
KISS Sorcar: A Stupidly-Simple General-Purpose and Software Engineering AI Assistant

Koushik Sen

EECS Department, UC Berkeley
ksen@berkeley.edu

"Everything should be made as simple as possible, but not simpler." — Albert Einstein

Abstract

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

We present **KISS Sorcar**, a general-purpose assistant and integrated development environment (IDE) built on top of the **KISS Agent Framework**, a stupidly-simple AI agent framework of roughly 1,850 lines of code. The framework addresses these gaps using a robust system prompt and through a five-layer agent hierarchy in which each layer adds exactly one concern: budget-tracked ReAct execution, automatic continuation across sub-sessions via summarization, coding, and browser tools with parallel sub-agents, persistent multi-turn chat with history recall, and git worktree isolation so every task runs on its own branch.

To assess the power of the KISS agent framework, we implemented KISS Sorcar as a free, open-source Visual Studio Code extension that runs locally and effectively for long-horizon tasks, and supports browser automation, multimodal input, and Docker containers.

In this research, we deliberately prioritize output quality over latency: giving a frontier model adequate time to validate its own output—running linters, type checkers, and tests—dramatically reduces the low-quality code that plagues faster but less thorough agents. The entire system was built using itself in 4.5 months, providing a continuous stress test in which any agent-introduced bug immediately impairs its own ability to work. On Terminal Bench 2.0, KISS Sorcar achieves a 62.2% overall pass rate with Claude Opus 4.6, comparing favorably to Claude Code (58%) and Cursor Composer 2 (61.7%).

1 Introduction

Modern Large language models (LLMs), such as Anthropic’s Claude Opus 4.6, OpenAI’s GPT 5.4, and Google’s Gemini 3.1, have demonstrated a remarkable ability to generate code, reason about software architecture, and use developer tools [Chen et al., 2021, Rozière et al., 2023]. A growing body of work has explored how to harness these capabilities for autonomous software engineering, from single-session agents that resolve GitHub issues [Yang et al., 2024, Wang et al., 2024b] to industrial products marketed as AI software and general assistants [GitHub, 2021, Cursor, 2024, Cognition Labs, 2024, Anthropic, 2025, OpenAI, 2025, OpenClaw AI, 2025]. Yet using an LLM as a practical software engineering or general assistant still exposes several stubborn gaps: context windows are finite, a single mistake can derail an entire session, agents get stuck in dead ends,

