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

Anonymous Author(s)

Affiliation

Address

email

Abstract

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

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

17 We implemented KS Assistant as a free, open-source Visual Studio Code exten-
18 sion that runs locally, is effective for long-horizon tasks, and supports browser
19 automation, multimodal input, and Docker containers. We deliberately prioritize
20 output quality over speed: giving a frontier model adequate time to validate its
21 own output—running linters, type checkers, and tests—dramatically reduces the
22 low-quality code that plagues faster but less thorough agents. The entire system
23 was built using itself over four months, providing a continuous stress test in which
24 any bug was patched immediately after its manifestation. On Terminal-Bench 2.0,
25 KS Assistant achieves a 62.2% overall pass rate with Claude Opus 4.6 [Anthropic,
26 2026a], comparing favorably to Claude Code (58%) and Cursor Composer 2
27 (61.7%). These results are achieved without benchmark-specific prompt tuning or
28 model fine-tuning.

1 Introduction

30 Modern large language models (LLMs), such as Claude Opus 4.7 [Anthropic, 2026b], GPT 5.5 [Ope-
31 nAI, 2026], and Gemini 3.1 [Google DeepMind, 2026], have demonstrated a remarkable ability to
32 generate code, reason about software architecture, and use developer tools [Chen et al., 2021, Rozière
33 et al., 2023]. A growing body of work has explored how to harness these capabilities for autonomous
34 software engineering, from single-session agents that resolve GitHub issues [Yang et al., 2024,
35 Wang et al., 2024] to industrial products marketed as AI software and general assistants [GitHub,
36 2021, Cursor, 2024, Cognition Labs, 2024, Anthropic, 2025, OpenAI, 2025a, OpenClaw AI, 2025].
37 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 principles and techniques are invaluable for building reliable, practical agent systems.

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. Eventually, no prompt optimization techniques were used when creating the system prompt for KS Assistant; it was hand-tuned based on long-term experience with the agent and its behavior. The framework uses five agent layers combining composition and inheritance. Each layer delegates upward for concerns it does not own.

2.1 KS Agent

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

90

```
from ks.core.ks_agent import KSAgent

def calculate(expression: str) -> str:
    """Evaluate a math expression."""
    return str(eval(expression))

agent = KSAgent(name="Math Buddy")
result = agent.run(
    model_name="gemini-2.5-flash",
    prompt_template="Calculate: {question}",
    arguments={"question": "15% of 847?"},
    tools=[calculate]
)
print(result) # 127.05
```

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

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 (1/5 passes)	78.7% (70/89)
pass@all (5/5 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 / trial	202 s
Mean duration / trial	446 s

Native function calling. Tools are registered as ordinary Python callables. The agent builds an OpenAI-compatible tool schema once at setup time and caches it. A special `finish` tool signals task completion and returns the result. Using a model’s native calling API avoids the mistakes that models make on custom function calling conventions.

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

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

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

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

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

2.2 Relentless Agent

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

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

123 **Continuation protocol.** The `finish` tool accepts three fields: a success flag, a continue flag, and
124 a summary. When `is_continue=True`, the Relentless Agent starts a new sub-session with a fresh
125 context window. The prompt for the new session includes a chronologically ordered list of all prior
126 attempt summaries, instructs the agent not to redo completed work, and advises it to step back and
127 rethink the strategy from scratch if it has been retrying the same approach without progress.

128 **Forced continuation on failure.** If a sub-session raises an exception (e.g., the step limit is hit before
129 calling `finish`), the Relentless Agent saves the full trajectory to a temporary file, spawns a separate
130 summarizer agent to produce a concise summary, and uses that summary as the progress text for the
131 next sub-session. This ensures that even crashed sessions contribute useful context. The summarizer
132 is instructed to read the (potentially large) trajectory file and return a precise, chronologically-ordered
133 list of things the agent did, with the reason for each action and relevant code snippets.

134 **Pre-emptive continuation.** To force the agent to self-continue before exhausting its step budget, the
135 system prompt is augmented with a step-threshold instruction that requires the agent, at a designated
136 step or whenever the task is at risk of running out of steps or context length, to call `finish` with
137 `success=False`, `is_continue=True`, and a precise chronologically-ordered summary of work
138 done so far. This instruction is injected near the end of the budget window. Without it, the agent
139 exhibits a “rush to finish” behavior when steps are running low—skipping verification, making hasty
140 edits, and calling `finish` with an incomplete result. The explicit step threshold redirects that urgency
141 into the continuation protocol, ensuring that a clean handoff to a new sub-session produces better
142 results than a frantic attempt to squeeze everything into the remaining steps.

