

---

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

---

Anonymous Author(s)

Affiliation

Address

email

## Abstract

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

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

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

## 1 Introduction

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

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

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

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

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

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

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

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

## 2 Agent architecture

The KS Agent Framework was originally built to rapidly prototype and experiment with prompt optimization techniques [Agrawal et al., 2026] and evolutionary algorithms for algorithmic optimization [Novikov et al., 2025, Algorithmic Superintelligence, 2025]. The emphasis on simplicity enabled coding agents to write simple, bug-free code. Eventually, no prompt optimization techniques were used when creating the system prompt for KS Assistant; it was hand-tuned based on long-term experience with the agent and its behavior. The framework uses five agent layers combining composition and inheritance. Each layer delegates upward for concerns it does not own.

## 87 2.1 KS Agent

88 The KS Agent is the innermost execution unit implementing a standard ReAct loop [Yao et al.,  
89 2023b].

90

```
from ks.core.ks_agent import KSAgent

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

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

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

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

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

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

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

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

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

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

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

## 109 2.2 Relentless Agent

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

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

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

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

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

## 143 2.3 Tool Agent

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

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

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

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

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

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

## 161 2.4 Chat Agent

162 The Chat Agent adds multi-turn conversation persistence.

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

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

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

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

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

## 178 2.5 Worktree Agent

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

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

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

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

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

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

## 193 3 Evaluation on Terminal-Bench 2.0

194 We evaluate on Terminal-Bench 2.0,<sup>1</sup> a benchmark comprising 89 diverse terminal-based program-  
195 ming tasks, ranging from building legacy compilers and configuring servers to solving cryptanalysis  
196 challenges and training machine-learning models. Each task runs in an isolated Docker container;  
197 a separate verifier automatically judges the result. We use the Harbor<sup>2</sup> framework to orchestrate  
198 execution and Claude Opus 4.6 [Anthropic, 2026a] as the underlying LLM. We do not modify the  
199 general system prompt or inject benchmark-specific instructions. Evaluation was carried out on a  
200 2025 MacBook Air 15” with an M4 processor and 24 GB RAM.

### 201 3.1 Setup

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

### 206 3.2 Aggregate results

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

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

---

<sup>1</sup><https://www.tbench.ai/>

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

### 3.3 Task-level breakdown

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

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

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

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

## 4 The system prompt

The system prompt is a structured document that governs the agent’s behavior across all tasks. It is not a generic instruction to “be helpful” but rather a precise specification of 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 set the tone for the entire session:

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

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

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

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

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

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

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

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

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

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

## 281 4.2 Web research protocol

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

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

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

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

## 309 5 VS Code extension: unique features

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

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

project invariant during task execution, it writes it to the file. The agent also resolves conflicts: if a new preference contradicts an existing one, the old entry is removed. Over time, the file accumulates a curated set of project-specific knowledge, making the agent progressively more effective without any manual configuration. This mechanism is detailed in Appendix A.8.

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

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

## 6 Painless software engineering with KS Assistant

A central claim of our system is that natural-language interaction can replace manual code inspection and ad hoc scripting to understand and evolve nontrivial subsystems. While developing KS Assistant, we found a particular workflow to be useful. In the workflow, we first ask KS Assistant to generate a detailed, step-by-step description of a workflow or algorithm we found buggy, and then we ask KS Assistant to revise some of the buggy steps in natural language. We illustrate this with a real development session drawn from the project’s own history, in which the worktree merge workflow (Section 2.5) was first understood and then redesigned entirely through conversational prompts. The session comprises four consecutive tasks; we reproduce the prompts verbatim and summarize the agent’s responses.

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

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

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

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

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

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



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

## 373 7 Related work

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

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

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

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

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

## 409 8 Conclusion

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

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

## References

- Lakshya A. Agrawal, Shangyin Tan, Dilara Soylu, Noah Ziemis, Rishi Khare, Krista Opsahl-Ong, Arnav Singhvi, Herumb Shandilya, Michael J. Ryan, Meng Jiang, Christopher Potts, Koushik Sen, Alexandros G. Dimakis, Ion Stoica, Dan Klein, Matei Zaharia, and Omar Khattab. GEPA: Reflective prompt evolution can outperform reinforcement learning. In *International Conference on Learning Representations (ICLR)*, 2026. Oral. arXiv preprint arXiv:2507.19457.
- Algorithmic Superintelligence. OpenEvolve: Open-source implementation of AlphaEvolve. <https://github.com/algorithmicsuperintelligence/openevolve>, 2025.
- Anthropic. Claude Code: Anthropic’s agentic coding system. <https://www.anthropic.com/product/claude-code>, 2025.
- Anthropic. Introducing Claude Opus 4.6. <https://www.anthropic.com/news/claude-opus-4-6>, 2026a.
- Anthropic. Introducing Claude Opus 4.7. <https://www.anthropic.com/news/claude-opus-4-7>, 2026b.
- Worawalan Chatlatanagulchai, Hao Li, Yutaro Kashiwa, Brittany Reid, Kundjanasith Thonglek, Pattara Lee-laprute, 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.

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

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

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

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

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

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

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

486 Alexander Novikov, Ngân Vŭ, Marvin Eisenberger, Emilien Dupont, Po-Sen Huang, Adam Zsolt Wagner, Sergey  
487 Shirobokov, Borislav Kozlovskii, Francisco J. R. Ruiz, Abbas Mehrabian, M. Pawan Kumar, Abigail See,  
488 Swarat Chaudhuri, George Holland, Alex Davies, Sebastian Nowozin, Pushmeet Kohli, and Matej Balog.  
489 AlphaEvolve: A coding agent for scientific and algorithmic discovery. *arXiv preprint arXiv:2506.13131*,  
490 2025.

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

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

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

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

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

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

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

505 Aymeric Roucher, Albert Villanova del Moral, Thomas Wolf, Leandro von Werra, and Erik Kaunismäki. smola-  
506 gents: A smol library to build great agentic systems. <https://github.com/huggingface/smolagents>,  
507 2025.

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

511 Noah Shinn, Federico Cassano, Ashwin Gopinath, Karthik Narasimhan, and Shunyu Yao. Reflexion: Language  
512 agents with verbal reinforcement learning. In *Advances in Neural Information Processing Systems (NeurIPS)*,  
513 2023.

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

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

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

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

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

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

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

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

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

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

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

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

544 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 KISS Sorcar 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.

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

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

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

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

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

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

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

### 639 A.3 Pre-flight Checks

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

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

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

649 **“Read every file before modifying it.”** The most common source of agent-introduced bugs is  
650 modifying a file based on an incorrect assumption about its current contents. The model may  
651 “remember” an older version of the file from its training data, or it may extrapolate from a partial  
652 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.

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

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

## 717 A.5 Deep Work Rules

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

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

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

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

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

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

## 748 A.6 Planning for Complex Tasks

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