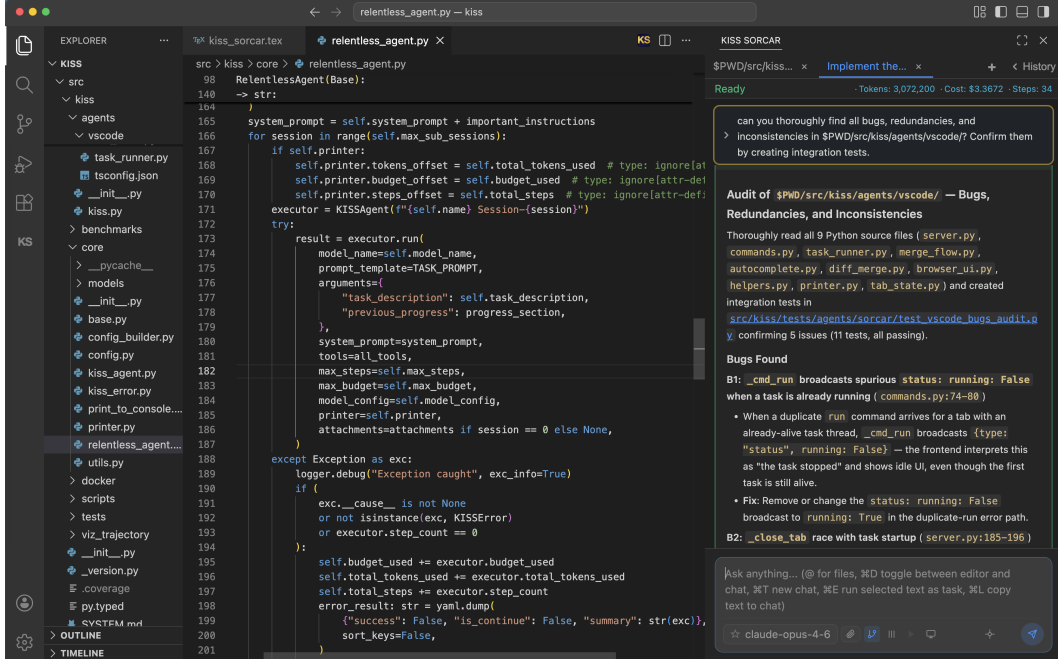


Figure 1: Screenshot of KISS Sorcar running as a VS Code extension. The sidebar shows the agent’s chat interface with real-time budget tracking, while the editor displays the code being modified.

models generate AI slop, and generated changes are difficult to review or revert once applied to a live codebase.

We propose the *KISS Agent Framework*, a stupidly simple AI agent framework containing around 1,850 lines of code. We address the above-mentioned gaps through a robust system prompt (which we describe in depth in Section 4) and a five-layer agent hierarchy in which each layer solves exactly one concern:

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

The name “KISS” reflects the *Keep It Simple, Stupid* design philosophy in software engineering: each layer is small, each concern is isolated, and the overall system avoids unnecessary abstraction.

To access the power and capabilities of the KISS agent framework, we implement KISS Sorcar. KISS Sorcar is a general-purpose assistant and an integrated development environment (IDE) built on top of the KISS agent framework. It codes really well and works pretty fast. We implemented KISS Sorcar as a Visual Studio Code extension and runs locally. It has full browser support (using open-source Chromium browser and Playwright), multimodal support, Docker container support, and OpenClaw-like features (whose discussion is beyond the scope of the paper). The good part is that KISS Sorcar is completely free and open-source; all one needs is a model API key from a major LLM provider, such as Anthropic. We implemented this framework in roughly 4.5 months, and the repository is available at https://github.com/ksenxx/kiss_ai. The name “Sorcar” pays homage to P. C. Sorcar, the legendary Bengali magician, evoking the idea of an agent that performs feats that appear magical yet are grounded in disciplined engineering.

In the project, we deliberately prioritize output quality over latency. In our experience, using a weaker or cheaper model often forces the developer to discard the agent’s work and retry, ultimately increasing the total cost of completing a task. Conversely, giving a frontier model adequate time to validate its own output—running linters, type checkers, and tests before declaring success—dramatically reduces the “slop” (low-quality, subtly incorrect code) that plagues faster but less thorough agents. We expect

token costs and inference latencies to continue to fall [Gao et al., 2025], making this quality-first posture increasingly practical. In the meantime, the code produced by our agent is consistently well-organized, simple, and idiomatic.

We built the system using itself. The entire codebase—the KISS Agent framework, the Sorcar agent layers, the VS Code extension, and the system prompt—was developed by KISS Sorcar 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, we immediately ask the agent to fix it by analyzing the trajectory and code of KISS Sorcar. The simplicity of the layered architecture was both a prerequisite for and a consequence of this bootstrapping process. The five core agent classes are remarkably compact: the KISS Agent comprises 409 lines, the Relentless Agent 297 lines, the Sorcar Agent 323 lines, the Chat Sorcar Agent 120 lines, and the Worktree Sorcar Agent 692 lines—a total of roughly 1,850 lines of code (excluding empty lines and comments).

We evaluate on Terminal Bench 2.0 and achieve a 62.2% overall pass rate using Claude Opus 4.6, comparing favorably to Claude Code (58%) and Cursor Composer 2 (61.7%) [Cursor Research, 2026] on the same benchmark (Section 3).