143 2.3 Tool Agent

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

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

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

152 **Parallel sub-agents.** An optional parallel execution tool spawns independent Tool Agent instances
153 in a thread pool. Each sub-agent gets its own LLM context and tool set, useful for embarrassingly
154 parallel tasks such as summarizing multiple files or researching independent topics. Results are
155 collected and returned in input order. This tool is not activated by default because the IDE cannot
156 stream multiple agent outputs coherently in the chat window.

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

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

161 2.4 Chat Agent

162 The Chat Agent adds multi-turn conversation persistence.

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

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

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

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

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

178 2.5 Worktree Agent

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

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

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

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

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

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

193 3 Evaluation on Terminal-Bench 2.0

194 We evaluate on Terminal-Bench 2.0,¹ a benchmark comprising 89 diverse terminal-based program-
195 ming tasks, ranging from building legacy compilers and configuring servers to solving cryptanalysis
196 challenges and training machine-learning models. Each task runs in an isolated Docker container;
197 a separate verifier automatically judges the result. We use the Harbor² framework to orchestrate
198 execution and Claude Opus 4.6 [Anthropic, 2026a] as the underlying LLM. We do not modify the
199 general system prompt or inject benchmark-specific instructions. Evaluation was carried out on a
200 2025 MacBook Air 15” with an M4 processor and 24 GB RAM.

201 3.1 Setup

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

206 3.2 Aggregate results

207 Table 1 (shown earlier alongside Listing 1) summarizes the aggregate statistics.

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

¹<https://www.tbench.ai/>

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

3.3 Task-level breakdown

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

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

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

Leaderboard context. KS Assistant does not score as high as some coding agents on the Terminal-Bench 2.0 leaderboard, but we believe that our results are significant because we didn’t tune our prompts or any model specifically for the Terminal Bench 2.0. We simply used the general system prompt and the Claude Opus 4.6 model. Regarding low score compared to other coding agents, we wanted to note that recent analysis has found widespread cheating on popular agent benchmarks, including Terminal-Bench 2.0: the top three submissions commit harness-level cheating (e.g., leaking verifier code or answer keys into the agent’s environment), and task-level cheating (e.g., Googling answers, mining git history, hardcoding test outputs) affects 28+ submissions across 9 benchmarks [Stein et al., 2026a,b]. Separately, an automated benchmark audit found 45 confirmed hacking solutions across 13 widely used benchmarks exhibiting process-isolation failures, answer leakage, and weak test assertions that allow perfect scores without solving a single problem [Wang et al., 2026].

4 The system prompt

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

4.1 Execution mindset

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

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

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

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

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

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

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

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

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

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

277 These two sentences occupy only two lines, yet they establish a behavioral contract that permeates
278 every subsequent interaction. We observe that removing any single directive degrades the agent’s
279 behavior along the corresponding dimension while developing KS Assistant (e.g., removing “BE
280 RELENTLESS” causes the agent to give up after the first error more frequently).

281 4.2 Web research protocol

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

```
284 ## Web Research (MANDATORY)
285
286
287 - **Visit >=30 websites every search. Hard requirement---don't stop before 30 or rationalize fewer.**
288 - Procedure:
289   1. Create PWD/tmp/information-{unique_id}.md: '# Web Research --- Websites visited: 0/30'
290   1. Per site, append: '## [N/30] URL' + extracted info. Update header counter each visit.
291   1. **Don't proceed until counter >=30.**
292   1. If results dry up, try different queries, synonyms, official docs, GitHub repos/issues, Stack Overflow
293     , blogs, Reddit, papers, API refs.
294   1. After 30, review and think deeply.
295 - Ask user for login help when needed.
```

297 The rationale is a two-phase separation between *collection* and *synthesis*. LLMs tend to anchor on
298 the first few results they encounter, biasing their solutions toward a narrow slice of the design space.
299 By forcing the agent to accumulate a broad set of information into a file *before* reasoning about it, the
300 protocol counteracts anchoring bias and encourages the model to consider diverse approaches. The
301 “visit ≥ 30 websites” threshold is deliberately aggressive: it ensures the agent does not shortcut the
302 research phase after two or three hits, and the resulting information file serves as an auditable artifact
303 of what the agent considered.

