
KS Assistant: A Simple AI Agent for Software Engineering

Anonymous Author(s)

Affiliation

Address

email

Abstract

1 Large language models can generate code and call tools effectively, yet deploying
2 them as practical software engineering assistants still reveals persistent gaps: finite
3 context windows, errors that can derail entire sessions, agents that get stuck in dead
4 ends, low-quality output, and changes that are difficult to review or revert.

5 We present **KS Assistant**, a software engineering assistant and integrated develop-
6 ment environment (IDE) built on top of the **KS Agent Framework**, a simple AI
7 agent framework of roughly 1,900 lines of code for the core agents. The framework
8 addresses these gaps through a structured system prompt and a five-layer agent
9 hierarchy in which each layer adds one concern: budget-tracked ReAct execution,
10 automatic continuation across sub-sessions via summarization, coding and browser
11 tools with parallel sub-agents, persistent multi-turn chat with history recall, and
12 git worktree isolation so every task runs on its own branch. Both KS Assistant
13 and the KS Agent Framework encode disciplined software engineering practices
14 directly into the agent’s system prompt, including file review, test-first bug fixing,
15 and pre-finish verification.

16 We implemented KS Assistant as a local Visual Studio Code extension that supports
17 browser automation, multimodal input, Docker containers, and web/mobile access.
18 The system prioritizes verification over latency by giving the model time to check
19 its own output with linters, type checkers, and tests before reporting success. The
20 codebase was developed with assistance from KS Assistant over four months,
21 providing a longitudinal self-hosting case study. On Terminal-Bench 2.0, KS
22 Assistant achieves a 62.2% overall pass rate with Claude Opus 4.6 [Anthropic,
23 2026a]; for context, the public leaderboard reports Claude Code at 58% and Cursor
24 Composer 2 at 61.7%. These results are achieved without benchmark-specific
25 prompt tuning or model fine-tuning.

1 Introduction

27 Modern large language models (LLMs), such as Claude Opus 4.7 [Anthropic, 2026b], GPT 5.5 [Ope-
28 nAI, 2026], and Gemini 3.1 [Google DeepMind, 2026], can generate code, reason about software
29 architecture, and use developer tools [Chen et al., 2021, Rozière et al., 2023]. A growing body of
30 work has explored how to harness these capabilities for autonomous software engineering, from
31 single-session agents that resolve GitHub issues [Yang et al., 2024, Wang et al., 2024] to industrial
32 products marketed as AI software and general assistants [GitHub, 2021, Cursor, 2024, Cognition
33 Labs, 2024, Anthropic, 2025, OpenAI, 2025a, OpenClaw AI, 2025]. Yet using an LLM as a practical
34 software engineering assistant still exposes several persistent gaps: context windows are finite, errors
35 can derail an entire session, agents get stuck in dead ends, models generate low-quality output, and
36 generated changes are difficult to review or revert once applied to a live codebase.

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

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

47 We implemented KS Assistant as a Visual Studio Code extension that runs locally. It includes browser
48 support (using open-source Chromium and Playwright), multimodal support, Docker container
49 support, and a mobile/web app. The system is intended to be released as open-source software; the
50 repository link is withheld to respect double-blind review.

51 KS Assistant has been used to develop its own codebase. The KS Agent framework, the agent
52 layers, the VS Code extension, and the system prompt were developed with KS Assistant, which
53 operates in its own repository. This self-hosting process served as a continuous stress test: when
54 the agent introduced a bug that impaired its functionality, the developers asked the agent to analyze
55 the trajectory and code and propose a fix. The five core agent classes remain compact: the KS
56 Agent comprises 413 lines, the Relentless Agent 321 lines, the Tool Agent 322 lines, the Chat Agent
57 125 lines, and the Worktree Agent 704 lines—a total of roughly 1,885 lines of code (excluding empty
58 lines and comments).

59 We prioritize output quality over speed. Using a weaker or cheaper model can force the developer
60 to discard the agent’s work and start over, increasing the total cost. Conversely, giving a frontier
61 model time to validate its own output—running linters, type checkers, and tests before declaring
62 success—can reduce the number of low-quality changes. We expect token costs and inference
63 latencies to continue to fall [Gao et al., 2025], making this verification-first posture increasingly
64 practical.

65 We evaluate on Terminal-Bench 2.0 and achieve a 62.2% overall pass rate using Claude Opus 4.6 [An-
66 thropic, 2026a]; for context, the public leaderboard reports Claude Code at 58% and Cursor Com-
67 poser 2 at 61.7% [Cursor Research, 2026] on the same benchmark (Section 3). These results are
68 achieved without benchmark-specific prompt tuning or model fine-tuning. The results suggest that
69 a simple layered framework, even without trajectory compaction or asynchronous multi-agent or-
70 chestration, can be competitive on this benchmark. The self-hosting case study further suggests that
71 conventional software engineering practices can be useful when expressed as concrete instructions
72 for LLM agents.

73 2 Agent architecture

74 The KS Agent Framework was originally built to rapidly prototype and experiment with prompt
75 optimization techniques [Agrawal et al., 2026] and evolutionary algorithms for algorithmic optimiza-
76 tion [Novikov et al., 2025, Algorithmic Superintelligence, 2025]. The emphasis on simplicity made
77 the framework easier for coding agents to inspect and modify. Eventually, no prompt optimization
78 techniques were used when creating the system prompt for KS Assistant; it was hand-tuned based
79 on long-term experience with the agent and its behavior. The framework uses five agent layers
80 combining composition and inheritance. Each layer delegates upward for concerns it does not own.

81 2.1 KS Agent

82 The KS Agent is the innermost execution unit implementing a standard ReAct loop [Yao et al.,
83 2023b].

```

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 models make with 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: the per-agent budget, the global budget, and the 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* improved coherence across sub-sessions in our development use. 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.

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

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

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

137 2.3 Tool Agent

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

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

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

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

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

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

155 2.4 Chat Agent

156 The Chat Agent adds multi-turn conversation persistence.

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

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

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

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

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

172 2.5 Worktree Agent

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

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

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

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

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

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

187 3 Evaluation on Terminal-Bench 2.0

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

195 3.1 Setup

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

200 3.2 Aggregate results

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

202 The 62.2% overall pass rate is close to or above two public leaderboard entries: at the time of
203 writing, Claude Code (also Opus 4.6) scores approximately 58% on the Terminal-Bench 2.0 leader-
204 board, and Cursor’s Composer 2—a custom fine-tuned model trained with large-scale reinforcement
205 learning [Cursor Research, 2026]—achieves 61.7%. Because we did not run these systems under a
206 controlled protocol, these numbers should be interpreted as leaderboard context rather than a causal
207 comparison.

208 3.3 Task-level breakdown

209 **Consistently solved tasks (39 of 89).** These include cryptanalysis (FEAL differential), game-playing
210 (chess best move), git operations (leak recovery), server configuration (gRPC key-value store, PyPI
211 server, NGINX logging), data processing (resharding), formal verification (Coq plus_comm), ML

¹<https://www.tbench.ai/>

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

inference (HuggingFace model serving, LLM batching scheduler), and system emulation (QEMU startup). The breadth of this set indicates that the benchmark successes are not concentrated in a single task category.

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 agents on the Terminal-Bench 2.0 leaderboard. We report the result because the evaluation used the general system prompt and Claude Opus 4.6 without benchmark-specific prompt tuning or model fine-tuning. Leaderboard comparisons should be interpreted cautiously: recent analyses report benchmark-gaming and harness-leakage problems on several agent benchmarks, including Terminal-Bench 2.0 [Stein et al., 2026a,b, 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 an 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 define the expected execution style:

```
# 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.

