 2023a] generalizes to search over reasoning paths. Reflexion [Shinn et al., 2023] introduces verbal
398 reinforcement learning. Our continuation protocol is conceptually related to Reflexion, but uses
399 summaries to continue tasks rather than retry them.

400 **Multi-agent systems.** ChatDev [Qian et al., 2024] and MetaGPT [Hong et al., 2024] model software
401 development as conversations between role-playing agents. AutoGen [Wu et al., 2023] provides a
402 general multi-agent framework. We use a single agent with broad tool access and optional parallel
403 sub-agents, prioritizing practical utility over role-playing fidelity.

404 **Agent frameworks.** LangChain [LangChain, 2022], DSPy [Khattab et al., 2024], CrewAI [CrewAI,
405 Inc., 2024], smolagents [Roucher et al., 2025], the OpenAI Agents SDK [OpenAI, 2025b], and
406 Google’s ADK [Google, 2025] provide general-purpose agent infrastructure. Our architecture is
407 purpose-built for software engineering, with each layer addressing a specific practical concern.

408 **Benchmarks and evaluation.** SWE-bench [Jimenez et al., 2024], HumanEval [Chen et al., 2021],
409 LiveCodeBench [Jain et al., 2024], and SWE-bench Pro [Deng et al., 2025] evaluate coding agents at
410 various scales. Prathifkumar et al. [2025] raises concerns about benchmark contamination. We evalu-
411 ate on Terminal-Bench 2.0, which tests diverse terminal-based tasks in isolated Docker containers.

412 **Agentic software engineering.** Hassan et al. [2025] articulate the foundational pillars of agentic
413 software engineering. Li et al. [2025] surveys the landscape of autonomous coding agents. Wang et al.
414 [2025] surveys AI agentic programming techniques. Chatlatanagulchai et al. [2025] studies agent
415 context files. Mohsenimofidi et al. [2025] investigates context engineering for AI agents. Our system
416 prompt and per-repository override mechanisms are instances of the context-engineering patterns
417 these works advocate.

418 **Self-improvement and community prompts.** Robeyns et al. [2025] demonstrates a self-improving
419 coding agent. Our self-improvement loop captures a lighter-weight form of this idea via accumulated
420 preferences. Gao et al. [2025] applies test-time compute scaling to software engineering. Get Shit
421 Done (GSD) [TÂCHES, 2025] is a meta-prompting system for coding agents that addresses context
422 rot through phased planning documents. Parts of our system prompt were inspired by this work.

423 7 Conclusion

424 We have shown that a simple layered, single-concern architecture can address the practical challenges
425 of deploying LLM agents for real-world software development. On Terminal-Bench 2.0, a benchmark
426 of 89 diverse terminal-based tasks evaluated across 5 trials each, our system achieves a 62.2% overall
427 pass rate (277/445)—outperforming Claude Code (58%) and Cursor Composer 2 (61.7%) on the
428 same benchmark with the same underlying model (Claude Opus 4.6 [Anthropic, 2026a]). These
429 results are achieved without benchmark-specific optimizations, fine-tuning, or reinforcement learning:
430 the entire agent is roughly 1,900 lines of straightforward Python.

431 We complement the architecture with a system prompt that encodes engineering discipline directly
432 into the agent’s behavior: read before writing, test before fixing, plan before executing, verify before
433 finishing. These are not novel insights—they are the practices of careful software engineering,
434 translated into instructions that an LLM can follow. The evaluation suggests that giving a frontier
435 model the time and tools to validate its own output matters more than model-level customization.

436 **Limitations.** Our evaluation uses a single frontier model and benchmark; broader evaluation across
437 open-weight models and benchmarks such as SWE-bench Pro is future work. The continuation
438 protocol’s summaries may lose fidelity over many sub-sessions, and the quality-over-speed posture
439 may not suit latency-sensitive workflows.