304 This two-phase collect-then-synthesize discipline is also applied to local file browsing (Appendix A.9):
305 the agent writes a structured summary of each file’s relevant information into a temporary markdown
306 file, externalizing its understanding into a compact artifact that persists across context boundaries.
307 Analysis happens in the second phase, when the agent reads its own summary and reasons about the
308 collected information as a whole.

309 5 VS Code extension: unique features

310 We release our system as a VS Code extension and a web app. While the underlying agent architecture
311 already differs from existing AI coding assistants, the extension introduces several features that,
312 to our knowledge, are not present in competing assistants—including GitHub Copilot [GitHub,
313 2021], Cursor [Cursor, 2024], Windsurf [Codeium, 2024], Devin [Cognition Labs, 2024], and
314 Aider [Gauthier, 2023].

315 **Cross-session self-improvement.** The extension maintains a `USER_PREFS.md` file that the agent
316 reads at the start of each task and *updates* at the end. When the agent discovers a new preference or

project invariant during task execution, it writes it to the file. The agent also resolves conflicts: if a new preference contradicts an existing one, the old entry is removed. Over time, the file accumulates a curated set of project-specific knowledge, making the agent progressively more effective without any manual configuration. This mechanism is detailed in Appendix A.7.

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

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

6 Painless software engineering with KS Assistant

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. While developing KS Assistant, we found a particular workflow to be useful. In the workflow, we first ask KS Assistant to generate a detailed, step-by-step description of a workflow or algorithm we found buggy, and then we ask KS Assistant to revise some of the buggy steps in natural language. We illustrate this with a real development session drawn from the project’s own history, in which the worktree merge workflow (Section 2.5) was first understood and then redesigned entirely through conversational prompts. The session comprises four consecutive tasks; we reproduce the prompts verbatim and summarize the agent’s responses.

Step 1: Understanding the existing workflow. The developer begins by asking the agent to explain the current post-task git lifecycle:

Can you tell me what happens, step by step, with git in `worktree_sorcar_agent.py` when a task finishes?

The agent reads the source code and returns a structured summary of the four-phase lifecycle: (1) during `run()`, a new branch and worktree are created and the task executes inside the worktree; when the task completes, *nothing is committed or merged*—the result is returned with merge instructions appended and the worktree stays pending; (2) `merge()` calls `_finalize_worktree()`, which stages all changes, generates a commit message via the LLM, commits, removes the worktree, checks out the original branch, and runs `git merge`; (3) `discard()` removes the worktree, prunes, checks out the original branch, and deletes the task branch; (4) in CLI mode an interactive prompt forces the user to choose `[c]ommit` and `merge` or `[d]iscard` before exiting. The agent also notes a key design invariant: nothing auto-merges; auto-commit occurs only at merge/finalize time; all steps are idempotent; and state can be recovered from git on restart.

Step 2: Simplifying the workflow via natural language. Armed with the workflow description, the developer decides the three-way choice (auto-merge, manual merge, discard) is unnecessarily complex and issues a redesign request:

Can you change `worktree_sorcar_agent.py` and the extension so that after the agent finishes its task, it simply asks "Commit and Merge" or "Discard". When "Commit and Merge" is clicked by the user, the agent must commit the changes with a generated commit message, merge the branch with the original branch, and delete the worktree and the branch associated with the worktree. If the user clicks "discard", it must delete the worktree and the branch, and checkout the original branch.

The agent modifies six files across Python and TypeScript: it updates `discard()` to check out the original branch before deleting the task branch, removes the `manual_merge()` method entirely, simplifies `merge_instructions()` to show only two options, updates the webview UI to replace the three-button toolbar with a two-button “Commit and Merge or Discard?” bar, removes the `manual` action type from the TypeScript type definitions, and removes the corresponding handler from the Python backend. Three tests for the deleted manual-merge path have been removed, and one routing test has been updated. All 28 worktree tests pass after the change.

371 We skip the description of the remaining 2 tasks in the paper in the interest of space, but they can be
372 found in the Appendix.

373 7 Related work

374 **Code-specialized language models.** Code Llama [Rozière et al., 2023] fine-tunes Llama 2 for code
375 generation. StarCoder [Li et al., 2023] trains on permissively licensed code. DeepSeek-Coder [Guo
376 et al., 2024] trains on a 2-trillion-token corpus. Frontier models such as Claude Opus 4.7 [Anthropic,
377 2026b], GPT 5.5 [OpenAI, 2026], Kimi K2.5 [Kimi Team, 2026], and GLM-5.1 [Z.ai, 2026] target
378 agentic software engineering. Our system is model-agnostic and benefits from advances in code-
379 specialized pre-training without architectural changes.