```
751 ## Complex Task Planning
752
753 For 3+ files, cross-module, or architectural work:
754
755 1. List files to change and why.
756 1. State exact intended change per file.
757 1. Identify dependencies and execution order.
758 1. State verification method per change.
759
760 Skip for simple single-file tasks.
```

763 Each planning step targets a specific failure mode:



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

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

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

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

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

## 781 A.7 Testing Instructions

782 The testing section is perhaps the most opinionated:

```
783 ## Testing
784
785 - Run lint/typecheckers; fix all errors. Achieve 100% branch coverage. Every error, including pre-existing
786 ones, is yours---don't skip.
787 - NO mocks, patches, fakes, or test doubles. Write integration/e2e tests. Each test independent, verifying
788 actual behavior.
789 - **Only run impacted tests after modifications.**
790 - To confirm races: add random sleep (<0.1s) before racing statements.
791
792
```

793 Each testing instruction addresses a specific concern:

794 **“Run lint/typecheckers; fix all errors. Achieve 100% branch coverage.”** Before committing any  
795 change, the agent must ensure it does not introduce lint violations or type errors. This catches a  
796 broad class of issues—unused imports, type mismatches, style violations—that would otherwise  
797 accumulate across tasks. Full branch coverage ensures that every conditional path in the code under  
798 test has been exercised. LLMs tend to write happy-path tests that cover the main code path but ignore  
799 error handling, edge cases, and early-return branches. The 100% target forces the model to write  
800 tests for every branch, including error paths and boundary conditions. Moreover, such tests help with  
801 regression—developers can confidently use AI coding agents without fear that changes will break  
802 existing program behavior.

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

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

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

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

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

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

## 831 A.8 Self-Improvement Loop

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