440 References

- 441 Lakshya A. Agrawal, Shangyin Tan, Dilara Soylu, Noah Ziems, Rishi Khare, Krista Opsahl-Ong, Arnav Singhvi,
442 Herumb Shandilya, Michael J. Ryan, Meng Jiang, Christopher Potts, Koushik Sen, Alexandros G. Dimakis,
443 Ion Stoica, Dan Klein, Matei Zaharia, and Omar Khattab. GEPA: Reflective prompt evolution can outperform
444 reinforcement learning. In *International Conference on Learning Representations (ICLR)*, 2026. Oral. arXiv
445 preprint arXiv:2507.19457.
- 446 Algorithmic Superintelligence. OpenEvolve: Open-source implementation of AlphaEvolve. <https://github.com/algorithmicsuperintelligence/openevolve>, 2025.
- 448 Anthropic. Claude Code: Anthropic’s agentic coding system. <https://www.anthropic.com/product/claude-code>, 2025.
- 450 Anthropic. Introducing Claude Opus 4.6. <https://www.anthropic.com/news/claude-opus-4-6>, 2026a.
- 451 Anthropic. Introducing Claude Opus 4.7. <https://www.anthropic.com/news/claude-opus-4-7>, 2026b.
- 452 Worawalan Chatlatanagulchai, Hao Li, Yutaro Kashiwa, Brittany Reid, Kundjanasith Thonglek, Pattara Lee-
453 laprute, Arnon Rungsawang, Bundit Manaskasemsak, Bram Adams, Ahmed E. Hassan, and Hajimu Iida.
454 Agent READMEs: An empirical study of context files for agentic coding. *arXiv preprint arXiv:2511.12884*,
455 2025.
- 456 Mark Chen, Jerry Tworek, Heewoo Jun, Qiming Yuan, Henrique Pondé Pinto, Jared Kaplan, Harri Edwards,
457 Yuri Burda, Nicholas Joseph, Greg Brockman, et al. Evaluating large language models trained on code. *arXiv*
458 *preprint arXiv:2107.03374*, 2021.
- 459 Codeium. Windsurf: The AI-powered IDE. <https://windsurf.com>, 2024.
- 460 Cognition Labs. Devin: The first AI software engineer. <https://devin.ai>, 2024.
- 461 CrewAI, Inc. CrewAI: Framework for orchestrating role-playing, autonomous AI agents. <https://github.com/crewAIInc/crewAI>, 2024.
- 463 Cursor. Cursor: The AI-first code editor. <https://cursor.sh>, 2024.
- 464 Cursor Research. Composer 2 technical report. *arXiv preprint arXiv:2603.24477*, 2026.
- 465 Xiang Deng, Jeff Da, Edwin Pan, Yannis Yiming He, Charles Ide, Kanak Garg, Niklas Lauffer, Andrew Park,
466 Nitin Pasari, Chetan Rane, et al. SWE-Bench Pro: Can AI agents solve long-horizon software engineering
467 tasks? *arXiv preprint arXiv:2509.16941*, 2025.
- 468 Pengfei Gao, Zhao Tian, Xiangxin Meng, and Trae Research Team. Trae agent: An LLM-based agent for
469 software engineering with test-time scaling. *arXiv preprint arXiv:2507.23370*, 2025.
- 470 Paul Gauthier. Aider: AI pair programming in your terminal. <https://github.com/paul-gauthier/aider>,
471 2023.
- 472 GitHub. GitHub Copilot: Your AI pair programmer. <https://github.com/features/copilot>, 2021.
- 473 Google. Agent Development Kit (ADK): An open-source framework for building AI agents. <https://google.github.io/adk-docs/>, 2025.
- 475 Google DeepMind. Gemini 3.1 Pro. <https://deepmind.google/models/gemini/pro/>, 2026.
- 476 Daya Guo, Qihao Zhu, Dejian Yang, Zhenda Xie, Kai Dong, Wentao Zhang, Guanting Chen, Xiao Bi, Y. Wu,
477 Y. K. Li, Fuli Luo, Yun Xiong, and Wenfeng Liang. DeepSeek-Coder: When the large language model meets
478 programming — the rise of code intelligence. *arXiv preprint arXiv:2401.14196*, 2024.
- 479 Ahmed E. Hassan, Hao Li, Dayi Lin, Bram Adams, Tse-Hsun Chen, Yutaro Kashiwa, and Dong Qiu. Agentic
480 software engineering: Foundational pillars and a research roadmap. *arXiv preprint arXiv:2509.06216*, 2025.

481 Sirui Hong, Mingchen Zhuge, Jonathan Chen, Xiwu Zheng, Yuheng Cheng, Ceyao Zhang, Jinlin Wang, Zili
482 Wang, Steven Ka Shing Yau, Zijuan Lin, Liyang Zhou, Chenyu Ran, Lingfeng Xiao, Chenglin Wu, and Jürgen
483 Schmidhuber. MetaGPT: Meta programming for a multi-agent collaborative framework. In *International*
484 *Conference on Learning Representations (ICLR)*, 2024.

485 Naman Jain, King Han, Alex Gu, Wen-Ding Li, Fanjia Yan, Tianjun Zhang, Sida Wang, Armando Solar-Lezama,
486 Koushik Sen, and Ion Stoica. LiveCodeBench: Holistic and contamination free evaluation of large language
487 models for code. *arXiv preprint arXiv:2403.07974*, 2024.

488 Carlos E. Jimenez, John Yang, Alexander Wettig, Shunyu Yao, Kexin Pei, Ofir Press, and Karthik Narasimhan.
489 SWE-bench: Can language models resolve real-world GitHub issues? In *International Conference on*
490 *Learning Representations (ICLR)*, 2024.

491 Omar Khattab, Arnav Singhvi, Paridhi Maheshwari, Zhiyuan Zhang, Keshav Santhanam, Sri Vardhamanan,
492 Saiful Haq, Ashutosh Sharma, Thomas T. Joshi, Hanna Moazam, Heather Miller, Matei Zaharia, and
493 Christopher Potts. DSPy: Compiling declarative language model calls into self-improving pipelines. In *The*
494 *Twelfth International Conference on Learning Representations*, 2024.

495 Kimi Team. Kimi K2.5: Visual agentic intelligence. *arXiv preprint arXiv:2602.02276*, 2026.

496 LangChain. LangChain: Build context-aware reasoning applications. [https://github.com/langchain-ai/](https://github.com/langchain-ai/langchain)
497 [langchain](https://github.com/langchain-ai/langchain), 2022.

498 Hao Li, Haoxiang Zhang, and Ahmed E. Hassan. The rise of AI teammates in software engineering (SE 3.0):
499 How autonomous coding agents are reshaping software engineering. *arXiv preprint arXiv:2507.15003*, 2025.

500 Raymond Li, Loubna Ben Allal, Yangtian Zi, Niklas Muennighoff, Denis Kocetkov, Chenghao Mou, Marc
501 Marone, Christopher Akiki, Jia Li, Jenny Chim, et al. StarCoder: May the source be with you! *Transactions*
502 *on Machine Learning Research (TMLR)*, 2023.

503 Seyedmoein Mohsenimofidi, Matthias Galster, Christoph Treude, and Sebastian Baltes. Context engineering for
504 AI agents in open-source software. *arXiv preprint arXiv:2510.21413*, 2025.