380 **Code generation agents.** SWE-Agent [Yang et al., 2024] and OpenHands [Wang et al., 2024] provide
381 LLM-based agents for resolving software engineering tasks. Agentless [Xia et al., 2024] shows that
382 a two-phase localize-then-repair pipeline can achieve competitive results. Devin [Cognition Labs,
383 2024], Claude Code [Anthropic, 2025], Codex [OpenAI, 2025a], and Aider [Gauthier, 2023] offer
384 various approaches to autonomous coding. Cursor released Composer 2 [Cursor Research, 2026], a
385 fine-tuned coding model trained with large-scale reinforcement learning.

386 **Reasoning, planning, and multi-agent systems.** ReAct [Yao et al., 2023b], chain-of-thought
387 prompting [Wei et al., 2022], Tree of Thoughts [Yao et al., 2023a], and Reflexion [Shinn et al., 2023]
388 structure step-by-step reasoning, search, and verbal self-critique; our continuation protocol relates
389 to Reflexion but uses summaries to continue tasks rather than retry them. ChatDev [Qian et al.,
390 2024], MetaGPT [Hong et al., 2024], and AutoGen [Wu et al., 2023] model software development
391 as multi-agent conversations; we use a single agent with broad tool access and optional parallel
392 sub-agents.

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

397 **Benchmarks, agentic SE, and self-improvement.** SWE-bench [Jimenez et al., 2024], Hu-
398 manEval [Chen et al., 2021], LiveCodeBench [Jain et al., 2024], and SWE-bench Pro [Deng et al.,
399 2025] evaluate coding agents at various scales; Prathifkumar et al. [2025] raises concerns about
400 benchmark contamination. Recent surveys and studies articulate the pillars of agentic software
401 engineering, autonomous coding agents, agentic programming, agent context files, and context
402 engineering [Hassan et al., 2025, Li et al., 2025, Wang et al., 2025, Chatlatanagulchai et al., 2025,
403 Mohsenimofidi et al., 2025]; our system prompt and per-repository override mechanisms instantiate
404 these context-engineering patterns. Robeyns et al. [2025] demonstrates a self-improving coding
405 agent; our self-improvement loop is a lighter-weight form via accumulated preferences. Gao et al.
406 [2025] applies test-time compute scaling to software engineering, and Get Shit Done [TÂCHES,
407 2025] is a meta-prompting system for coding agents addressing context rot through phased planning
408 documents that inspired parts of our system prompt.

409 8 Conclusion

410 A simple layered, single-concern architecture combined with a system prompt that encodes en-
411 gineering discipline addresses the practical challenges of deploying LLM agents for real-world
412 software development. On Terminal-Bench 2.0, our system reaches 62.2% (277/445), outperforming
413 Claude Code (58%) and Cursor Composer 2 (61.7%) on the same benchmark and base model [An-
414 thropic, 2026a] without benchmark-specific tuning, fine-tuning, or reinforcement learning. Time-
415 tested software engineering principles, expressed as concrete instructions an LLM can follow, matter
416 more than model-level customization for building reliable, practical agent systems.

417 **Limitations.** Our evaluation uses a single model and benchmark; broader evaluation, especially on
418 benchmarks like SWE-bench Pro, is future work. Continuation summaries may lose fidelity, and our
419 quality-over-speed posture may not suit latency-sensitive workflows.