```
833 ## Self-Improvement Loop
834
835
836 Read PWD/USER_PREFS.md at task start. Update with user preferences/invariants (no code snippets/symbols;
837 skip one-off tasks). Remove conflicting old entries carefully and thoroughly.
```

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

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

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

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

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

## 864 A.9 Pre-Finish Verification

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

```
866 ## Pre-Finish Verification
867
868
869 Before finish(success=True):
870
871 1. Re-read and verify every modified file.
872 1. Run required checks (lint, typecheck, tests); fix failures.
```

```

873 1. Check each user requirement against delivery.
874 1. If any check fails, keep working.
875 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:

```

900 ## Web Research (MANDATORY)
901
902 - **Visit >=30 websites every search. Hard requirement---don't stop before 30 or rationalize fewer.**
903 - Procedure:
904   1. Create PWD/tmp/information-{unique_id}.md: '# Web Research --- Websites visited: 0/30'
905   1. Per site, append: '## [N/30] URL' + extracted info. Update header counter each visit.
906   1. **Don't proceed until counter >=30.**
907   1. If results dry up, try different queries, synonyms, official docs, GitHub repos/issues, Stack Overflow
908     , blogs, Reddit, papers, API refs.
909   1. After 30, review and think deeply.
910 - Ask user for login help when needed.
911

```

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  $\geq 30$  websites” threshold is deliberately aggressive: it ensures the agent does not shortcut 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.

## 928 A.11 File Browsing Protocol

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

```
932 ## File Browsing
933
934 Collect info and code snippets in PWD/tmp/file-information-{unique_id}.md without overthinking, then
935 review and think deeply.
936
```

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

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

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

## 957 A.12 Desktop Application Control

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

```
959 ## Desktop Apps
960
961 Use screenshots, keyboard, and mouse. Don't launch VS Code or its extensions.
962
```

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

## 970 A.13 Sorcar-Specific Overrides

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

```
973 ## Sorcar-specific
974
975 - Lint/typecheck/format: 'uv run check --full'. Test: 'uv run pytest -v' (timeout 900s).
976 - **Do NOT install the KISS Sorcar extension from inside Sorcar.**
977 - To open/edit system prompt: ~/.vscode/extensions/ksenxx.kiss-sorcar-2026.5.8/kiss_project/src/kiss/
978   SYSTEM.md
979 - KISS Sorcar info: https://github.com/ksenxx/kiss_ai/blob/main/papers/kissorcar/kiss_sorcar.tex
980 - Third-party agents: kiss/agents/third_party_agents
981 - Official Claude SKILLS: kiss/agents/claude_skills
982 - Authenticate unauthenticated third-party agents; ask user only when a page needs auth.
983 - Read PWD/SORCAR.md as overriding instructions.
984
```

986 These overrides serve seven purposes. First, they specify the exact toolchain commands for the KISS  
 987 project itself (`uv run check --full, uv run pytest`), eliminating guesswork about which linter,  
 988 formatter, or test runner to use. Second, they prevent dangerous self-modification: just as a surgeon  
 989 should not operate on themselves, the agent must not install or reinstall the VS Code extension it  
 990 is running inside, since doing so would terminate its own process. Third, they provide navigation  
 991 instructions for the agent’s own source: when the user asks to edit the system prompt, the agent  
 992 knows the exact file path within the installed VS Code extension directory, avoiding guesswork about  
 993 where the file lives. Moreover, it does not create unnecessary UI components in KISS Sorcar just  
 994 to edit the system prompt. Fourth, the agent is given a URL to its own paper’s source as a source  
 995 of self-knowledge: when a user asks the agent about its own capabilities, design, or identity, it  
 996 can retrieve and read the paper rather than hallucinating an answer. This creates a self-referential  
 997 loop in which the documentation describing the agent is also consumed by the agent. Fifth, the  
 998 instructions expose a third-party agent integration layer: the agent is told where third-party agents  
 999 live (`kiss/agents/third_party_agents`). When a third-party agent requires authentication, the  
 1000 agent handles it autonomously and only prompts the user when a page genuinely requires user  
 1001 credentials—reducing unnecessary interruptions while maintaining security. Sixth, official Claude  
 1002 SKILLS [Anthropic, 2025] are bundled in `kiss/agents/claude_skills`, giving the agent access  
 1003 to Anthropic’s curated skill library for common software engineering patterns; by indicating where  
 1004 these skills reside, the agent can reference and apply them when relevant tasks arise. Seventh, the  
 1005 `SORCAR.md` override mechanism allows per-repository instructions to further customize the agent’s  
 1006 behavior, forming a hierarchy: general system prompt → Sorcar-specific instructions → repository-  
 1007 specific `SORCAR.md`. This approach also enables us to not hardcode `SORCAR.md` in the KISS Sorcar  
 1008 code, removing one more dependency. In the `SORCAR.md`, one can include more markdown files for  
 1009 further instructions.