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

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

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

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

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

265 These two sentences define a behavioral contract used throughout later prompt sections. During
266 development, we observed failures that corresponded to these directives (e.g., stopping after the
267 first error when the persistence instruction was absent), but we have not yet performed a controlled
268 ablation of each phrase.

269 4.2 Web research protocol

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

```
272 ## Web Research (MANDATORY)
273
274 - **Visit  $\geq 30$  websites every search. Hard requirement --- don't stop before 30, or rationalize fewer.**
275 - Procedure:
276   1. Create PWD/tmp/information- $\{unique\_id\}$ .md: '# Web Research --- Websites visited: 0/30'
277   1. Per site, append: '## [N/30] URL' + extracted info. Update the header counter on each visit.
278   1. **Don't proceed until counter  $\geq 30$ .**
279   1. If results dry up, try different queries, synonyms, official docs, GitHub repos/issues, Stack Overflow
280     , blogs, Reddit, papers, API refs.
281   1. After 30, review and think deeply.
282 - Ask the user for login help when needed.
283
```

285 The rationale is a two-phase separation between *collection* and *synthesis*. LLMs tend to anchor on
286 the first few results they encounter, biasing their solutions toward a narrow slice of the design space.
287 By forcing the agent to accumulate a broad set of information into a file *before* reasoning about it, the
288 protocol counteracts anchoring bias and encourages the model to consider diverse approaches. The
289 “visit ≥ 30 websites” threshold is a conservative guardrail: it discourages the agent from stopping the
290 research phase after two or three hits, and the resulting information file serves as an auditable artifact
291 of what the agent considered.

292 This two-phase collect-then-synthesize discipline is also applied to local file browsing (Ap-
293 pendix A.11): the agent writes a structured summary of each file’s relevant information into a
294 temporary markdown file, externalizing its understanding into a compact artifact that persists across
295 context boundaries. Analysis happens in the second phase, when the agent reads its own summary
296 and reasons about the collected information as a whole.

297 5 VS Code extension features

298 We release our system as a VS Code extension and a web app. The extension emphasizes several
299 mechanisms that are less central in many existing AI coding assistants, including persistent preference
300 files, real-time budget accounting, and integrated browser automation.

301 **Cross-session self-improvement.** The extension maintains a `USER_PREFS.md` file that the agent
302 reads at the start of each task and *updates* at the end. When the agent discovers a new preference or
303 project invariant during task execution, it writes it to the file. The agent also resolves conflicts: if a
304 new preference contradicts an existing one, the old entry is removed. Over time, the file accumulates
305 project-specific knowledge that can be reused in later sessions. This mechanism is detailed in
306 Appendix A.8.

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

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

6 Conversational development case study

During development, we used natural-language interaction to inspect and revise nontrivial subsystems. One recurring workflow was to ask KS Assistant for a detailed, step-by-step description of a buggy workflow or algorithm, then issue a follow-up request that revised specific steps in natural language. We illustrate this with a development session drawn from the project’s own history, in which the worktree merge workflow (Section 2.5) was first described and then redesigned 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_Assistant_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_Assistant_agent.py and the extension so that after the agent finishes its task, it simply asks "Commit and Merge" or "Discard"? When "Commit" 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.

We omit the remaining two tasks from the main text for space; they appear in Appendix B.

7 Related work

Code-specialized language models. Code Llama [Rozière et al., 2023] fine-tunes Llama 2 for code generation and infilling. StarCoder [Li et al., 2023] trains on permissively licensed GitHub code with a fill-in-the-middle objective, and DeepSeek-Coder [Guo et al., 2024] uses a 2-trillion-token code-and-language corpus with repository-level context. Frontier systems increasingly target agentic software engineering directly: Claude Opus 4.6/4.7 [Anthropic, 2026a,b], GPT 5.5 [OpenAI, 2026], Kimi K2.5 [Kimi Team, 2026], and GLM-5.1 [Z.ai, 2026] emphasize long-horizon coding, tool use, and autonomous task execution. KS Assistant is model-agnostic, so these improvements enter through the backend model rather than through changes to the agent architecture.

Code generation agents and IDEs. SWE-Agent [Yang et al., 2024] and OpenHands [Wang et al., 2024] provide LLM agents for repository-level software tasks such as resolving GitHub issues, typically within one execution session. Agentless [Xia et al., 2024] shows that a simple localize-then-repair pipeline can compete with more autonomous loops on SWE-bench. Devin [Cognition Labs, 2024], Claude Code [Anthropic, 2025], Codex [OpenAI, 2025a], Aider [Gauthier, 2023], GitHub Copilot [GitHub, 2021], Cursor [Cursor, 2024], and Windsurf [Codeium, 2024] expose different combinations of shell, editor, browser, cloud sandbox, and git integration. Cursor Composer 2 [Cursor Research, 2026] further couples an IDE workflow with a reinforcement-learning-trained coding model. KS Assistant differs by making continuation, budget accounting, and branch-per-task worktree isolation first-class layers rather than product-specific behaviors.

Reasoning, tool use, and multi-agent orchestration. ReAct [Yao et al., 2023b], chain-of-thought prompting [Wei et al., 2022], Tree of Thoughts [Yao et al., 2023a], and Reflexion [Shinn et al., 2023] structure reasoning, search, and verbal self-critique. Our Relentless Agent borrows the idea of feeding natural-language artifacts into later attempts, but uses chronological summaries to *continue* a partially completed task after context or step exhaustion, not merely to retry from scratch. ChatDev [Qian et al., 2024], MetaGPT [Hong et al., 2024], and AutoGen [Wu et al., 2023] model development as conversations among role-specialized agents. We instead keep a single primary agent with broad tools and optional parallel sub-agents for independent research or summarization subtasks, avoiding the coordination overhead of always-on role play.