References

- Lakshya A. Agrawal, Shangyin Tan, Dilara Soylu, Noah Ziemis, Rishi Khare, Krista Opsahl-Ong, Arnav Singhvi, Herumb Shandilya, Michael J. Ryan, Meng Jiang, Christopher Potts, Koushik Sen, Alexandros G. Dimakis, Ion Stoica, Dan Klein, Matei Zaharia, and Omar Khattab. GEPA: Reflective prompt evolution can outperform reinforcement learning. In *International Conference on Learning Representations (ICLR)*, 2026. Oral. arXiv preprint arXiv:2507.19457.
- Algorithmic Superintelligence. OpenEvolve: Open-source implementation of AlphaEvolve. <https://github.com/algorithmicsuperintelligence/openevolve>, 2025.
- Anthropic. Claude Code: Anthropic’s agentic coding system. <https://www.anthropic.com/product/claude-code>, 2025.
- Anthropic. Introducing Claude Opus 4.6. <https://www.anthropic.com/news/claude-opus-4-6>, 2026a.
- Anthropic. Introducing Claude Opus 4.7. <https://www.anthropic.com/news/claude-opus-4-7>, 2026b.
- Worawalan Chatlatanagulchai, Hao Li, Yutaro Kashiwa, Brittany Reid, Kundjanasith Thonglek, Pattara Leelaprite, Arnon Rungsawang, Bundit Manaskasemsak, Bram Adams, Ahmed E. Hassan, and Hajimu Iida. Agent READMEs: An empirical study of context files for agentic coding. *arXiv preprint arXiv:2511.12884*, 2025.
- Mark Chen, Jerry Tworek, Heewoo Jun, Qiming Yuan, Henrique Pondé Pinto, Jared Kaplan, Harri Edwards, Yuri Burda, Nicholas Joseph, Greg Brockman, et al. Evaluating large language models trained on code. *arXiv preprint arXiv:2107.03374*, 2021.
- Codeium. Windsurf: The AI-powered IDE. <https://windsurf.com>, 2024.
- Cognition Labs. Devin: The first AI software engineer. <https://devin.ai>, 2024.
- CrewAI, Inc. CrewAI: Framework for orchestrating role-playing, autonomous AI agents. <https://github.com/crewAIInc/crewAI>, 2024.
- Cursor. Cursor: The AI-first code editor. <https://cursor.sh>, 2024.
- Cursor Research. Composer 2 technical report. *arXiv preprint arXiv:2603.24477*, 2026.
- Xiang Deng, Jeff Da, Edwin Pan, Yannis Yiming He, Charles Ide, Kanak Garg, Niklas Lauffer, Andrew Park, Nitin Pasari, Chetan Rane, et al. SWE-Bench Pro: Can AI agents solve long-horizon software engineering tasks? *arXiv preprint arXiv:2509.16941*, 2025.
- Pengfei Gao, Zhao Tian, Xiangxin Meng, and Trae Research Team. Trae agent: An LLM-based agent for software engineering with test-time scaling. *arXiv preprint arXiv:2507.23370*, 2025.
- Paul Gauthier. Aider: AI pair programming in your terminal. <https://github.com/paul-gauthier/aider>, 2023.
- GitHub. GitHub Copilot: Your AI pair programmer. <https://github.com/features/copilot>, 2021.
- Google. Agent Development Kit (ADK): An open-source framework for building AI agents. <https://google.github.io/adk-docs/>, 2025.
- Google DeepMind. Gemini 3.1 Pro. <https://deepmind.google/models/gemini/pro/>, 2026.
- Daya Guo, Qihao Zhu, Dejian Yang, Zhenda Xie, Kai Dong, Wentao Zhang, Guanting Chen, Xiao Bi, Y. Wu, Y. K. Li, Fuli Luo, Yun Xiong, and Wenfeng Liang. DeepSeek-Coder: When the large language model meets programming — the rise of code intelligence. *arXiv preprint arXiv:2401.14196*, 2024.
- Ahmed E. Hassan, Hao Li, Dayi Lin, Bram Adams, Tse-Hsun Chen, Yutaro Kashiwa, and Dong Qiu. Agentic software engineering: Foundational pillars and a research roadmap. *arXiv preprint arXiv:2509.06216*, 2025.
- Sirui Hong, Mingchen Zhuge, Jonathan Chen, Xiwu Zheng, Yuheng Cheng, Ceyao Zhang, Jinlin Wang, Zili Wang, Steven Ka Shing Yau, Zijuan Lin, Liyang Zhou, Chenyu Ran, Lingfeng Xiao, Chenglin Wu, and Jürgen Schmidhuber. MetaGPT: Meta programming for a multi-agent collaborative framework. In *International Conference on Learning Representations (ICLR)*, 2024.
- Naman Jain, King Han, Alex Gu, Wen-Ding Li, Fanjia Yan, Tianjun Zhang, Sida Wang, Armando Solar-Lezama, Koushik Sen, and Ion Stoica. LiveCodeBench: Holistic and contamination free evaluation of large language models for code. *arXiv preprint arXiv:2403.07974*, 2024.

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

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

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

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

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

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

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

485 Alexander Novikov, Ngân Vŭ, Marvin Eisenberger, Emilien Dupont, Po-Sen Huang, Adam Zsolt Wagner, Sergey
486 Shirobokov, Borislav Kozlovskii, Francisco J. R. Ruiz, Abbas Mehrabian, M. Pawan Kumar, Abigail See,
487 Swarat Chaudhuri, George Holland, Alex Davies, Sebastian Nowozin, Pushmeet Kohli, and Matej Balog.
488 AlphaEvolve: A coding agent for scientific and algorithmic discovery. *arXiv preprint arXiv:2506.13131*,
489 2025.