## 1010 **B Painless software engineering (continued)**

```
1011 ## Desktop Apps
1012
1013 Use screenshots, keyboard, and mouse. Don't launch VS Code or its extensions.
1014
```

1016 The prohibition on launching VS Code prevents a recursive loop: since the agent runs inside a  
 1017 VS Code extension, launching another instance could corrupt the host session or create deadlocks.

1018 **Step 3: Investigating unexpected post-merge state.** After testing the simplified flow, the developer  
1019 notices that files appear in the Source Control panel of VS Code on the original branch after pressing  
1020 “Commit and Merge” and asks:

1021 After the user presses "Commit and Merge", why do the modified files show up as  
1022 committed in the original branch?  
1023

1025 The agent traces the exact execution path through `squash_merge_branch()` and discovers that  
1026 the implementation deliberately *unstages* the squash-merged changes via `git reset HEAD` so they  
1027 appear as uncommitted working-tree modifications in VS Code’s Source Control panel. The design  
1028 intention was to let the user review the diff before committing manually. The agent reports this  
1029 finding along with the relevant code snippet and a verified end state showing no new commits on the  
1030 branch.

1031 **Step 4: Directing a design change in one sentence.** The developer realizes that code review  
1032 already happened while the worktree was active and decides the unstage step is unnecessary:

1033 The review is already happening in the worktree branch. You don’t need the user  
1034 to review the modified and new files in the original branch. Fix it.  
1035

1037 The agent replaces the `git reset HEAD` call in `squash_merge_branch()` with a conditional `git`  
1038 `commit -no-edit` that uses the auto-generated squash message. It adds a `git diff -cached`  
1039 `-quiet` guard for the edge case where the merge produces no changes. One existing test  
1040 (`test_merge_leaves_changes_uncommitted`) is renamed to `test_merge_commits_changes`  
1041 and its assertions are updated to verify a clean working tree. All 104 worktree tests pass.

## 1042 NeurIPS Paper Checklist

### 1043 1. Claims

1044 Question: Do the main claims made in the abstract and introduction accurately reflect the  
1045 paper’s contributions and scope?

1046 Answer: [Yes]

1047 Justification: The abstract and introduction (Section 1) clearly state the contributions—a  
1048 five-layer agent architecture, a structured system prompt, a VS Code extension, and a 62.2%  
1049 pass rate on Terminal-Bench 2.0—and the scope is limited to software engineering tasks.  
1050 The conclusion’s claim that KS Assistant matches or beats both the Cursor Composer 2 and  
1051 Claude Code agents is directly supported by the head-to-head Terminal-Bench 2.0 results  
1052 in Section 3 (62.2% vs. 61.7% and 58%, on the same benchmark and base model), and the  
1053 claim that KS Assistant was built using itself is documented in Section 1 and discussed  
1054 throughout Sections 2–3.

1055 Guidelines:

- 1056 • The answer [N/A] means that the abstract and introduction do not include the claims  
1057 made in the paper.
- 1058 • The abstract and/or introduction should clearly state the claims made, including the  
1059 contributions made in the paper and important assumptions and limitations. A [No] or  
1060 [N/A] answer to this question will not be perceived well by the reviewers.
- 1061 • The claims made should match theoretical and experimental results, and reflect how  
1062 much the results can be expected to generalize to other settings.
- 1063 • It is fine to include aspirational goals as motivation as long as it is clear that these goals  
1064 are not attained by the paper.

### 1065 2. Limitations

1066 Question: Does the paper discuss the limitations of the work performed by the authors?

1067 Answer: [Yes]

1068 Justification: Section 3 discusses consistently failed tasks and their failure categories (graph-  
1069 ical/multimedia requirements, resource-intensive builds, niche domain knowledge). The  
1070 conclusion acknowledges that results depend on a single frontier model (Claude Opus 4.6)  
1071 and a single benchmark.