505 Alexander Novikov, Ngân Vũ, Marvin Eisenberger, Emilien Dupont, Po-Sen Huang, Adam Zsolt Wagner, Sergey
506 Shirobokov, Borislav Kozlovskii, Francisco J. R. Ruiz, Abbas Mehrabian, M. Pawan Kumar, Abigail See,
507 Swarat Chaudhuri, George Holland, Alex Davies, Sebastian Nowozin, Pushmeet Kohli, and Matej Balog.
508 AlphaEvolve: A coding agent for scientific and algorithmic discovery. *arXiv preprint arXiv:2506.13131*,
509 2025.

510 OpenAI. Introducing Codex: A cloud-based software engineering agent. [https://openai.com/index/](https://openai.com/index/introducing-codex/)
511 [introducing-codex/](https://openai.com/index/introducing-codex/), 2025a.

512 OpenAI. OpenAI Agents SDK: A lightweight, powerful framework for multi-agent workflows. [https:](https://github.com/openai/openai-agents-python)
513 [//github.com/openai/openai-agents-python](https://github.com/openai/openai-agents-python), 2025b.

514 OpenAI. Introducing GPT-5.5. <https://openai.com/index/introducing-gpt-5-5/>, 2026.

515 OpenClaw AI. OpenClaw: Personal AI assistant. <https://openclaw.ai>, 2025.

516 Thanosan Prathifkumar, Noble Saji Mathews, and Meiyappan Nagappan. Does SWE-Bench-Verified test agent
517 ability or model memory? *arXiv preprint arXiv:2512.10218*, 2025.

518 Chen Qian, Wei Liu, Hongzhang Liu, Nuo Chen, Yufan Dang, Jiahao Li, Cheng Yang, Weize Chen, Yusheng
519 Su, Xin Cong, Juyuan Xu, Dahai Li, Zhiyuan Liu, and Maosong Sun. ChatDev: Communicative agents for
520 software development. In *Proceedings of the 62nd Annual Meeting of the Association for Computational*
521 *Linguistics (ACL)*, 2024.

522 Maxime Robeyns, Martin Szummer, and Laurence Aitchison. A self-improving coding agent. *arXiv preprint*
523 *arXiv:2504.15228*, 2025.

524 Aymeric Roucher, Albert Villanova del Moral, Thomas Wolf, Leandro von Werra, and Erik Kaunismäki. smola-
525 gents: A smol library to build great agentic systems. <https://github.com/huggingface/smolagents>,
526 2025.

527 Baptiste Rozière, Jonas Gehring, Fabian Gloeckle, Sten Sootla, Itai Gat, Xiaoqing Ellen Tan, Yossi Adi, Jingyu
528 Liu, Tal Remez, Jérémy Rapin, et al. Code Llama: Open foundation models for code. *arXiv preprint*
529 *arXiv:2308.12950*, 2023.

530 Noah Shinn, Federico Cassano, Ashwin Gopinath, Karthik Narasimhan, and Shunyu Yao. Reflexion: Language
531 agents with verbal reinforcement learning. In *Advances in Neural Information Processing Systems (NeurIPS)*,
532 2023.

533 Adam Stein, Davis Brown, Hamed Hassani, Mayur Naik, and Eric Wong. Finding widespread cheating on
534 popular agent benchmarks. Blog post, <https://debugml.github.io/cheating-agents/>, 2026a.

535 Adam Stein, Davis Brown, Hamed Hassani, Mayur Naik, and Eric Wong. Detecting safety violations across
536 many agent traces. *arXiv preprint arXiv:2604.11806*, 2026b.

537 TÂCHES. Get Shit Done: A light-weight meta-prompting, context engineering and spec-driven development
538 system for AI coding agents. <https://github.com/gsd-build/get-shit-done>, 2025. Initial commit
539 December 2025.

540 Hao Wang, Qiuyang Mang, Alvin Cheung, Koushik Sen, and Dawn Song. We scored 100% on AI bench-
541 marks without solving a single problem. Blog post, [https://moogician.github.io/blog/2026/](https://moogician.github.io/blog/2026/trustworthy-benchmarks/)
542 [trustworthy-benchmarks/](https://moogician.github.io/blog/2026/trustworthy-benchmarks/), 2026.

543 Huanting Wang, Jingzhi Gong, Huawei Zhang, Jie Xu, and Zheng Wang. AI agentic programming: A survey of
544 techniques, challenges, and opportunities. *arXiv preprint arXiv:2508.11126*, 2025.

545 Xingyao Wang, Yangyi Chen, Lifan Yuan, Yizhe Zhang, Yunzhi Li, Hao Peng, and Heng Ji. OpenHands: An
546 open platform for AI software developers as generalist agents. *arXiv preprint arXiv:2407.16741*, 2024.

547 Jason Wei, Xuezhi Wang, Dale Schuurmans, Maarten Bosma, Brian Ichter, Fei Xia, Ed Chi, Quoc Le, and
548 Denny Zhou. Chain-of-thought prompting elicits reasoning in large language models. In *Advances in Neural*
549 *Information Processing Systems (NeurIPS)*, 2022.

550 Qingyun Wu, Gagan Bansal, Jieyu Zhang, Yiran Wu, Beibin Li, Erkang Zhu, Li Jiang, Xiaoyun Zhang, Shaokun
551 Zhang, Jiale Liu, Ahmed Hassan Awadallah, Ryen W. White, Doug Burger, and Chi Wang. AutoGen:
552 Enabling next-gen LLM applications via multi-agent conversation. *arXiv preprint arXiv:2308.08155*, 2023.

553 Chunqiu Steven Xia, Yinlin Deng, Soren Dunn, and Lingming Zhang. Agentless: Demystifying LLM-based
554 software engineering agents. *arXiv preprint arXiv:2407.01489*, 2024.

555 John Yang, Carlos E. Jimenez, Alexander Wettig, Kilian Liber, Karthik Narasimhan, and Ofir Press. SWE-agent:
556 Agent-computer interfaces enable automated software engineering. *arXiv preprint arXiv:2405.15793*, 2024.