Agent frameworks and context engineering. LangChain [LangChain, 2022], DSPy [Khattab et al., 2024], CrewAI [CrewAI, Inc., 2024], smolagents [Roucher et al., 2025], the OpenAI Agents SDK [OpenAI, 2025b], and Google’s ADK [Google, 2025] offer general agent abstractions such as tool registries, handoffs, guardrails, tracing, and prompt optimization. They are broad infrastructure libraries; KS Assistant is a narrow software-engineering system in which each layer owns one operational concern: budget tracking, continuation, tools, chat memory, or git isolation. Recent work on agentic software engineering, autonomous coding agents, agentic programming, persistent context files, and context engineering [Hassan et al., 2025, Li et al., 2025, Wang et al., 2025, Chatlatanagulchai et al., 2025, Mohsenimofidi et al., 2025] argues that reliable agents need curated task context and controllability. Our system prompt, `USER_PREFS.md`, and per-repository override files instantiate this pattern in a lightweight form.

Benchmarks, self-improvement, and test-time compute. HumanEval [Chen et al., 2021], SWE-bench [Jimenez et al., 2024], LiveCodeBench [Jain et al., 2024], and SWE-bench Pro [Deng et al., 2025] evaluate coding at increasing levels of repository and temporal complexity; Prathifkumar et al. [2025] warns that benchmark contamination can inflate apparent progress. Terminal-Bench 2.0 complements these benchmarks by stressing terminal operation, system configuration, long builds, and tool use. Robeyns et al. [2025] studies agents that improve their own scaffolding; KS Assistant uses a lighter self-improvement loop that accumulates preferences and project invariants across sessions. Gao et al. [2025] studies test-time compute scaling for software engineering, while Get Shit Done [TÂCHES, 2025] uses phased planning documents to fight context rot; both motivate our emphasis on verification, explicit handoff summaries, and disciplined pre-finish checks.

8 Conclusion

KS Assistant shows that a simple layered agent and disciplined system prompt can address practical failure modes in LLM-based software development. On Terminal-Bench 2.0, it reaches 62.2% (277/445) with Claude Opus 4.6 [Anthropic, 2026a], without benchmark-specific tuning, fine-tuning, or reinforcement learning. The result is competitive with two public leaderboard entries, though not a controlled head-to-head comparison. The self-hosting case study suggests that conventional software engineering practices can translate into useful agent instructions.

Limitations. The evaluation covers one frontier model and one benchmark; the 9 hard-skipped tasks count as failures, but reflect manual infeasibility judgments. Leaderboard comparisons are contextual. Terminal-Bench underrepresents review quality, long-project merge conflicts, privacy-sensitive repositories, GUI-heavy workflows, and multi-developer coordination. Summaries can lose details; worktree isolation requires git and adds merge overhead; non-Git projects fall back to direct execution. Verification-first execution costs more tokens and wall-clock time and depends on closed frontier models.

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 Leelaprute, 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.

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

A Full system prompt details

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 an engineering discipline. We describe its key sections.

A.1 Execution Mindset

The prompt opens with a directive that sets the tone:

```
# 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.

“BE CALM.” The complement of relentlessness. An overly aggressive agent that panics on encountering an error may thrash between approaches without deliberation. This directive encourages the model to analyze errors methodically before reacting.

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

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

“CHECK FACTS.” A more specific version of “be accurate” for information collected by KS Assistant uses web tools.

“NO AI SLOP.” “AI slop” refers to the low-quality, generic, hedging text that LLMs produce when they lack confidence: filler phrases like “certainly!”, unnecessary caveats, and boilerplate explanations that the user did not ask for. This directive discourages such output and encourages the model to be concise and substantive.

A.2 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 tool rule addresses a specific failure mode:

“PWD = current working directory.” The acronym “PWD” appears throughout the prompt and in user task descriptions (e.g., “edit PWD/src/main.py”). Without this definition, the model might interpret PWD as the Unix environment variable \$PWD and attempt to expand it, or misinterpret it as a literal directory name. The explicit definition ensures consistent interpretation.

596 **“Write() for new files; Edit() for small changes.”** Without this distinction, the model may use
597 Write() to overwrite an existing file with a slightly modified version, losing content it forgot to
598 include. By reserving Write() for new files and requiring Edit() for modifications, the instruction
599 ensures that changes are surgical and that unchanged portions of a file are never at risk.

600 **Bash timeout guidance.** LLMs frequently launch shell commands without considering their runtime.
601 A compilation or test suite that takes five minutes will time out at the default 30-second shell timeout
602 in most agent frameworks, causing spurious failures. The instruction to use 300 seconds as the
603 default, retry with higher timeouts on timeouts, and run long-running commands in the background
604 with output redirected to a file provides a mechanical protocol that handles common cases without
605 requiring the model to estimate runtime from first principles.

606 **“Use go_to_url() for browser. Search the internet extensively.”** The agent has access to multiple
607 tools that could plausibly interact with the web (shell-based curl, a Python program, the browser
608 tool, etc.). The first clause eliminates ambiguity by specifying which tool to use for browser-
609 based interactions. The second clause encourages the agent to proactively search the web when
610 it encounters unfamiliar APIs, libraries, or error messages, rather than relying on potentially stale
611 parametric memory.

612 **“User only sees the finish() summary.”** Users can get lost in the detailed trajectory generated by KS
613 Assistant in the chat window. Often users want to see only the final summary returned by the finish
614 tool, not the intermediate chain-of-thought or tool calls. Without this instruction, the model may
615 “tell” the user something in an intermediate message and then assume the user has seen it, leading to
616 confusion when the user asks for information the model believes it already provided. The companion
617 clause “Never include meta-descriptions” prevents the model from returning vague summaries like
618 “Fixed the bug in Y” instead of showing the actual fix; it forces the model to include concrete details,
619 results, and outputs in the summary.

620 **“Read large files in chunks. Temp files in PWD/tmp; clean up after.”** Reading a 10,000-line
621 file in a single tool call consumes a large fraction of the context window. By instructing the model
622 to read files in chunks, the prompt prevents context window exhaustion caused by a single-file
623 read, preserving capacity for the rest of the task. Without the temporary-file directive, the model
624 creates temporary files in unpredictable locations (the system /tmp, the home directory, or scattered
625 throughout the project). Centralizing temporary files in a known directory makes cleanup predictable,
626 and the explicit cleanup directive prevents the project tree from being polluted with stale artifacts
627 after the task completes.

628 **“Use ULTRA thinking ALWAYS.”** This instruction activates the model’s extended reasoning mode
629 (also known as “thinking” or “chain-of-thought” mode), in which the model performs additional
630 internal deliberation before producing a response. Extended reasoning is particularly valuable for
631 complex, multi-step tasks in which the model must plan before acting.

632 **“If running out of context/steps, don’t rush—call finish(is_continue=true).”** When the context
633 window is nearly full, LLMs exhibit a “rush to finish” behavior: they skip verification steps, make
634 hasty edits, and call finish with an incomplete result. This instruction redirects that urgency into
635 the continuation protocol (Section 2.2), ensuring that a clean handoff to a new sub-session produces
636 better results than a frantic attempt to squeeze everything into the remaining tokens.

637 A.3 Pre-flight Checks

638 Before modifying any file, we instruct the agent to read it first.