1072 Guidelines:

- 1073 • The answer [N/A] means that the paper has no limitation while the answer [No] means  
1074 that the paper has limitations, but those are not discussed in the paper.
- 1075 • The authors are encouraged to create a separate “Limitations” section in their paper.
- 1076 • The paper should point out any strong assumptions and how robust the results are to  
1077 violations of these assumptions (e.g., independence assumptions, noiseless settings,  
1078 model well-specification, asymptotic approximations only holding locally). The authors  
1079 should reflect on how these assumptions might be violated in practice and what the  
1080 implications would be.
- 1081 • The authors should reflect on the scope of the claims made, e.g., if the approach was  
1082 only tested on a few datasets or with a few runs. In general, empirical results often  
1083 depend on implicit assumptions, which should be articulated.
- 1084 • The authors should reflect on the factors that influence the performance of the approach.  
1085 For example, a facial recognition algorithm may perform poorly when image resolution  
1086 is low or images are taken in low lighting. Or a speech-to-text system might not be  
1087 used reliably to provide closed captions for online lectures because it fails to handle  
1088 technical jargon.
- 1089 • The authors should discuss the computational efficiency of the proposed algorithms  
1090 and how they scale with dataset size.
- 1091 • If applicable, the authors should discuss possible limitations of their approach to  
1092 address problems of privacy and fairness.

- While the authors might fear that complete honesty about limitations might be used by reviewers as grounds for rejection, a worse outcome might be that reviewers discover limitations that aren't acknowledged in the paper. The authors should use their best judgment and recognize that individual actions in favor of transparency play an important role in developing norms that preserve the integrity of the community. Reviewers will be specifically instructed to not penalize honesty concerning limitations.

### 3. Theory assumptions and proofs

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

Answer: [N/A]

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

Guidelines:

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

### 4. Experimental result reproducibility

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

Answer: [Yes]

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

Guidelines:

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



- 1146 (b) If the contribution is primarily a new model architecture, the paper should describe  
1147 the architecture clearly and fully.
- 1148 (c) If the contribution is a new model (e.g., a large language model), then there should  
1149 either be a way to access this model for reproducing the results or a way to reproduce  
1150 the model (e.g., with an open-source dataset or instructions for how to construct  
1151 the dataset).
- 1152 (d) We recognize that reproducibility may be tricky in some cases, in which case  
1153 authors are welcome to describe the particular way they provide for reproducibility.  
1154 In the case of closed-source models, it may be that access to the model is limited in  
1155 some way (e.g., to registered users), but it should be possible for other researchers  
1156 to have some path to reproducing or verifying the results.

## 1157 5. Open access to data and code

1158 Question: Does the paper provide open access to the data and code, with sufficient instruc-  
1159 tions to faithfully reproduce the main experimental results, as described in supplemental  
1160 material?

1161 Answer: [No] The code repository link is withheld to preserve double-blind review. The  
1162 system is open-source and the code will be made available upon acceptance.

1163 Justification: The open-source repository link is withheld to respect double-blind review  
1164 (stated in Section 1). The system will be publicly released upon acceptance.

1165 Guidelines:

- 1166 • The answer [N/A] means that paper does not include experiments requiring code.
- 1167 • Please see the NeurIPS code and data submission guidelines ([https://neurips.cc/  
1168 public/guides/CodeSubmissionPolicy](https://neurips.cc/public/guides/CodeSubmissionPolicy)) for more details.
- 1169 • While we encourage the release of code and data, we understand that this might not  
1170 be possible, so [No] is an acceptable answer. Papers cannot be rejected simply for not  
1171 including code, unless this is central to the contribution (e.g., for a new open-source  
1172 benchmark).
- 1173 • The instructions should contain the exact command and environment needed to run to  
1174 reproduce the results. See the NeurIPS code and data submission guidelines ([https:  
1175 //neurips.cc/public/guides/CodeSubmissionPolicy](https://neurips.cc/public/guides/CodeSubmissionPolicy)) for more details.
- 1176 • The authors should provide instructions on data access and preparation, including how  
1177 to access the raw data, preprocessed data, intermediate data, and generated data, etc.
- 1178 • The authors should provide scripts to reproduce all experimental results for the new  
1179 proposed method and baselines. If only a subset of experiments are reproducible, they  
1180 should state which ones are omitted from the script and why.
- 1181 • At submission time, to preserve anonymity, the authors should release anonymized  
1182 versions (if applicable).
- 1183 • Providing as much information as possible in supplemental material (appended to the  
1184 paper) is recommended, but including URLs to data and code is permitted.