557 Shunyu Yao, Dian Yu, Jeffrey Zhao, Izhak Shafran, Thomas L. Griffiths, Yuan Cao, and Karthik Narasimhan.
558 Tree of thoughts: Deliberate problem solving with large language models. In *Advances in Neural Information*
559 *Processing Systems (NeurIPS)*, 2023a.

560 Shunyu Yao, Jeffrey Zhao, Dian Yu, Nan Du, Izhak Shafran, Karthik Narasimhan, and Yuan Cao. ReAct: Syner-
561 gizing reasoning and acting in language models. In *International Conference on Learning Representations*
562 *(ICLR)*, 2023b.

563 Z.ai. GLM-5.1: Towards long-horizon tasks. Technical blog, <https://z.ai/blog/glm-5.1>, 2026.

564 A Full System Prompt Details

565 This appendix presents the remaining sections of the system prompt not discussed in the main body.
566 The execution mindset and web research protocol are covered in Sections 4.1 and 4.2, respectively.

567 A.1 Tool Rules

568 Tool usage rules are explicit and mechanical:

```
569 # Rules
570
571 - PWD denotes your current working directory.
572 - Write() for new files. Edit() for small changes.
573 - Run Bash commands synchronously using the
574   'timeout_seconds' parameter. Use 300s (default) for
575   commands. If a command times out, retry with a higher
576   timeout. Only for commands expected to take more than
577   10 minutes, run them in the background, redirect output
578   to a file, and poll periodically.
579
```

```

580 - Use go_to_url() for browser tool.
581 - **The user cannot see intermediate chat. Show whatever
582   the user asks in the summary of the 'finish' tool
583   call.**
584 - READ large files in chunks.
585 - Create temporary files in PWD/tmp. Cleanup temporary
586   files after the task is done.
587 - Use ULTRA thinking ALWAYS.
588 - **If you are running out of context length or steps,
589   DO NOT rush to complete the task urgently, but continue
590   the task by calling 'finish' with 'is_continue' set
591   to true**
592

```

Each rule addresses a specific failure mode:

“Write() for new files. Edit() for small changes.” Without this distinction, the model may use Write() to overwrite an existing file with a slightly modified version, losing content it forgot to include.

Bash timeout guidance. LLMs frequently launch shell commands without considering their runtime. The instruction to use 300 seconds as the default, retry with higher timeouts, and run long-running commands in the background provides a mechanical protocol that handles common cases.

“The user cannot see intermediate chat.” This forces the model to put all user-facing information into the finish summary, preventing situations where the model “tells” the user something in an intermediate message and then assumes it was seen.

“READ large files in chunks.” Reading a 10,000-line file in a single tool call consumes a large fraction of the context window. Chunked reading preserves capacity for the rest of the task.

“Create temporary files in PWD/tmp.” Centralizing temporary files in a known directory makes cleanup predictable and prevents project-tree pollution.

“Use ULTRA thinking ALWAYS.” This activates the model’s extended reasoning mode, which is particularly valuable for complex, multi-step tasks.

“If you are running out of context length or steps...” When the context window is nearly full, LLMs exhibit a “rush to finish” behavior, skipping verification steps. This instruction redirects that urgency into the continuation protocol (Section 2.2).

A.2 Pre-flight Checks

Before modifying any file, the agent is instructed to read it first:

```

614 ## Pre-flight Checks
615
616 - Read every file you will modify before changing it.
617 - If the task depends on existing architecture or behavior,
618   read the relevant source files first.
619 - If the task references files, commands, or config that do
620   not exist, stop and ask or report instead of guessing.
621 - **When fixing a bug, an issue, or a race, write tests to
622   confirm them. Then fix them.**
623

```

“Read every file you will modify before changing it.” The most common source of agent-introduced bugs is modifying a file based on an incorrect assumption about its current contents. By requiring a fresh read immediately before any edit, the instruction ensures the model operates on the file’s current state.

“When fixing a bug... write tests to confirm them. Then fix them.” This mandates test-first discipline: a test that reproduces the bug provides concrete verification that the fix is correct. Writing the test forces the model to understand the bug precisely before attempting a fix.

A.3 Code Style Guidelines

The prompt encodes a minimalist code philosophy:

```

635 ## Code Style Guidelines
636
637 - Write simple, clean, and readable code with minimal
638   indirection.
639 - Organize the code in multiple files based on the code's
640   functionality.
641 - Avoid unnecessary object attributes, local variables, and
642   config variables.
643 - Avoid tight coupling among files and modules.
644 - Avoid object/struct attribute redirections.
645 - DO NOT USE CLOSURES.
646 - No redundant abstractions or duplicate code.
647 - Public methods MUST have full documentation.
648 - Understand the root cause of an issue or bug, and patch
649   the root cause instead of an ad hoc superficial fix.
650 - Before you write code, wait and think if the code is
651   simple, elegant, general, and minimal.
652 - Once you finish the task, DO NOT write documentation
653   unless the task specifically requires it.

```

Each guideline addresses an anti-pattern commonly exhibited by LLM-generated code. For example, “DO NOT USE CLOSURES” steers the model toward explicit data structures (plain functions with arguments, classes with attributes) that are easier to test and reason about. “Before you write code, wait and think” is a metacognitive instruction that asks the model to pause and evaluate its plan, spending more inference-time compute on design.

660 A.4 Deep Work Rules

```

661 ## Deep Work Rules
662
663
664 - When the task says "align", "match", or "make
665   consistent", read the target to determine the exact
666   target state before editing.
667 - Use concrete values, not indirections.
668 - For multi-part work, list the concrete planned changes
669   before executing them.
670 - Every meaningful change should have a concrete
671   verification method (test, grep, CLI command).