```
639 ## Pre-flight Checks
640
641 - Read every file before modifying it. Read relevant sources if the task depends on existing architecture.
642 - If referenced files/commands/config don't exist, stop and ask or report---don't guess.
643 - **When fixing bugs/issues/races: write tests to confirm first, then fix.**
```

646 Each pre-flight check targets a specific category of avoidable error:

647 **“Read every file before modifying it.”** The most common source of agent-introduced bugs is
648 modifying a file based on an incorrect assumption about its current contents. The model may
649 “remember” an older version of the file from its training data, or it may extrapolate from a partial
650 reading. By requiring a fresh read immediately before any edit, the instruction ensures that the model

operates on the file’s current state rather than a stale mental model. The companion clause “Read relevant sources if the task depends on existing architecture” extends this rule from individual files to architectural context. A task like “add a caching layer to the database module” requires understanding not just the file to be modified, but also how callers interact with it, what interfaces it provides, and what invariants it assumes. The instruction prevents the model from jumping straight to code generation without understanding the broader context.

“If referenced files/commands/config don’t exist, stop and ask or report—don’t guess.” LLMs have a strong tendency to confabulate: when asked to modify a file that does not exist, the model will often proceed as if it does, producing edits against phantom content. This instruction converts a silent failure (incorrect edits applied to a nonexistent file, which silently creates it) into an explicit clarification request.

“When fixing bugs/issues/races: write tests to confirm first, then fix.” This instruction mandates a test-first discipline for bug fixes. The motivation is two-fold: first, a test that reproduces the bug provides concrete verification that the fix is correct (the test should pass after the fix and fail before it). Second, writing the test forces the model to understand the bug precisely before attempting a fix, reducing the risk of an ad hoc patch that addresses a symptom rather than the root cause.

A.4 Code Style Guidelines

The prompt encodes a minimalist code philosophy:

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

Each guideline addresses a specific anti-pattern commonly exhibited by LLM-generated code:

“Simple, clean, readable code with minimal indirection. Organize in multiple files by functionality.” LLMs tend to over-engineer solutions, introducing unnecessary abstractions, helper classes, and levels of indirection. Simple code is easier to review, test, and maintain. LLMs also often pile new code onto whichever file is currently being edited, producing 2,000-line modules that conflate unrelated concerns. This directive nudges the model toward a modular layout in which each file has a single, coherent responsibility.

“Avoid unnecessary attributes, locals, config vars, tight coupling, and attribute redirections.” LLMs frequently introduce intermediate variables that serve no purpose—for example, assigning a return value to a local variable only to immediately return it on the next line, or storing a constant in a configuration file when it is used in exactly one place. When the model adds a feature that touches multiple files, it may introduce imports, shared global state, or cross-module function calls that create tight coupling. An attribute redirection occurs when an object stores a reference to another object solely to forward method calls to it—for example, `self.x = other.x` at construction time, creating two paths to the same value. This single consolidated rule addresses all of these anti-patterns.

“DO NOT USE CLOSURES. No redundant abstractions or duplicate code.” LLMs reach for closures whenever a small piece of state needs to be carried alongside a function—producing nested defs that capture mutable variables from the enclosing scope. Such closures are difficult to test in isolation, opaque to type checkers, and a frequent source of subtle bugs due to the late binding of captured variables. The blanket prohibition steers the model toward explicit data structures (plain functions with arguments, classes with attributes), which are easier to reason about, easier to test, and play well with our no-mocks testing discipline. LLMs also sometimes create utility functions or classes that duplicate existing functionality; this instruction reminds the model to check for existing implementations before creating new ones.

“Public methods MUST have full documentation.” While the prompt generally discourages unnecessary documentation (see the last item), public methods are the API surface that other developers and modules depend on. Documentation on public methods is not optional—it specifies the contract.

706 **“Fix root causes, not symptoms. Think first: is the code simple, elegant, general, minimal?”**
707 LLMs frequently apply symptom-level fixes: adding a null check where the real problem is that a
708 variable should never be null, or catching an exception where the real problem is that the caller passes
709 invalid arguments. This instruction forces the model to trace the causal chain to the root and fix it
710 there. The companion metacognitive instruction asks the model to pause and evaluate its plan before
711 committing to an implementation, spending more inference-time compute on design and reducing the
712 likelihood of producing an unnecessarily complex first draft.

713 **“Don’t write documentation unless the task requires it.”** Claude Opus 4.6 tends to generate many
714 documentation files. This instruction prevents the behavior.

715 A.5 Deep Work Rules

```
716 ## Deep Work
717
718 - For "align"/"match"/"make consistent": read the target state before editing. Never edit from vague
719   references.
720 - Use concrete values, not indirections (read Y first, then write specific values into X).
721 - List concrete planned changes before executing multi-part work.
722 - Every meaningful change needs a concrete verification method (test, grep, CLI).
```

725 The deep work rules address a failure mode where the model interprets an instruction loosely and
726 makes changes that are directionally correct but concretely wrong:

727 **“For ‘align’/‘match’/‘make consistent’: read the target state before editing.”** When a user says
728 “make file A consistent with file B,” the model often reads file A, infers what file B probably contains,
729 and edits A based on that inference—without ever reading B. This instruction mandates reading the
730 target first, ensuring that the alignment is based on concrete facts rather than assumptions.

731 **“Use concrete values, not indirections (read Y first, then write specific values into X).”** A related
732 failure mode occurs when the model’s plan says “update X to match Y” but the model never resolves
733 what Y actually is. The instruction requires the model to first read Y, extract the specific values, and
734 then write those values into X. This eliminates a class of errors where the model’s mental model of Y
735 differs from reality.

736 **“List concrete planned changes before executing multi-part work.”** When a task requires changes
737 to multiple files, executing them one at a time without a plan leads to inconsistencies: the model may
738 change a function signature in one file but forget to update a caller in another. Listing all planned
739 changes before executing any of them forces the model to consider the full scope of the change and
740 identify dependencies.

741 **“Every meaningful change needs a concrete verification method.”** A change without a verification
742 method is a change that cannot be confirmed to work. This instruction requires the model to pair each
743 change with a specific check—a test, a grep for the expected pattern, a CLI command that exercises
744 the changed behavior—ensuring that the change can be validated programmatically rather than by
745 visual inspection of a diff.

746 A.6 Planning for Complex Tasks

747 The planning instructions use a complexity threshold—three or more files, cross-module changes, or
748 architectural work—to decide when formal planning is required:

```
749 ## Complex Task Planning
750
751 For 3+ files, cross-module, or architectural work:
752
753 1. List files to change and why.
754 1. State exact intended change per file.
755 1. Identify dependencies and execution order.
756 1. State verification method per change.
757
758 Skip for simple single-file tasks.
```

761 Each planning step targets a specific failure mode:

762 **“List files to change and why.”** This forces the model to enumerate the full blast radius of a change
763 before touching any file. Without this step, the model often discovers mid-task that additional files
764 need changes, leading to incomplete or inconsistent modifications.

765 **“State exact intended change per file.”** Listing files alone is insufficient; the model must also
766 articulate *what* will change in each file. This converts a vague plan (“update the database module”)
767 into a concrete specification (“add a `cache_ttl` parameter to `DatabaseClient.__init__`, modify
768 the query method to check the cache before hitting the database, add a cache invalidation method).

769 **“Identify dependencies and execution order.”** Some changes must precede others: a new utility
770 function must be written before callers can import it, a migration must run before code that depends
771 on the new schema. Identifying these dependencies prevents the model from applying changes in an
772 order that produces intermediate states where the code does not compile, or tests do not pass.

773 **“State verification method per change.”** The verification requirement from the Deep Work section
774 is reinforced here at the planning stage, ensuring that verification is planned alongside the changes
775 rather than treated as an afterthought.

776 The escape clause—“Skip for simple single-file tasks”—avoids the overhead of planning trivial
777 changes. Requiring a formal plan for a one-line typo fix would waste tokens and slow down the agent
778 without any compensating benefit.

779 A.7 Testing Instructions

780 The testing section is perhaps the most opinionated:

```
781 ## Testing
782
783 - Run lint/typecheckers; fix all errors. Achieve 100% branch coverage. Every error, including pre-existing
784   ones, is yours---don't skip.
785 - NO mocks, patches, fakes, or test doubles. Write integration/e2e tests. Each test independent, verifying
786   actual behavior.
787 - **Only run impacted tests after modifications.**
788 - To confirm races: add random sleep (<0.1s) before racing statements.
789
```

791 Each testing instruction addresses a specific concern:

792 **“Run lint/typecheckers; fix all errors. Achieve 100% branch coverage.”** Before committing any
793 change, the agent must ensure it does not introduce lint violations or type errors. This catches a
794 broad class of issues—unused imports, type mismatches, style violations—that would otherwise
795 accumulate across tasks. Full branch coverage ensures that every conditional path in the code under
796 test has been exercised. LLMs tend to write happy-path tests that cover the main code path but ignore
797 error handling, edge cases, and early-return branches. The 100% target pushes the model to write
798 tests for every branch, including error paths and boundary conditions. Such tests help with regression
799 by making failures visible before changes are accepted.

800 **“Every error, including pre-existing ones, is yours—don’t skip.”** Without this clause, the model
801 frequently rationalizes pre-existing lint or type errors as “not my problem” and calls `finish` with
802 a passing result despite a broken build. The instruction makes the agent responsible for the entire
803 codebase health, not just the delta it introduced.

804 **“NO mocks, patches, fakes, or test doubles. Write integration/e2e tests.”** This is the most
805 opinionated rule. Mock tests that verify code calls certain methods in a certain order test the
806 implementation, not the behavior. A test suite built on mocks can pass with flying colors while the
807 system is fundamentally broken, because the mocks hide the real dependencies. Integration tests that
808 exercise actual behavior are more expensive to run but provide genuine confidence that the system
809 works. Moreover, writing integration tests forces the model to think more deeply, often enabling the
810 agent to find deeper bugs in the code. The distinction between unit and integration tests matters: a
811 unit test in isolation may verify that a function produces the right output for a given input, but an
812 integration or end-to-end test verifies that the function works correctly within the larger system—with
813 real file I/O, real database connections, and real inter-module interactions.

814 **“Each test independent, verifying actual behavior.”** Test independence means that running tests in
815 any order produces the same results. Tests that depend on shared state or execution order are brittle

816 and difficult to debug when they fail. “Verifying actual behavior” reiterates that tests should assert on
817 observable outcomes (return values, side effects, system state) rather than implementation details.

818 **“Only run impacted tests after modifications.”** Running the full test suite after every small change
819 is wasteful when only a few modules are affected. For a large project, a full test run may take minutes,
820 and doing it after every edit adds up to significant wasted time and compute. This instruction directs
821 the model to identify which tests are affected by its changes and run only those, improving iteration
822 speed.

823 **“To confirm races: add random sleep (<0.1s) before racing statements.”** Race conditions are noto-
824 riously difficult to reproduce because they depend on precise timing. By inserting small sleep delays
825 at strategic points, the model can widen the race window and make the bug manifest deterministically
826 during testing. The 0.1-second upper bound keeps the test fast while still being sufficient to expose
827 most races.

828 A.8 Self-Improvement Loop

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

```
830 ## Self-Improvement Loop
831
832
833 Read PWD/USER_PREFS.md at task start. Update with user preferences/invariants (no code snippets/symbols;
834 skip one-off tasks). Remove conflicting old entries carefully and thoroughly.
```

836 Each clause in this compact instruction serves the goal of cross-session learning:

837 **“Read PWD/USER_PREFS.md at task start.”** The preferences file contains invariants learned from
838 previous tasks—coding conventions, project-specific rules, architectural decisions. Reading it at the
839 start of each task ensures that the agent’s behavior is consistent across sessions, even though each
840 session starts with a fresh context window.

841 **“Update with user preferences/invariants.”** After completing a task, the agent may have learned
842 new information about the user’s preferences: a preferred naming convention, a disliked pattern, a
843 project-specific invariant. Writing these to the preferences file makes them available to future sessions.
844 This mechanism allows the agent to accumulate project knowledge over time without requiring the
845 user to repeat themselves.

846 **“No code snippets/symbols; skip one-off tasks.”** Code snippets in the preferences file are fragile:
847 they become stale as the codebase evolves, and they consume tokens that would be better spent
848 on natural-language descriptions of invariants. The companion rule about one-off tasks prevents
849 preference drift in the opposite direction: without it, the agent would record an entry every time it
850 completed any task, gradually polluting the file with idiosyncratic details (a particular file path, a
851 single throwaway experiment) that have no bearing on future work. Together, the two clauses keep
852 the file compact, robust to code changes, and focused on durable invariants.

853 **“Remove conflicting old entries carefully and thoroughly.”** Over time, preferences may become
854 contradictory—for example, an early preference might say “use camelCase for test methods” while a
855 later correction says “use snake_case.” Without explicit conflict resolution, the file would accumulate
856 contradictions, confusing the agent. This instruction mandates that the agent actively resolve conflicts
857 when updating the file to maintain internal consistency. *Note that we do not keep learnings in a*
858 *database or in various folders because it makes the information stale when lots of code changes are*
859 *happening. It is impossible for an agent to eliminate stale information across databases and multiple*
860 *folders.*

861 A.9 Pre-Finish Verification

862 Before declaring a task complete, the agent must pass a structured verification checklist:

```
863 ## Pre-Finish Verification
864
865
866 Before finish(success=True):
867
868 1. Re-read and verify every modified file.
869 1. Run required checks (lint, typecheck, tests); fix failures.
```

```

870 1. Check each user requirement against delivery.
871 1. If any check fails, keep working.
872 1. After 3 failed retries of same fix, rethink from scratch.

```

Each step in this checklist addresses a specific way agents declare premature success:

“Re-read and verify every modified file.” This is the analog of a code review performed by the agent on its own work. The model may have introduced a typo, forgotten to close a bracket, or made an edit that looked correct in the diff but was wrong in the full-file context. Re-reading the file after all edits are complete catches these errors.

“Run required checks (lint, typecheck, tests); fix failures.” This converts the subjective assessment “I think my changes are correct” into an objective, automated verification. If the lint, typecheck, or test suite fails, the agent must fix the failure before declaring success.

“Check each user requirement against delivery.” The model may have completed a task that it *thinks* satisfies the user’s request, but actually misses a requirement. This instruction forces a systematic comparison between the original task description and the delivered result, catching gaps and misinterpretations.

“If any check fails, keep working.” Without this instruction, the model may call `finish(success=True)` even when it knows a check has failed, rationalizing that the failure is “minor” or “unrelated.” The instruction makes the rule absolute: no finishing until all checks pass.

“After 3 failed retries of same fix, rethink from scratch.” LLMs can enter repetitive loops where they apply the same incorrect fix repeatedly, each time hoping for a different result. The three-retry threshold forces the model to break out of such loops by abandoning the current approach and reconsidering the problem from first principles. This is analogous to the debugging heuristic “if you’ve been staring at the same code for twenty minutes, you’re looking in the wrong place.”

A.10 Web Research Protocol

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

```

897 ## Web Research (MANDATORY)
898
899 - **Visit >=30 websites every search. Hard requirement---don't stop before 30 or rationalize fewer.**
900 - Procedure:
901   1. Create PWD/tmp/information-{unique_id}.md: '# Web Research --- Websites visited: 0/30'
902   1. Per site, append: '## [N/30] URL' + extracted info. Update header counter each visit.
903   1. **Don't proceed until counter >=30.**
904   1. If results dry up, try different queries, synonyms, official docs, GitHub repos/issues, Stack Overflow
905     , blogs, Reddit, papers, API refs.
906   1. After 30, review and think deeply.
907 - Ask user for login help when needed.
908

```

The rationale is a two-phase separation between *collection* and *synthesis*. LLMs tend to anchor on the first few results they encounter, which biases their solutions toward a narrow slice of the design space. By forcing the agent to accumulate a broad set of information into a file *before* reasoning about it, the protocol counteracts anchoring bias and encourages the model to consider diverse approaches. The “visit ≥ 30 websites” threshold is a conservative guardrail: it discourages the agent from stopping the research phase after two or three hits, and the resulting information file serves as an auditable artifact of what the agent considered. The structured procedure with a counter header (`# Web Research --- Websites visited: 0/30`) and per-site entries (`## [N/30] URL`) addresses an empirically observed failure mode in which the model claims to have “visited many sites” after only a handful of fetches; a concrete counter forces the model to verify the actual number visited before declaring the collection phase complete. The instruction to try “different queries, synonyms, official docs” when results dry up prevents the agent from giving up prematurely on a narrow set of search terms.

The login instruction addresses a practical obstacle in web research: many websites require authentication before revealing their content. Rather than silently skipping gated pages or hallucinating their contents, we instruct the agent to ask the user for help with login.

925 A.11 File Browsing Protocol

926 When a task requires understanding multiple source files before making changes, the prompt pre-
927 scribes the same two-phase collect-then-synthesize discipline used for web research, but applied to
928 the local file system:

```
929 ## File Browsing
930
931 Collect info and code snippets in PWD/tmp/file-information-{unique_id}.md without overthinking, then
932 review and think deeply.
933
```

935 This instruction addresses a failure mode distinct from the web research case. When an agent must
936 read many project files to understand a codebase before making changes, it tends to read a file, form a
937 hypothesis, and immediately begin editing—anchoring on the first few files it encounters and missing
938 relevant context in files it never opens. Worse, each file read consumes context window tokens; by
939 the time the agent has read enough files to understand the full picture, it may have already spent most
940 of its context window on the raw file contents, leaving little room for reasoning and code generation.

941 The file browsing protocol counteracts both problems. By writing a structured summary of each
942 file’s relevant information into a temporary markdown file, the agent externalizes its understanding
943 into a compact artifact that persists across context boundaries. The instruction to collect “without
944 overthinking” is deliberate: during the collection phase, the agent should extract and record facts
945 (function signatures, class hierarchies, call sites, invariants) rather than analyze or plan. Analysis
946 happens in the second phase, when the agent reads its own summary file and reasons about the
947 collected information as a whole.

948 This two-phase separation provides three benefits. First, it prevents premature commitment: the agent
949 cannot start editing until it has surveyed the relevant files, reducing the risk of changes that are locally
950 correct but globally inconsistent. Second, the summary file is typically much smaller than the raw
951 source files, freeing up context window capacity for subsequent reasoning and editing phases. Third,
952 the summary file serves as an auditable artifact—the developer can inspect it to verify that the agent
953 considered the right files and extracted the right information before making changes.

954 A.12 Desktop Application Control

955 The agent can interact with graphical desktop applications using screenshots, keyboard, and mouse:

```
956 ## Desktop Apps
957
958 Use screenshots, keyboard, and mouse. Don't launch VS Code or its extensions.
959
```

961 This instruction enables the agent to operate GUI applications (Preview, browsers, graphical diff tools)
962 when command-line alternatives are insufficient. The explicit prohibition on launching VS Code
963 prevents a recursive loop: since the agent runs *inside* a VS Code extension, launching another
964 VS Code instance or modifying extension state from within the agent could corrupt the host session
965 or create deadlocks. Note that modern LLMs support desktop control abilities, and we are merely
966 exploiting them.

967 A.13 Assistant-Specific Overrides

968 A final section provides project-specific instructions that are injected when the agent operates on its
969 own codebase:

```
970 ## Assistant-specific
971
972 - Lint/typecheck/format: 'uv run check --full'. Test: 'uv run pytest -v' (timeout 900s).
973 - **Do NOT install the KS Assistant extension from inside Assistant.**
974 - To open/edit system prompt: hidden for double-blind review
975 - KS Assistant info: hidden for double-blind review
976 - Third-party agents: KS/agents/third_party_agents
977 - Official Claude SKILLS: KS/agents/claude_skills
978 - Authenticate unauthenticated third-party agents; ask user only when a page needs auth.
979 - Read PWD/Assistant.md as overriding instructions.
980
```

982 These overrides serve seven purposes. First, they specify the exact toolchain commands for the
 983 KS project itself (`uv run check --full`, `uv run pytest`), eliminating guesswork about which
 984 linter, formatter, or test runner to use. Second, they prevent dangerous self-modification: just as
 985 a surgeon should not operate on themselves, the agent must not install or reinstall the VS Code
 986 extension it is running inside, since doing so would terminate its own process. Third, they provide
 987 navigation instructions for the agent’s own source: when the user asks to edit the system prompt, the
 988 agent knows the exact file path within the installed VS Code extension directory, avoiding guesswork
 989 about where the file lives. Moreover, it does not create unnecessary UI components in KS Assistant
 990 just to edit the system prompt. Fourth, the agent is given a URL to its own paper’s source as a
 991 source of self-knowledge: when a user asks the agent about its own capabilities, design, or identity, it
 992 can retrieve and read the paper rather than hallucinating an answer. This creates a self-referential
 993 loop in which the documentation describing the agent is also consumed by the agent. Fifth, the
 994 instructions expose a third-party agent integration layer: the agent is told where third-party agents
 995 live (`KS/agents/third_party_agents`). When a third-party agent requires authentication, the
 996 agent handles it autonomously and only prompts the user when a page genuinely requires user
 997 credentials—reducing unnecessary interruptions while maintaining security. Sixth, official Claude
 998 SKILLS [Anthropic, 2025] are bundled in `KS/agents/claude_skills`, giving the agent access
 999 to Anthropic’s curated skill library for common software engineering patterns; by indicating where
 1000 these skills reside, the agent can reference and apply them when relevant tasks arise. Seventh,
 1001 the `Assistant.md` override mechanism allows per-repository instructions to further customize the
 1002 agent’s behavior, forming a hierarchy: general system prompt → Assistant-specific instructions →
 1003 repository-specific `Assistant.md`. This approach avoids hard-coding repository-specific instructions
 1004 in the KS Assistant code. The `Assistant.md` file can include additional markdown files for further
 1005 instructions.

1006 B Conversational development case study (continued)