## 1185 6. Experimental setting/details

1186 Question: Does the paper specify all the training and test details (e.g., data splits, hyperpa-  
1187 rameters, how they were chosen, type of optimizer) necessary to understand the results?

1188 Answer: [Yes]

1189 Justification: Section 3 specifies the benchmark, model, hardware, number of trials, skipped  
1190 tasks, and evaluation methodology. No training or fine-tuning is performed.

1191 Guidelines:

- 1192 • The answer [N/A] means that the paper does not include experiments.
- 1193 • The experimental setting should be presented in the core of the paper to a level of detail  
1194 that is necessary to appreciate the results and make sense of them.
- 1195 • The full details can be provided either with the code, in appendix, or as supplemental  
1196 material.

## 1197 7. Experiment statistical significance

1198 Question: Does the paper report error bars suitably and correctly defined or other appropriate  
1199 information about the statistical significance of the experiments?

1200 Answer: [Yes]

1201 Justification: We run 5 independent trials per task (445 total runs) and report pass@any  
1202 (78.7%), pass@all (43.8%), and overall pass rate (62.2%), as well as the distribution of  
1203 always-pass, always-fail, and mixed-result tasks. Median and mean cost and duration are  
1204 reported.

1205 Guidelines:

- 1206 • The answer [N/A] means that the paper does not include experiments.
- 1207 • The authors should answer [Yes] if the results are accompanied by error bars, confidence  
1208 intervals, or statistical significance tests, at least for the experiments that support the  
1209 main claims of the paper.
- 1210 • The factors of variability that the error bars are capturing should be clearly stated (for  
1211 example, train/test split, initialization, random drawing of some parameter, or overall  
1212 run with given experimental conditions).
- 1213 • The method for calculating the error bars should be explained (closed form formula,  
1214 call to a library function, bootstrap, etc.)
- 1215 • The assumptions made should be given (e.g., Normally distributed errors).
- 1216 • It should be clear whether the error bar is the standard deviation or the standard error  
1217 of the mean.
- 1218 • It is OK to report 1-sigma error bars, but one should state it. The authors should  
1219 preferably report a 2-sigma error bar than state that they have a 96% CI, if the hypothesis  
1220 of Normality of errors is not verified.
- 1221 • For asymmetric distributions, the authors should be careful not to show in tables or  
1222 figures symmetric error bars that would yield results that are out of range (e.g., negative  
1223 error rates).
- 1224 • If error bars are reported in tables or plots, the authors should explain in the text how  
1225 they were calculated and reference the corresponding figures or tables in the text.

## 1226 8. Experiments compute resources

1227 Question: For each experiment, does the paper provide sufficient information on the com-  
1228 puter resources (type of compute workers, memory, time of execution) needed to reproduce  
1229 the experiments?

1230 Answer: [Yes]

1231 Justification: Section 3 specifies the hardware (2025 MacBook Air 15" with M4 processor  
1232 and 24 GB RAM) and reports median/mean cost (\$0.45/\$0.90 per trial) and median/mean  
1233 duration (202 s/446 s per trial).

1234 Guidelines:

- 1235 • The answer [N/A] means that the paper does not include experiments.
- 1236 • The paper should indicate the type of compute workers CPU or GPU, internal cluster,  
1237 or cloud provider, including relevant memory and storage.
- 1238 • The paper should provide the amount of compute required for each of the individual  
1239 experimental runs as well as estimate the total compute.
- 1240 • The paper should disclose whether the full research project required more compute  
1241 than the experiments reported in the paper (e.g., preliminary or failed experiments that  
1242 didn't make it into the paper).

## 1243 9. Code of ethics

1244 Question: Does the research conducted in the paper conform, in every respect, with the  
1245 NeurIPS Code of Ethics <https://neurips.cc/public/EthicsGuidelines>?

1246 Answer: [Yes]

1247 Justification: We have reviewed the NeurIPS Code of Ethics. The research involves building  
1248 and evaluating a software engineering assistant; no human subjects, sensitive data, or  
1249 dual-use concerns are involved.