```

These rules address failures where the model interprets instructions loosely and makes changes that are directionally correct but concretely wrong.

675 A.5 Planning for Complex Tasks

```

676 ## Planning for Complex Tasks
677
678 For tasks involving 3+ files, cross-module changes, or
679 architectural work:
680
681 1. List the files that need to change and why.
682 1. State the exact intended change in each file.
683 1. Identify dependencies and execution order.
684 1. State how each change will be verified.
685
686 For simple single-file tasks, skip formal planning and
687 execute directly.

```

The complexity threshold—three or more files—avoids the overhead of planning trivial changes while ensuring comprehensive planning for tasks with cross-module dependencies.

692 A.6 Testing Instructions

```

693 ## Testing Instructions
694
695
696 - Run lint and typecheckers and fix any lint and typecheck
697   errors.
698 - You MUST achieve 100% branch coverage.
699 - Tests MUST NOT use mocks, patches, fakes, or any form of

```

```

700     test doubles.
701     - You MUST write integration tests or end-to-end tests.
702     - Each test should be independent and verify actual behavior.
703     - **Do NOT run all tests after modifications. Only run the
704       impacted tests.**
705     - To confirm a race condition, add sleep statements before
706       racing statements with delays less than 0.1s.

```

708 The most opinionated rule is the no-mocks requirement. Mock tests verify that code calls certain
 709 methods in a certain order—they test the implementation, not the behavior. A test suite built on
 710 mocks can pass with flying colors while the system is fundamentally broken. Integration tests that
 711 exercise actual behavior provide genuine confidence that the system works.

712 A.7 Self-Improvement Loop

713 The agent maintains a preferences file that captures user invariants discovered during task execution:

```

714 ## Self-Improvement Loop
715
716
717 - Read the instructions in PWD/USER_PREFS.md at the start
718   of each task.
719 - Then update PWD/USER_PREFS.md to capture the user
720   preferences and invariants by analyzing the task.
721 - You MUST carefully and thoroughly get rid of the user
722   preferences and invariants that conflict with the newly
723   added ones.

```

725 This mechanism allows the agent to accumulate project knowledge over time without requiring the
 726 user to repeat themselves. The conflict-resolution instruction mandates that the agent actively resolves
 727 contradictions when updating the file.

728 A.8 Pre-Finish Verification

```

729 ## Pre-Finish Verification
730
731 Before calling finish(success=True, ...), you MUST:
732
733 1. Re-read every file you modified and verify the changes
734   are correct.
735 1. Run the required checks (lint, typecheck, tests) and fix
736   any failures.
737 1. Explicitly check each user requirement against what was
738   delivered.
739 1. If any check fails, continue working instead of
740   finishing.
741 1. If you have retried the same fix 3 times without
742   progress, step back, rethink the approach from scratch,
743   and try a different strategy.
744

```

746 The three-retry threshold forces the model to break out of repetitive loops by abandoning the current
 747 approach and reconsidering the problem from first principles.

748 A.9 File Browsing Protocol

```

749 ## Browsing files for a task
750
751
752 - When you need to read files for a task, collect
753   information, including code snippets necessary for the
754   task, without much thinking, in a new file
755   PWD/tmp/file-information-{unique_id}.md. Then go over
756   the information in
757   PWD/tmp/file-information-{unique_id}.md, think deeply
758   on how to complete the task at hand.

```

760 This instruction applies the same two-phase collect-then-synthesize discipline used for web research
 761 (Section 4.2) to the local file system. By writing a structured summary of each file's relevant

information into a temporary markdown file, the agent externalizes its understanding into a compact artifact that is typically much smaller than the raw source files, freeing up context window capacity for subsequent reasoning and editing.

A.10 Desktop Application Control

```
## Launch desktop apps
- Use screenshots, keyboard, and mouse to control a desktop
  app.
- Do not launch VS Code or its extensions.
```

The prohibition on launching VS Code prevents a recursive loop: since the agent runs inside a VS Code extension, launching another instance could corrupt the host session or create deadlocks.

B Painless Software Engineering

A central claim of our system is that natural-language interaction can replace manual code inspection and ad hoc scripting for understanding and evolving nontrivial subsystems. We illustrate this with a real development session in which the worktree merge workflow (Section 2.5) was first understood and then redesigned entirely through conversational prompts.

Step 1: Understanding the existing workflow. The developer asks the agent to explain the current post-task git lifecycle. The agent reads the source code and returns a structured summary of the four-phase lifecycle.

Step 2: Simplifying the workflow. The developer decides the three-way choice (auto-merge, manual merge, discard) is unnecessarily complex and issues a redesign request in natural language. The agent modifies six files across Python and TypeScript, removes the manual-merge method, simplifies the UI to two buttons, updates tests, and verifies all 28 worktree tests pass.

Step 3: Investigating unexpected state. After testing, the developer notices files appearing in the Source Control panel and asks why. The agent traces the execution path and discovers the implementation deliberately unstages changes for user review.

Step 4: Directing a design change. The developer decides the extra review step is unnecessary and says “Fix it” in one sentence. The agent replaces the unstage call with a conditional commit, adds an edge-case guard, updates one test, and verifies all 104 worktree tests pass.

The developer never opens a source file, never writes a line of code, and never runs a test manually. This style of development becomes possible because of the agent hierarchy: the Worktree Agent isolates changes on a branch, the Chat Agent preserves context across tasks, and the Relentless Agent continues when the context window is exhausted.

NeurIPS Paper Checklist

1. Claims

Question: Do the main claims made in the abstract and introduction accurately reflect the paper’s contributions and scope?