```

1007 ## Desktop Apps
1008
1009 Use screenshots, keyboard, and mouse. Don't launch VS Code or its extensions.
1010

```

1012 The prohibition on launching VS Code prevents a recursive loop: since the agent runs inside a
 1013 VS Code extension, launching another instance could corrupt the host session or create deadlocks.

1014 **Step 3: Investigating unexpected post-merge state.** After testing the simplified flow, the developer
1015 notices that files appear in the Source Control panel of VS Code on the original branch after pressing
1016 “Commit and Merge” and asks:

1017 After the user presses "Commit and Merge", why do the modified files show up as
1018 committed in the original branch?
1019

1021 The agent traces the exact execution path through `squash_merge_branch()` and discovers that
1022 the implementation deliberately *unstages* the squash-merged changes via `git reset HEAD` so they
1023 appear as uncommitted working-tree modifications in VS Code’s Source Control panel. The design
1024 intention was to let the user review the diff before committing manually. The agent reports this
1025 finding along with the relevant code snippet and a verified end state showing no new commits on the
1026 branch.

1027 **Step 4: Directing a design change in one sentence.** The developer realizes that code review
1028 already happened while the worktree was active and decides the unstage step is unnecessary:

1029 The review is already happening in the worktree branch. You don’t need the user
1030 to review the modified and new files in the original branch. Fix it.
1031

1033 The agent replaces the `git reset HEAD` call in `squash_merge_branch()` with a conditional `git`
1034 `commit -no-edit` that uses the auto-generated squash message. It adds a `git diff -cached`
1035 `-quiet` guard for the edge case where the merge produces no changes. One existing test
1036 (`test_merge_leaves_changes_uncommitted`) is renamed to `test_merge_commits_changes`
1037 and its assertions are updated to verify a clean working tree. All 104 worktree tests pass.

1038 NeurIPS Paper Checklist

1039 1. Claims

1040 Question: Do the main claims made in the abstract and introduction accurately reflect the
1041 paper’s contributions and scope?

1042 Answer: [Yes]

1043 Justification: The abstract and introduction (Section 1) state the contributions—a five-layer
1044 agent architecture, a structured system prompt, a VS Code extension, and a 62.2% pass
1045 rate on Terminal-Bench 2.0—and limit the empirical scope to software engineering tasks
1046 evaluated on this benchmark with Claude Opus 4.6. The paper now treats Claude Code
1047 and Cursor Composer 2 numbers as public leaderboard context rather than a controlled
1048 head-to-head comparison. The self-hosting claim is presented as a development case study
1049 rather than causal evidence of performance.

1050 Guidelines:

- 1051 • The answer [N/A] means that the abstract and introduction do not include the claims
1052 made in the paper.
- 1053 • The abstract and/or introduction should clearly state the claims made, including the
1054 contributions made in the paper and important assumptions and limitations. A [No] or
1055 [N/A] answer to this question will not be perceived well by the reviewers.
- 1056 • The claims made should match theoretical and experimental results, and reflect how
1057 much the results can be expected to generalize to other settings.
- 1058 • It is fine to include aspirational goals as motivation as long as it is clear that these goals
1059 are not attained by the paper.

1060 2. Limitations

1061 Question: Does the paper discuss the limitations of the work performed by the authors?

1062 Answer: [Yes]

1063 Justification: Section 3 discusses consistently failed tasks and their failure categories (graph-
1064 ical/multimedia requirements, resource-intensive builds, niche domain knowledge). The
1065 conclusion acknowledges that results depend on a single frontier model (Claude Opus 4.6)
1066 and a single benchmark.

1067 Guidelines:

- 1068 • The answer [N/A] means that the paper has no limitation while the answer [No] means
1069 that the paper has limitations, but those are not discussed in the paper.
- 1070 • The authors are encouraged to create a separate “Limitations” section in their paper.
- 1071 • The paper should point out any strong assumptions and how robust the results are to
1072 violations of these assumptions (e.g., independence assumptions, noiseless settings,
1073 model well-specification, asymptotic approximations only holding locally). The authors
1074 should reflect on how these assumptions might be violated in practice and what the
1075 implications would be.
- 1076 • The authors should reflect on the scope of the claims made, e.g., if the approach was
1077 only tested on a few datasets or with a few runs. In general, empirical results often
1078 depend on implicit assumptions, which should be articulated.
- 1079 • The authors should reflect on the factors that influence the performance of the approach.
1080 For example, a facial recognition algorithm may perform poorly when image resolution
1081 is low or images are taken in low lighting. Or a speech-to-text system might not be
1082 used reliably to provide closed captions for online lectures because it fails to handle
1083 technical jargon.
- 1084 • The authors should discuss the computational efficiency of the proposed algorithms
1085 and how they scale with dataset size.
- 1086 • If applicable, the authors should discuss possible limitations of their approach to
1087 address problems of privacy and fairness.
- 1088 • While the authors might fear that complete honesty about limitations might be used by
1089 reviewers as grounds for rejection, a worse outcome might be that reviewers discover
1090 limitations that aren’t acknowledged in the paper. The authors should use their best

1091 judgment and recognize that individual actions in favor of transparency play an impor-
1092 tant role in developing norms that preserve the integrity of the community. Reviewers
1093 will be specifically instructed to not penalize honesty concerning limitations.

1094 3. Theory assumptions and proofs

1095 Question: For each theoretical result, does the paper provide the full set of assumptions and
1096 a complete (and correct) proof?

1097 Answer: [N/A]

1098 Justification: The paper does not include theoretical results. It is a systems paper presenting
1099 an architecture and empirical evaluation.

1100 Guidelines:

- 1101 • The answer [N/A] means that the paper does not include theoretical results.
- 1102 • All the theorems, formulas, and proofs in the paper should be numbered and cross-
1103 referenced.
- 1104 • All assumptions should be clearly stated or referenced in the statement of any theorems.
- 1105 • The proofs can either appear in the main paper or the supplemental material, but if
1106 they appear in the supplemental material, the authors are encouraged to provide a short
1107 proof sketch to provide intuition.
- 1108 • Inversely, any informal proof provided in the core of the paper should be complemented
1109 by formal proofs provided in appendix or supplemental material.
- 1110 • Theorems and Lemmas that the proof relies upon should be properly referenced.

1111 4. Experimental result reproducibility

1112 Question: Does the paper fully disclose all the information needed to reproduce the main ex-
1113 perimental results of the paper to the extent that it affects the main claims and/or conclusions
1114 of the paper (regardless of whether the code and data are provided or not)?

1115 Answer: [Yes]

1116 Justification: Section 3 describes the benchmark (Terminal-Bench 2.0), the evaluation frame-
1117 work (Harbor), the model (Claude Opus 4.6), the hardware (2025 MacBook Air M4, 24 GB
1118 RAM), the number of trials (5 per task), and the 9 skipped tasks. The full agent architecture
1119 is described in Section 2 and the system prompt in Section 4 and Appendix A. Because the
1120 code is not included with the submission, reproduction would require reimplementing the
1121 described system or obtaining the released code later.

1122 Guidelines:

- 1123 • The answer [N/A] means that the paper does not include experiments.
- 1124 • If the paper includes experiments, a [No] answer to this question will not be perceived
1125 well by the reviewers: Making the paper reproducible is important, regardless of
1126 whether the code and data are provided or not.
- 1127 • If the contribution is a dataset and/or model, the authors should describe the steps taken
1128 to make their results reproducible or verifiable.
- 1129 • Depending on the contribution, reproducibility can be accomplished in various ways.
1130 For example, if the contribution is a novel architecture, describing the architecture fully
1131 might suffice, or if the contribution is a specific model and empirical evaluation, it may
1132 be necessary to either make it possible for others to replicate the model with the same
1133 dataset, or provide access to the model. In general, releasing code and data is often
1134 one good way to accomplish this, but reproducibility can also be provided via detailed
1135 instructions for how to replicate the results, access to a hosted model (e.g., in the case
1136 of a large language model), releasing of a model checkpoint, or other means that are
1137 appropriate to the research performed.
- 1138 • While NeurIPS does not require releasing code, the conference does require all submis-
1139 sions to provide some reasonable avenue for reproducibility, which may depend on the
1140 nature of the contribution. For example
 - 1141 (a) If the contribution is primarily a new algorithm, the paper should make it clear how
1142 to reproduce that algorithm.
 - 1143 (b) If the contribution is primarily a new model architecture, the paper should describe
1144 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]

Justification: The repository link is withheld to respect double-blind review (stated in Section 1). No anonymized code package or reproduction scripts are currently provided with the submission.

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: [No]

Justification: We report 5 independent trials per task (445 total runs), pass@any (78.7%), pass@all (43.8%), overall pass rate (62.2%), always-pass/always-fail/mixed-result counts, and median/mean cost and duration. However, the paper does not report confidence intervals, error bars, or statistical significance tests, so the checklist answer is No.

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: [No]

Justification: The paper focuses on architecture and benchmark evaluation. It does not include a dedicated broader-impact discussion covering both potential benefits and potential harms, so the checklist answer is No.

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: [No]

Justification: The paper cites the main tools, benchmarks, and models used, including Terminal-Bench 2.0, Harbor, Claude Opus 4.6, Playwright, VS Code, and related work. It does not yet explicitly list the license names or terms of use for each existing asset, so the checklist answer is No.

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: [No]

Justification: The paper introduces a software system, but the anonymized code package and accompanying documentation are not included with the submission. No new datasets or pretrained models are released.

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.

- 1352
- 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.
- 1353
- 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.
- 1354
- 1355
- 1356
- 1357

1358 **15. Institutional review board (IRB) approvals or equivalent for research with human**

1359 **subjects**

1360 Question: Does the paper describe potential risks incurred by study participants, whether

1361 such risks were disclosed to the subjects, and whether Institutional Review Board (IRB)

1362 approvals (or an equivalent approval/review based on the requirements of your country or

1363 institution) were obtained?

1364 Answer: [N/A]

1365 Justification: The paper does not involve human subjects research.

1366 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.

1377 **16. Declaration of LLM usage**

1378 Question: Does the paper describe the usage of LLMs if it is an important, original, or

1379 non-standard component of the core methods in this research? Note that if the LLM is used

1380 only for writing, editing, or formatting purposes and does *not* impact the core methodology,

1381 scientific rigor, or originality of the research, a declaration is not required.

1382 Answer: [Yes]

1383 Justification: The entire system is built around LLM usage. The paper also discloses that the

1384 system was used to develop its own codebase (Section 1). An initial rough draft of the paper

1385 was generated from the code of KS Assistant and its README.md.

1386 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.