- 1250 Guidelines:
- 1251 • The answer [N/A] means that the authors have not reviewed the NeurIPS Code of
  - 1252 Ethics.
  - 1253 • If the authors answer [No], they should explain the special circumstances that require a
  - 1254 deviation from the Code of Ethics.
  - 1255 • The authors should make sure to preserve anonymity (e.g., if there is a special consid-
  - 1256 eration due to laws or regulations in their jurisdiction).

## 10. Broader impacts

1258 Question: Does the paper discuss both potential positive societal impacts and negative

1259 societal impacts of the work performed?

1260 Answer: [Yes]

1261 Justification: The paper discusses positive impacts (making software development more

1262 accessible, reducing developer toil). Potential negative impacts include job displacement for

1263 routine programming tasks and the risk of generating vulnerable code if the system prompt's

1264 verification disciplines are removed. The system mitigates the latter through mandatory

1265 pre-finish verification (Appendix A.9) and test-first discipline (Appendix A.7).

1266 Guidelines:

- 1267 • The answer [N/A] means that there is no societal impact of the work performed.
- 1268 • If the authors answer [N/A] or [No], they should explain why their work has no societal
- 1269 impact or why the paper does not address societal impact.
- 1270 • Examples of negative societal impacts include potential malicious or unintended uses
- 1271 (e.g., disinformation, generating fake profiles, surveillance), fairness considerations
- 1272 (e.g., deployment of technologies that could make decisions that unfairly impact specific
- 1273 groups), privacy considerations, and security considerations.
- 1274 • The conference expects that many papers will be foundational research and not tied
- 1275 to particular applications, let alone deployments. However, if there is a direct path to
- 1276 any negative applications, the authors should point it out. For example, it is legitimate
- 1277 to point out that an improvement in the quality of generative models could be used to
- 1278 generate Deepfakes for disinformation. On the other hand, it is not needed to point out
- 1279 that a generic algorithm for optimizing neural networks could enable people to train
- 1280 models that generate Deepfakes faster.
- 1281 • The authors should consider possible harms that could arise when the technology is
- 1282 being used as intended and functioning correctly, harms that could arise when the
- 1283 technology is being used as intended but gives incorrect results, and harms following
- 1284 from (intentional or unintentional) misuse of the technology.
- 1285 • If there are negative societal impacts, the authors could also discuss possible mitigation
- 1286 strategies (e.g., gated release of models, providing defenses in addition to attacks,
- 1287 mechanisms for monitoring misuse, mechanisms to monitor how a system learns from
- 1288 feedback over time, improving the efficiency and accessibility of ML).

## 11. Safeguards

1290 Question: Does the paper describe safeguards that have been put in place for responsible

1291 release of data or models that have a high risk for misuse (e.g., pre-trained language models,

1292 image generators, or scraped datasets)?

1293 Answer: [N/A]

1294 Justification: The paper does not release a pretrained language model or dataset. The system

1295 is an agent framework that wraps existing commercial LLM APIs.

1296 Guidelines:

- 1297 • The answer [N/A] means that the paper poses no such risks.
- 1298 • Released models that have a high risk for misuse or dual-use should be released with
- 1299 necessary safeguards to allow for controlled use of the model, for example by requiring
- 1300 that users adhere to usage guidelines or restrictions to access the model or implementing
- 1301 safety filters.

- 1302 • Datasets that have been scraped from the Internet could pose safety risks. The authors  
1303 should describe how they avoided releasing unsafe images.  
1304 • We recognize that providing effective safeguards is challenging, and many papers do  
1305 not require this, but we encourage authors to take this into account and make a best  
1306 faith effort.

1307 **12. Licenses for existing assets**

1308 Question: Are the creators or original owners of assets (e.g., code, data, models), used in  
1309 the paper, properly credited and are the license and terms of use explicitly mentioned and  
1310 properly respected?

1311 Answer: [Yes]

1312 Justification: All tools, benchmarks, and models used are cited: Terminal-Bench 2.0, Harbor,  
1313 Claude Opus 4.6, Playwright, VS Code, and all related works. The system is open-source.