Answer: [Yes]

Justification: The abstract and introduction (Section 1) clearly state the contributions—a five-layer agent architecture, a structured system prompt, a VS Code extension, and a 62.2% pass rate on Terminal-Bench 2.0—and the scope is limited to software engineering tasks.

Guidelines:

- The answer [N/A] means that the abstract and introduction do not include the claims made in the paper.

- The abstract and/or introduction should clearly state the claims made, including the contributions made in the paper and important assumptions and limitations. A [No] or [N/A] answer to this question will not be perceived well by the reviewers.
- The claims made should match theoretical and experimental results, and reflect how much the results can be expected to generalize to other settings.
- It is fine to include aspirational goals as motivation as long as it is clear that these goals are not attained by the paper.

2. Limitations

Question: Does the paper discuss the limitations of the work performed by the authors?

Answer: [Yes]

Justification: Section 3 discusses consistently failed tasks and their failure categories (graphical/multimedia requirements, resource-intensive builds, niche domain knowledge). The conclusion acknowledges that results depend on a single frontier model (Claude Opus 4.6) and a single benchmark.

Guidelines:

- The answer [N/A] means that the paper has no limitation while the answer [No] means that the paper has limitations, but those are not discussed in the paper.
- The authors are encouraged to create a separate “Limitations” section in their paper.
- The paper should point out any strong assumptions and how robust the results are to violations of these assumptions (e.g., independence assumptions, noiseless settings, model well-specification, asymptotic approximations only holding locally). The authors should reflect on how these assumptions might be violated in practice and what the implications would be.
- The authors should reflect on the scope of the claims made, e.g., if the approach was only tested on a few datasets or with a few runs. In general, empirical results often depend on implicit assumptions, which should be articulated.
- The authors should reflect on the factors that influence the performance of the approach. For example, a facial recognition algorithm may perform poorly when image resolution is low or images are taken in low lighting. Or a speech-to-text system might not be used reliably to provide closed captions for online lectures because it fails to handle technical jargon.
- The authors should discuss the computational efficiency of the proposed algorithms and how they scale with dataset size.
- If applicable, the authors should discuss possible limitations of their approach to address problems of privacy and fairness.
- While the authors might fear that complete honesty about limitations might be used by reviewers as grounds for rejection, a worse outcome might be that reviewers discover limitations that aren’t acknowledged in the paper. The authors should use their best judgment and recognize that individual actions in favor of transparency play an important role in developing norms that preserve the integrity of the community. Reviewers will be specifically instructed to not penalize honesty concerning limitations.

3. Theory assumptions and proofs

Question: For each theoretical result, does the paper provide the full set of assumptions and a complete (and correct) proof?

Answer: [N/A]

Justification: The paper does not include theoretical results. It is a systems paper presenting an architecture and empirical evaluation.

Guidelines:

- The answer [N/A] means that the paper does not include theoretical results.
- All the theorems, formulas, and proofs in the paper should be numbered and cross-referenced.
- All assumptions should be clearly stated or referenced in the statement of any theorems.

- The proofs can either appear in the main paper or the supplemental material, but if they appear in the supplemental material, the authors are encouraged to provide a short proof sketch to provide intuition.
- Inversely, any informal proof provided in the core of the paper should be complemented by formal proofs provided in appendix or supplemental material.
- Theorems and Lemmas that the proof relies upon should be properly referenced.

4. Experimental result reproducibility

Question: Does the paper fully disclose all the information needed to reproduce the main experimental results of the paper to the extent that it affects the main claims and/or conclusions of the paper (regardless of whether the code and data are provided or not)?

Answer: [Yes]

Justification: Section 3 describes the benchmark (Terminal-Bench 2.0), the evaluation framework (Harbor), the model (Claude Opus 4.6), the hardware (2025 MacBook Air M4, 24 GB RAM), the number of trials (5 per task), and the 9 skipped tasks. The full agent architecture is described in Section 2 and the system prompt in Section 4 and Appendix A.

Guidelines:

- The answer [N/A] means that the paper does not include experiments.
- If the paper includes experiments, a [No] answer to this question will not be perceived well by the reviewers: Making the paper reproducible is important, regardless of whether the code and data are provided or not.
- If the contribution is a dataset and/or model, the authors should describe the steps taken to make their results reproducible or verifiable.
- Depending on the contribution, reproducibility can be accomplished in various ways. For example, if the contribution is a novel architecture, describing the architecture fully might suffice, or if the contribution is a specific model and empirical evaluation, it may be necessary to either make it possible for others to replicate the model with the same dataset, or provide access to the model. In general, releasing code and data is often one good way to accomplish this, but reproducibility can also be provided via detailed instructions for how to replicate the results, access to a hosted model (e.g., in the case of a large language model), releasing of a model checkpoint, or other means that are appropriate to the research performed.
- While NeurIPS does not require releasing code, the conference does require all submissions to provide some reasonable avenue for reproducibility, which may depend on the nature of the contribution. For example
 - (a) If the contribution is primarily a new algorithm, the paper should make it clear how to reproduce that algorithm.
 - (b) If the contribution is primarily a new model architecture, the paper should describe the architecture clearly and fully.
 - (c) If the contribution is a new model (e.g., a large language model), then there should either be a way to access this model for reproducing the results or a way to reproduce the model (e.g., with an open-source dataset or instructions for how to construct the dataset).
 - (d) We recognize that reproducibility may be tricky in some cases, in which case authors are welcome to describe the particular way they provide for reproducibility. In the case of closed-source models, it may be that access to the model is limited in some way (e.g., to registered users), but it should be possible for other researchers to have some path to reproducing or verifying the results.

5. Open access to data and code

Question: Does the paper provide open access to the data and code, with sufficient instructions to faithfully reproduce the main experimental results, as described in supplemental material?