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

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

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

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

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

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

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

504 Aymeric Roucher, Albert Villanova del Moral, Thomas Wolf, Leandro von Werra, and Erik Kaunismäki. smola-
505 gents: A smol library to build great agentic systems. <https://github.com/huggingface/smolagents>,
506 2025.

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

510 Noah Shinn, Federico Cassano, Ashwin Gopinath, Karthik Narasimhan, and Shunyu Yao. Reflexion: Language
511 agents with verbal reinforcement learning. In *Advances in Neural Information Processing Systems (NeurIPS)*,
512 2023.

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

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

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

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

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

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

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

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

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

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

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

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

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

A Full system prompt details

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

A.1 Tool rules

Tool usage rules are explicit and mechanical:

```
# Rules
- PWD = current working directory. Write() for new files; Edit() for small changes.
- Run Bash synchronously with 'timeout_seconds' (default 300s). Retry with higher timeout on timeout. For
  >10 min commands, run in background, redirect output to file, poll periodically.
- Use go_to_url() for browser. Search the internet extensively.
- **User only sees the finish() summary. Include full details/results/outputs. Never include meta-
  descriptions like "Answered the user's question about X" or "Fixed the bug in Y".**
- Read large files in chunks. Temp files in PWD/tmp; clean up after.
- Use ULTRA thinking ALWAYS.
- **If running out of context/steps, don't rush---call finish(is_continue=true).**
```

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.

“Use go_to_url() for browser. Search the internet extensively.” The first clause specifies which tool to use for browser-based interactions. The second encourages the agent to proactively search the web when it encounters unfamiliar APIs or error messages, rather than relying on parametric memory.

“User only sees the finish() summary.” 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. The companion clause about meta-descriptions prevents vague summaries, requiring concrete details and outputs.

“Read large files in chunks. Temp files in PWD/tmp; clean up after.” 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. 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 running out of context/steps, don’t rush—call finish(is_continue=true).” 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:

```
## Pre-flight Checks
- Read every file before modifying it. Read relevant sources if the task depends on existing architecture.
- If referenced files/commands/config don't exist, stop and ask or report---don't guess.
- **When fixing bugs/issues/races: write tests to confirm first, then fix.**
```

“Read every file before modifying 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.

598 **“When fixing bugs/issues/races: write tests to confirm first, then fix.”** This mandates test-first
599 discipline: a test that reproduces the bug provides concrete verification that the fix is correct. Writing
600 the test forces the model to understand the bug precisely before attempting a fix.

601 A.3 Code style guidelines

602 The prompt encodes a minimalist code philosophy:

```
603 ## Code Style
604
605 - Simple, clean, readable code with minimal indirection. Organize in multiple files by functionality.
606 - Avoid unnecessary attributes, locals, config vars, tight coupling, and attribute redirections.
607 - DO NOT USE CLOSURES. No redundant abstractions or duplicate code.
608 - Public methods MUST have full documentation.
609 - Fix root causes, not symptoms. Think first: is the code simple, elegant, general, minimal?
610 - Don't write documentation unless the task requires it.
611
```

613 Each guideline addresses an anti-pattern commonly exhibited by LLM-generated code. For example,
614 “DO NOT USE CLOSURES” steers the model toward explicit data structures (plain functions with
615 arguments, classes with attributes) that are easier to test and reason about. “Think first: is the code
616 simple, elegant, general, minimal?” is a metacognitive instruction that asks the model to pause and
617 evaluate its plan, spending more inference-time compute on design.

618 A.4 Deep work rules

```
619 ## Deep Work
620
621 - For "align"/"match"/"make consistent": read the target state before editing. Never edit from vague
622   references.
623 - Use concrete values, not indirections (read Y first, then write specific values into X).
624 - List concrete planned changes before executing multi-part work.
625 - Every meaningful change needs a concrete verification method (test, grep, CLI).
626
```

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

630 A.5 Planning for complex tasks

```
631 ## Complex Task Planning
632
633 For 3+ files, cross-module, or architectural work:
634
635 1. List files to change and why.
636 1. State exact intended change per file.
637 1. Identify dependencies and execution order.
638 1. State verification method per change.
639
640 Skip for simple single-file tasks.
641
```