1314 Guidelines:

- 1315 • The answer [N/A] means that the paper does not use existing assets.  
1316 • The authors should cite the original paper that produced the code package or dataset.  
1317 • The authors should state which version of the asset is used and, if possible, include a  
1318 URL.  
1319 • The name of the license (e.g., CC-BY 4.0) should be included for each asset.  
1320 • For scraped data from a particular source (e.g., website), the copyright and terms of  
1321 service of that source should be provided.  
1322 • If assets are released, the license, copyright information, and terms of use in the  
1323 package should be provided. For popular datasets, [paperswithcode.com/datasets](https://paperswithcode.com/datasets)  
1324 has curated licenses for some datasets. Their licensing guide can help determine the  
1325 license of a dataset.  
1326 • For existing datasets that are re-packaged, both the original license and the license of  
1327 the derived asset (if it has changed) should be provided.  
1328 • If this information is not available online, the authors are encouraged to reach out to  
1329 the asset's creators.

1330 **13. New assets**

1331 Question: Are new assets introduced in the paper well documented and is the documentation  
1332 provided alongside the assets?

1333 Answer: [N/A]

1334 Justification: No new datasets or pretrained models are released. The open-source code will  
1335 be documented upon release.

1336 Guidelines:

- 1337 • The answer [N/A] means that the paper does not release new assets.  
1338 • Researchers should communicate the details of the dataset/code/model as part of their  
1339 submissions via structured templates. This includes details about training, license,  
1340 limitations, etc.  
1341 • The paper should discuss whether and how consent was obtained from people whose  
1342 asset is used.  
1343 • At submission time, remember to anonymize your assets (if applicable). You can either  
1344 create an anonymized URL or include an anonymized zip file.

1345 **14. Crowdsourcing and research with human subjects**

1346 Question: For crowdsourcing experiments and research with human subjects, does the paper  
1347 include the full text of instructions given to participants and screenshots, if applicable, as  
1348 well as details about compensation (if any)?

1349 Answer: [N/A]

1350 Justification: The paper does not involve crowdsourcing or research with human subjects.

1351 Guidelines:

1352 • The answer [N/A] means that the paper does not involve crowdsourcing nor research  
1353 with human subjects.

1354 • Including this information in the supplemental material is fine, but if the main contribu-  
1355 tion of the paper involves human subjects, then as much detail as possible should be  
1356 included in the main paper.

1357 • According to the NeurIPS Code of Ethics, workers involved in data collection, curation,  
1358 or other labor should be paid at least the minimum wage in the country of the data  
1359 collector.

1360 **15. Institutional review board (IRB) approvals or equivalent for research with human**  
1361 **subjects**

1362 Question: Does the paper describe potential risks incurred by study participants, whether  
1363 such risks were disclosed to the subjects, and whether Institutional Review Board (IRB)  
1364 approvals (or an equivalent approval/review based on the requirements of your country or  
1365 institution) were obtained?

1366 Answer: [N/A]

1367 Justification: The paper does not involve human subjects research.

1368 Guidelines:

1369 • The answer [N/A] means that the paper does not involve crowdsourcing nor research  
1370 with human subjects.

1371 • Depending on the country in which research is conducted, IRB approval (or equivalent)  
1372 may be required for any human subjects research. If you obtained IRB approval, you  
1373 should clearly state this in the paper.

1374 • We recognize that the procedures for this may vary significantly between institutions  
1375 and locations, and we expect authors to adhere to the NeurIPS Code of Ethics and the  
1376 guidelines for their institution.

1377 • For initial submissions, do not include any information that would break anonymity (if  
1378 applicable), such as the institution conducting the review.

1379 **16. Declaration of LLM usage**

1380 Question: Does the paper describe the usage of LLMs if it is an important, original, or  
1381 non-standard component of the core methods in this research? Note that if the LLM is used  
1382 only for writing, editing, or formatting purposes and does *not* impact the core methodology,  
1383 scientific rigor, or originality of the research, a declaration is not required.

1384 Answer: [Yes]

1385 Justification: The entire system is built around LLM usage. The paper also discloses that the  
1386 system was built using itself (Section 1). An initial rough draft of the paper was generated  
1387 from the code of KS Asistant and it README.md.

1388 Guidelines:

1389 • The answer [N/A] means that the core method development in this research does not  
1390 involve LLMs as any important, original, or non-standard components.

1391 • Please refer to our LLM policy in the NeurIPS handbook for what should or should not  
1392 be described.