Answer: [No]The code repository link is withheld to preserve double-blind review. The system is open-source and the code will be made available upon acceptance.

Justification: The open-source repository link is withheld to respect double-blind review (stated in Section 1). The system will be publicly released upon acceptance.

Guidelines:

- The answer [N/A] means that paper does not include experiments requiring code.
- Please see the NeurIPS code and data submission guidelines (<https://neurips.cc/public/guides/CodeSubmissionPolicy>) for more details.
- While we encourage the release of code and data, we understand that this might not be possible, so [No] is an acceptable answer. Papers cannot be rejected simply for not including code, unless this is central to the contribution (e.g., for a new open-source benchmark).
- The instructions should contain the exact command and environment needed to run to reproduce the results. See the NeurIPS code and data submission guidelines (<https://neurips.cc/public/guides/CodeSubmissionPolicy>) for more details.
- The authors should provide instructions on data access and preparation, including how to access the raw data, preprocessed data, intermediate data, and generated data, etc.
- The authors should provide scripts to reproduce all experimental results for the new proposed method and baselines. If only a subset of experiments are reproducible, they should state which ones are omitted from the script and why.
- At submission time, to preserve anonymity, the authors should release anonymized versions (if applicable).
- Providing as much information as possible in supplemental material (appended to the paper) is recommended, but including URLs to data and code is permitted.

6. Experimental setting/details

Question: Does the paper specify all the training and test details (e.g., data splits, hyperparameters, how they were chosen, type of optimizer) necessary to understand the results?

Answer: [Yes]

Justification: Section 3 specifies the benchmark, model, hardware, number of trials, skipped tasks, and evaluation methodology. No training or fine-tuning is performed.

Guidelines:

- The answer [N/A] means that the paper does not include experiments.
- The experimental setting should be presented in the core of the paper to a level of detail that is necessary to appreciate the results and make sense of them.
- The full details can be provided either with the code, in appendix, or as supplemental material.

7. Experiment statistical significance

Question: Does the paper report error bars suitably and correctly defined or other appropriate information about the statistical significance of the experiments?

Answer: [Yes]

Justification: We run 5 independent trials per task (445 total runs) and report pass@any (78.7%), pass@all (43.8%), and overall pass rate (62.2%), as well as the distribution of always-pass, always-fail, and mixed-result tasks. Median and mean cost and duration are reported.

Guidelines:

- The answer [N/A] means that the paper does not include experiments.
- The authors should answer [Yes] if the results are accompanied by error bars, confidence intervals, or statistical significance tests, at least for the experiments that support the main claims of the paper.
- The factors of variability that the error bars are capturing should be clearly stated (for example, train/test split, initialization, random drawing of some parameter, or overall run with given experimental conditions).
- The method for calculating the error bars should be explained (closed form formula, call to a library function, bootstrap, etc.)
- The assumptions made should be given (e.g., Normally distributed errors).
- It should be clear whether the error bar is the standard deviation or the standard error of the mean.

- It is OK to report 1-sigma error bars, but one should state it. The authors should preferably report a 2-sigma error bar than state that they have a 96% CI, if the hypothesis of Normality of errors is not verified.
- For asymmetric distributions, the authors should be careful not to show in tables or figures symmetric error bars that would yield results that are out of range (e.g., negative error rates).
- If error bars are reported in tables or plots, the authors should explain in the text how they were calculated and reference the corresponding figures or tables in the text.

8. Experiments compute resources

Question: For each experiment, does the paper provide sufficient information on the computer resources (type of compute workers, memory, time of execution) needed to reproduce the experiments?

Answer: [Yes]

Justification: Section 3 specifies the hardware (2025 MacBook Air 15" with M4 processor and 24 GB RAM) and reports median/mean cost (\$0.45/\$0.90 per trial) and median/mean duration (202 s/446 s per trial).

Guidelines:

- The answer [N/A] means that the paper does not include experiments.
- The paper should indicate the type of compute workers CPU or GPU, internal cluster, or cloud provider, including relevant memory and storage.
- The paper should provide the amount of compute required for each of the individual experimental runs as well as estimate the total compute.
- The paper should disclose whether the full research project required more compute than the experiments reported in the paper (e.g., preliminary or failed experiments that didn't make it into the paper).

9. Code of ethics

Question: Does the research conducted in the paper conform, in every respect, with the NeurIPS Code of Ethics <https://neurips.cc/public/EthicsGuidelines>?

Answer: [Yes]

Justification: We have reviewed the NeurIPS Code of Ethics. The research involves building and evaluating a software engineering assistant; no human subjects, sensitive data, or dual-use concerns are involved.

Guidelines:

- The answer [N/A] means that the authors have not reviewed the NeurIPS Code of Ethics.
- If the authors answer [No], they should explain the special circumstances that require a deviation from the Code of Ethics.
- The authors should make sure to preserve anonymity (e.g., if there is a special consideration due to laws or regulations in their jurisdiction).

10. Broader impacts

Question: Does the paper discuss both potential positive societal impacts and negative societal impacts of the work performed?

Answer: [Yes]

Justification: The paper discusses positive impacts (making software development more accessible, reducing developer toil). Potential negative impacts include job displacement for routine programming tasks and the risk of generating vulnerable code if the system prompt's verification disciplines are removed. The system mitigates the latter through mandatory pre-finish verification (Appendix A.8) and test-first discipline (Appendix A.6).

Guidelines:

- The answer [N/A] means that there is no societal impact of the work performed.
- If the authors answer [N/A] or [No], they should explain why their work has no societal impact or why the paper does not address societal impact.