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

645 A.6 Testing instructions

```
646 ## Testing
647
648 - Run lint/typecheckers; fix all errors. Achieve 100% branch coverage. Every error, including pre-existing
649   ones, is yours---don't skip.
650 - NO mocks, patches, fakes, or test doubles. Write integration/e2e tests. Each test independent, verifying
651   actual behavior.
652 - **Only run impacted tests after modifications.**
653 - To confirm races: add random sleep (<0.1s) before racing statements.
654
```

656 The most opinionated rule is the no-mocks requirement. Mock tests verify that code calls certain
657 methods in a certain order—they test the implementation, not the behavior. A test suite built on

658 mocks can pass with flying colors while the system is fundamentally broken. Integration tests that
659 exercise actual behavior provide genuine confidence that the system works. The clause “Every error,
660 including pre-existing ones, is yours—don’t skip” makes the agent responsible for the entire codebase
661 health, not just the delta it introduced.

662 **A.7 Self-improvement loop**

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

```
664 ## Self-Improvement Loop
665
666 Read PWD/USER_PREFS.md at task start. Update with user preferences/invariants (no code snippets/symbols;
667 skip one-off tasks). Remove conflicting old entries carefully and thoroughly.
668
```

670 This mechanism allows the agent to accumulate project knowledge over time without requiring the
671 user to repeat themselves. The conflict-resolution clause mandates that the agent actively resolves
672 contradictions when updating the file. The prohibition on code snippets and symbols keeps the
673 file compact and robust to code changes; the one-off task exclusion prevents preference drift from
674 idiosyncratic details.

675 **A.8 Pre-finish verification**

```
676 ## Pre-Finish Verification
677
678 Before finish(success=True):
679
680 1. Re-read and verify every modified file.
681 1. Run required checks (lint, typecheck, tests); fix failures.
682 1. Check each user requirement against delivery.
683 1. If any check fails, keep working.
684 1. After 3 failed retries of same fix, rethink from scratch.
685
```

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

689 **A.9 File browsing protocol**

```
690 ## File Browsing
691
692 Collect info and code snippets in PWD/tmp/file-information-{unique_id}.md without overthinking, then
693 review and think deeply.
694
```

696 This instruction applies the same two-phase collect-then-synthesize discipline used for web research
697 (Section 4.2) to the local file system. By writing a structured summary of each file’s relevant
698 information into a temporary markdown file, the agent externalizes its understanding into a compact
699 artifact that is typically much smaller than the raw source files, freeing up context window capacity
700 for subsequent reasoning and editing.

701 **B Painless software engineering (continued)**

```
702 ## Desktop Apps
703
704 Use screenshots, keyboard, and mouse. Don’t launch VS Code or its extensions.
705
```

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

709 **Step 3: Investigating unexpected post-merge state.** After testing the simplified flow, the developer
710 notices that files appear in the Source Control panel of VS Code on the original branch after pressing
711 “Commit and Merge” and asks:

712 After the user presses "Commit and Merge", why do the modified files show up as
713 committed in the original branch?
714

716 The agent traces the exact execution path through `squash_merge_branch()` and discovers that
717 the implementation deliberately *unstages* the squash-merged changes via `git reset HEAD` so they
718 appear as uncommitted working-tree modifications in VS Code’s Source Control panel. The design
719 intention was to let the user review the diff before committing manually. The agent reports this
720 finding along with the relevant code snippet and a verified end state showing no new commits on the
721 branch.

722 **Step 4: Directing a design change in one sentence.** The developer realizes that code review
723 already happened while the worktree was active and decides the unstage step is unnecessary:

724 The review is already happening in the worktree branch. You don’t need the user
725 to review the modified and new files in the original branch. Fix it.
726

728 The agent replaces the `git reset HEAD` call in `squash_merge_branch()` with a conditional `git`
729 `commit -no-edit` that uses the auto-generated squash message. It adds a `git diff -cached`
730 `-quiet` guard for the edge case where the merge produces no changes. One existing test
731 (`test_merge_leaves_changes_uncommitted`) is renamed to `test_merge_commits_changes`
732 and its assertions are updated to verify a clean working tree. All 104 worktree tests pass.

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?