- Examples of negative societal impacts include potential malicious or unintended uses (e.g., disinformation, generating fake profiles, surveillance), fairness considerations (e.g., deployment of technologies that could make decisions that unfairly impact specific groups), privacy considerations, and security considerations.
- The conference expects that many papers will be foundational research and not tied to particular applications, let alone deployments. However, if there is a direct path to any negative applications, the authors should point it out. For example, it is legitimate to point out that an improvement in the quality of generative models could be used to generate Deepfakes for disinformation. On the other hand, it is not needed to point out that a generic algorithm for optimizing neural networks could enable people to train models that generate Deepfakes faster.
- The authors should consider possible harms that could arise when the technology is being used as intended and functioning correctly, harms that could arise when the technology is being used as intended but gives incorrect results, and harms following from (intentional or unintentional) misuse of the technology.
- If there are negative societal impacts, the authors could also discuss possible mitigation strategies (e.g., gated release of models, providing defenses in addition to attacks, mechanisms for monitoring misuse, mechanisms to monitor how a system learns from feedback over time, improving the efficiency and accessibility of ML).

11. Safeguards

Question: Does the paper describe safeguards that have been put in place for responsible release of data or models that have a high risk for misuse (e.g., pre-trained language models, image generators, or scraped datasets)?

Answer: [N/A]

Justification: The paper does not release a pretrained language model or dataset. The system is an agent framework that wraps existing commercial LLM APIs.

Guidelines:

- The answer [N/A] means that the paper poses no such risks.
- Released models that have a high risk for misuse or dual-use should be released with necessary safeguards to allow for controlled use of the model, for example by requiring that users adhere to usage guidelines or restrictions to access the model or implementing safety filters.
- Datasets that have been scraped from the Internet could pose safety risks. The authors should describe how they avoided releasing unsafe images.
- We recognize that providing effective safeguards is challenging, and many papers do not require this, but we encourage authors to take this into account and make a best faith effort.

12. Licenses for existing assets

Question: Are the creators or original owners of assets (e.g., code, data, models), used in the paper, properly credited and are the license and terms of use explicitly mentioned and properly respected?

Answer: [Yes]

Justification: All tools, benchmarks, and models used are cited: Terminal-Bench 2.0, Harbor, Claude Opus 4.6, Playwright, VS Code, and all related works. The system is open-source.

Guidelines:

- The answer [N/A] means that the paper does not use existing assets.
- The authors should cite the original paper that produced the code package or dataset.
- The authors should state which version of the asset is used and, if possible, include a URL.
- The name of the license (e.g., CC-BY 4.0) should be included for each asset.
- For scraped data from a particular source (e.g., website), the copyright and terms of service of that source should be provided.

- If assets are released, the license, copyright information, and terms of use in the package should be provided. For popular datasets, paperswithcode.com/datasets has curated licenses for some datasets. Their licensing guide can help determine the license of a dataset.
- For existing datasets that are re-packaged, both the original license and the license of the derived asset (if it has changed) should be provided.
- If this information is not available online, the authors are encouraged to reach out to the asset's creators.

13. New assets

Question: Are new assets introduced in the paper well documented and is the documentation provided alongside the assets?

Answer: [N/A]

Justification: No new datasets or pretrained models are released. The open-source code will be documented upon release.

Guidelines:

- The answer [N/A] means that the paper does not release new assets.
- Researchers should communicate the details of the dataset/code/model as part of their submissions via structured templates. This includes details about training, license, limitations, etc.
- The paper should discuss whether and how consent was obtained from people whose asset is used.
- At submission time, remember to anonymize your assets (if applicable). You can either create an anonymized URL or include an anonymized zip file.

14. Crowdsourcing and research with human subjects

Question: For crowdsourcing experiments and research with human subjects, does the paper include the full text of instructions given to participants and screenshots, if applicable, as well as details about compensation (if any)?

Answer: [N/A]

Justification: The paper does not involve crowdsourcing or research with human subjects.

Guidelines:

- The answer [N/A] means that the paper does not involve crowdsourcing nor research with human subjects.
- Including this information in the supplemental material is fine, but if the main contribution of the paper involves human subjects, then as much detail as possible should be included in the main paper.
- According to the NeurIPS Code of Ethics, workers involved in data collection, curation, or other labor should be paid at least the minimum wage in the country of the data collector.

15. Institutional review board (IRB) approvals or equivalent for research with human subjects

Question: Does the paper describe potential risks incurred by study participants, whether such risks were disclosed to the subjects, and whether Institutional Review Board (IRB) approvals (or an equivalent approval/review based on the requirements of your country or institution) were obtained?

Answer: [N/A]

Justification: The paper does not involve human subjects research.

Guidelines:

- The answer [N/A] means that the paper does not involve crowdsourcing nor research with human subjects.
- Depending on the country in which research is conducted, IRB approval (or equivalent) may be required for any human subjects research. If you obtained IRB approval, you should clearly state this in the paper.

- We recognize that the procedures for this may vary significantly between institutions and locations, and we expect authors to adhere to the NeurIPS Code of Ethics and the guidelines for their institution.
- For initial submissions, do not include any information that would break anonymity (if applicable), such as the institution conducting the review.

16. Declaration of LLM usage

Question: Does the paper describe the usage of LLMs if it is an important, original, or non-standard component of the core methods in this research? Note that if the LLM is used only for writing, editing, or formatting purposes and does *not* impact the core methodology, scientific rigor, or originality of the research, declaration is not required.

Answer: [\[Yes\]](#)

Justification: The entire system is built around LLM usage. Section 2 describes how LLMs are used as the core reasoning engine via native function calling (Section 2.1). The paper also discloses that the system was built using itself (Section 1).

Guidelines:

- The answer [\[N/A\]](#) means that the core method development in this research does not involve LLMs as any important, original, or non-standard components.
- Please refer to our LLM policy in the NeurIPS handbook for what should or should not be described.