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

998 Justification: All tools, benchmarks, and models used are cited: Terminal-Bench 2.0, Harbor,

999 Claude Opus 4.6, Playwright, VS Code, and all related works. The system is open-source.

1000 Guidelines:

- 1001 • The answer [\[N/A\]](#) means that the paper does not use existing assets.
- 1002 • The authors should cite the original paper that produced the code package or dataset.
- 1003 • The authors should state which version of the asset is used and, if possible, include a
- 1004 URL.
- 1005 • The name of the license (e.g., CC-BY 4.0) should be included for each asset.
- 1006 • For scraped data from a particular source (e.g., website), the copyright and terms of
- 1007 service of that source should be provided.
- 1008 • If assets are released, the license, copyright information, and terms of use in the
- 1009 package should be provided. For popular datasets, paperswithcode.com/datasets
- 1010 has curated licenses for some datasets. Their licensing guide can help determine the
- 1011 license of a dataset.
- 1012 • For existing datasets that are re-packaged, both the original license and the license of
- 1013 the derived asset (if it has changed) should be provided.
- 1014 • If this information is not available online, the authors are encouraged to reach out to
- 1015 the asset’s creators.

1016 **13. New assets**

1017 Question: Are new assets introduced in the paper well documented and is the documentation

1018 provided alongside the assets?

1019 Answer: [\[N/A\]](#)

1020 Justification: No new datasets or pretrained models are released. The open-source code will

1021 be documented upon release.

1022 Guidelines:

- 1023 • The answer [\[N/A\]](#) means that the paper does not release new assets.
- 1024 • Researchers should communicate the details of the dataset/code/model as part of their
- 1025 submissions via structured templates. This includes details about training, license,
- 1026 limitations, etc.
- 1027 • The paper should discuss whether and how consent was obtained from people whose
- 1028 asset is used.
- 1029 • At submission time, remember to anonymize your assets (if applicable). You can either
- 1030 create an anonymized URL or include an anonymized zip file.

1031 **14. Crowdsourcing and research with human subjects**

1032 Question: For crowdsourcing experiments and research with human subjects, does the paper

1033 include the full text of instructions given to participants and screenshots, if applicable, as

1034 well as details about compensation (if any)?

1035 Answer: [\[N/A\]](#)

1036 Justification: The paper does not involve crowdsourcing or research with human subjects.

1037 Guidelines:

- 1038 • The answer [\[N/A\]](#) means that the paper does not involve crowdsourcing nor research
- 1039 with human subjects.
- 1040 • Including this information in the supplemental material is fine, but if the main contribu-
- 1041 tion of the paper involves human subjects, then as much detail as possible should be
- 1042 included in the main paper.
- 1043 • According to the NeurIPS Code of Ethics, workers involved in data collection, curation,
- 1044 or other labor should be paid at least the minimum wage in the country of the data
- 1045 collector.

1046 **15. Institutional review board (IRB) approvals or equivalent for research with human**

1047 **subjects**

1048 Question: Does the paper describe potential risks incurred by study participants, whether
 1049 such risks were disclosed to the subjects, and whether Institutional Review Board (IRB)
 1050 approvals (or an equivalent approval/review based on the requirements of your country or
 1051 institution) were obtained?

1052 Answer: [N/A]

1053 Justification: The paper does not involve human subjects research.

1054 Guidelines:

- 1055 • The answer [N/A] means that the paper does not involve crowdsourcing nor research
 1056 with human subjects.
- 1057 • Depending on the country in which research is conducted, IRB approval (or equivalent)
 1058 may be required for any human subjects research. If you obtained IRB approval, you
 1059 should clearly state this in the paper.
- 1060 • We recognize that the procedures for this may vary significantly between institutions
 1061 and locations, and we expect authors to adhere to the NeurIPS Code of Ethics and the
 1062 guidelines for their institution.
- 1063 • For initial submissions, do not include any information that would break anonymity (if
 1064 applicable), such as the institution conducting the review.

1065 **16. Declaration of LLM usage**

1066 Question: Does the paper describe the usage of LLMs if it is an important, original, or
 1067 non-standard component of the core methods in this research? Note that if the LLM is used
 1068 only for writing, editing, or formatting purposes and does *not* impact the core methodology,
 1069 scientific rigor, or originality of the research, declaration is not required.

1070 Answer: [Yes]

1071 Justification: The entire system is built around LLM usage. Section 2 describes how LLMs
 1072 are used as the core reasoning engine via native function calling (Section 2.1). The paper
 1073 also discloses that the system was built using itself (Section 1).

1074 Guidelines:

- 1075 • The answer [N/A] means that the core method development in this research does not
 1076 involve LLMs as any important, original, or non-standard components.
- 1077 • Please refer to our LLM policy in the NeurIPS handbook for what should or should not
 1078 be described.