Outline. Section 2 presents the five-layer agent architecture and its motivating design principles. Section 3 reports evaluation results on Terminal Bench 2.0. Section 4 details the system prompt. Section 5 covers the VS Code extension. Section 6 illustrates painless software engineering through a real development session. Section 7 discusses related work, and Section 8 concludes.

2 Agent Architecture

We initially built the KISS Agent Framework to rapidly prototype and experiment with various prompt optimization techniques, such as Gepa [Agrawal et al., 2026], and evolutionary algorithms for algorithmic and code optimization, such as AlphaEvolve [Novikov et al., 2025] and OpenEvolve [Algorithmic Superintelligence, 2025]. We focused heavily on keeping the agent framework simple so we could rapidly prototype and experiment with ideas. The simplicity of the framework also enabled coding agents to write simple, bug-free code. We ended up not using any prompt optimization techniques when creating the prompts for KISS Sorcar. The KISS Agent Framework uses five agent layers using a strict inheritance chain. Each layer delegates upward for the concerns it does not own.

2.1 KISS Agent

The KISS Agent is the innermost execution unit. It implements a standard ReAct loop [Yao et al., 2023b] with the following characteristics. Listing 1 shows a complete, working agent in under ten lines of code. The developer defines an ordinary Python function (`calculate`), instantiates a `KISSAgent`, and calls its `run` method with a model name, a prompt template, template arguments, and a list of tools. The framework handles tool-schema generation, the ReAct loop, and budget tracking automatically.

```
from kiss.core.kiss_agent import KISSAgent

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

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

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

Native function calling. We register tools as ordinary Python callables. The agent builds an OpenAI-compatible tool schema once at setup time and caches it, avoiding redundant schema construction on every LLM call. A special `finish` tool signals task completion and returns the result to the caller.

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

Error resilience. The agent retries transient API errors (rate limits, server errors) up to a configurable threshold of consecutive failures. It detects non-retryable errors (authentication failures, permission denials) and raises them immediately.

Non-agentic mode. When tools are not needed, the agent can run a single generation without the ReAct loop, which is useful for summarization or question-answering sub-tasks.

The KISS Agent is stateless across runs: each call to its run method resets the conversation, token counters, and tool registry. This makes it safe to reuse a single agent instance for multiple sequential tasks.

2.2 Relentless Agent

The Relentless Agent wraps a KISS Agent in a continuation loop. Its core contribution is the ability to execute tasks that exceed a single context window by breaking them into sub-sessions.

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

Continuation protocol. The finish tool exposed to the inner KISS Agent accepts three fields: a success flag, a continue flag, and a summary. When the agent sets `is_continue=True`, the Relentless Agent starts a new sub-session with a fresh context window. The prompt for the new session includes a chronologically ordered list of all prior attempt summaries and instructs the agent not to redo completed work. The continuation prompt template is:

```
# Task Progress (Continuation {continuation_number})

{progress_text}

# Continue
- Complete the rest of the task.
- **DON'T** redo completed work.
- If you have been retrying the same approach without progress,
  step back and rethink the strategy from scratch.
```

Forced continuation on failure. If a sub-session raises an exception (e.g., the step limit is hit before the agent calls finish), the Relentless Agent does not abort. Instead, it saves the full trajectory to a temporary file, spawns a separate summarizer agent to read the trajectory and produce a concise summary, and then uses that summary as the progress text for the next sub-session. This ensures that even crashed sessions contribute useful context to subsequent attempts. The summarizer receives the following prompt:

```
# Summarizer

The trajectory of the agent is stored in the file: {trajectory_file}

# Instructions
- Read the trajectory file and analyze it. The trajectory file
  could be large.
- Return a precise chronologically-ordered list of things the
  agent did with the reason for doing that along with relevant
```

To force the agent to self-continue before hitting the step limit, we augment the system prompt with an instruction that fires near the end of the budget:

```
# MOST IMPORTANT INSTRUCTIONS
- **At step {step_threshold}: you MUST call
  finish(success=False, is_continue=True,
  summary="precise chronologically-ordered list of things
  the agent did with the reason for doing that along with
  relevant code snippets")** or if the task is not complete
  and you are at risk of running out of steps or context
  length.
- Work dir: {work_dir}
- Current process PID: {current_pid} -- NEVER kill this
  process.
```

2.3 Sorcar Agent

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

Coding tools. We provide four core tools: a shell command executor with streaming output, a file reader, a precise string-based file editor, and a file writer. The shell executor supports a configurable timeout, streams output to the user interface in real time, and respects a stop event that allows the user to cancel a running command. Note that we kept the tool names (Bash, Edit, Write, Read) the same as those in Claude Code, because the underlying Anthropic models do not make mistakes with those tools.

Browser automation. A web-use tool provides programmatic browser control: navigating to URLs, reading page accessibility trees, clicking elements, typing text, pressing keys, scrolling, and taking screenshots. This enables the agent to research documentation, verify deployed applications, and interact with web-based tools.

Parallel sub-agents. An optional parallel execution tool spawns independent Sorcar Agent instances in a thread pool. Each sub-agent gets its own LLM context and tool set. This is useful for embarrassingly parallel tasks such as summarizing multiple files or researching independent topics. We collect the results and return them in input order. We do not activate this tool by default because in the IDE, we cannot stream multiple agent outputs coherently in the chat window.

User interaction. An ask-user-question tool allows the agent to pause execution and request clarification from the user. In the VS Code integration, this renders as a text input in the sidebar; in CLI mode, it reads from standard input.

Docker isolation. When a Docker image is specified, we replace the coding tools with Docker-aware variants that execute commands inside a container, providing an additional layer of sandboxing for untrusted tasks.

2.4 Chat Sorcar Agent

The Chat Sorcar Agent adds multi-turn conversation persistence.

Chat sessions. We assign each task to a chat session identified by a stable chat ID. The agent persists tasks and their results to a local database. When a new task arrives within the same chat session, the agent loads all prior tasks and results and prepends them to the prompt as numbered context entries, allowing the LLM to reference earlier work.

Session management. The agent supports three operations: starting a new chat (with a fresh chat ID), resuming a chat by task description (which looks up the corresponding chat ID), and resuming by explicit chat ID. This enables both automatic session continuity in an IDE and manual session selection from the command line.

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

2.5 Worktree Sorcar Agent

The Worktree Sorcar Agent is the outermost layer and the one that users interact with in the VS Code extension. Its defining feature is git-worktree isolation.

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

Dirty-state preservation. If the user’s main working tree has uncommitted changes, the agent copies them into the worktree and creates a baseline commit. This ensures the agent sees the same state the user sees, while keeping the user’s actual index and working tree clean. During merge, we use cherry-pick from the baseline commit to replay only the agent’s changes, excluding the dirty-state snapshot.

Concurrency safety. A per-repository file lock serializes the checkout, stash, merge, and pop sequence so that concurrent tabs in the IDE cannot interleave operations on the same repository. Thread-local storage isolates per-task state (stream parsing buffers, bash output buffers, recording state) so that stopping one task does not corrupt another.

Crash recovery. We store all worktree state in git itself (branch names, git config entries) rather than in sidecar files. On process restart, the agent queries git for any pending branch matching its chat ID prefix and reconstructs all instance attributes from git config, enabling seamless recovery.

Graceful fallback. If the working directory is not inside a git repository, if the repository has no commits, or if HEAD is detached, the agent falls back to direct execution without worktree isolation, ensuring it never fails due to git preconditions.

3 Evaluation on Terminal Bench 2.0

Before we discuss a crucial aspect of KISS Sorcar, the system prompt, in a lengthy section, we describe the evaluation outcome of KISS Sorcar on the Terminal Bench 2.0, which was also recently used by the Cursor agent of Composer 2.0.

We evaluate our system on Terminal Bench 2.0,¹ a benchmark comprising 89 diverse terminal-based programming tasks, ranging from building legacy compilers and configuring servers to solving cryptanalysis challenges and training machine-learning models. Each task runs in an isolated Docker container; a separate verifier automatically judges the result. We use the Harbor² framework to orchestrate execution, and Claude Opus 4.6 as the underlying LLM. We do not modify the general system prompt or inject Terminal Bench 2.0-specific instructions during the evaluation. We carried out our evaluation on a 2025 MacBook Air 15".

3.1 Setup

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

3.2 Aggregate Results

Table 1 summarizes the aggregate statistics.

The 62.2% overall pass rate compares favorably to other agents using the same underlying model: at the time of writing, Claude Code (also Opus 4.6) scores approximately 58% on the Terminal Bench 2.0 leaderboard, and Cursor’s Composer 2—a custom fine-tuned model trained with large-scale reinforcement learning [Cursor Research, 2026]—achieves 61.7%. Our result suggests that the layered architecture and the structured system prompt described in Sections 2–4 contribute meaningfully beyond what the base model provides.

¹<https://www.tbench.ai/>

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

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

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

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—spanning systems, security, data engineering, and formal methods—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, MTEB leaderboard scraping, extracting moves from video), (2) tasks demanding very long or resource-intensive builds that exceed the container’s time or memory limits (CompCert, 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. KISS Sorcar does not score as high as other coding agents reported at the Terminal Bench 2.0 leaderboard, but we believe that our results are cheat-free and honest, showing reasonable improvements over the Claude Code and Cursor agents. 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, we discovered, using an automated benchmark audit agent, that 45 confirmed hacking solutions across 13 widely used benchmarks exhibited 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 describe its key sections.

4.1 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.

4.2 Tool Rules

Tool usage rules are explicit and mechanical:

```
# Rules

- Write() for new files. Edit() for small changes.
- Run Bash commands synchronously using the
  'timeout_seconds' parameter. Use 300s (default) for
  commands. If a command times out, retry with a higher
  timeout. Only for commands expected to exceed 10 minutes,
  run in the background with output redirected to a file
  and poll periodically.
- Use go_to_url() for browser tool.
- **The user cannot see intermediate chat. Show whatever
  user asks in the summary of the finish tool call.**
- READ large files in chunks.
- Create temporary files in PWD/tmp
- Use ULTRA thinking always
- **If you are running out of context length or steps, DO
  NOT try to complete the task urgently, but continue the
  task by calling 'finish'**
- PWD in the system prompt and user prompt denotes current
  working directory.
```

Each tool rule addresses a specific failure mode:

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

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

“Use `go_to_url()` for browser tool.” The agent has access to multiple tools that could plausibly interact with the web (shell-based `curl`, a Python program, the browser tool, etc.). This instruction eliminates ambiguity by specifying which tool to use for browser-based interactions.

“The user cannot see intermediate chat.” Users can get lost in the detailed trajectory generated by KISS Sorcar in the chat window. Often users want to see only the final summary returned by the `finish` tool, not the intermediate chain-of-thought or tool calls. Without this instruction, the model may “tell” the user something in an intermediate message and then assume the user has seen it, leading to confusion when the user asks for information the model believes it already provided. The instruction forces the model to put all user-facing information into the `finish` summary.

“READ large files in chunks.” Reading a 10,000-line file in a single tool call consumes a large fraction of the context window. By instructing the model to read files in chunks, the prompt prevents context window exhaustion from a single file read, preserving capacity for the rest of the task.

“Create temporary files in `PWD/tmp`.” Without this instruction, the model creates temporary files in unpredictable locations (the system `/tmp`, the home directory, or scattered throughout the project). Centralizing temporary files in a known directory makes cleanup predictable and prevents polluting the project tree with artifacts.

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

“If you are running out of context length or steps, DO NOT try to complete the task urgently, but continue the task by calling `finish`.” When the context window is nearly full, LLMs exhibit a “rush to finish” behavior: they skip verification steps, make hasty edits, and call `finish` with an incomplete result. This instruction redirects that urgency into the continuation protocol (Section 2.2), ensuring that a clean handoff to a new sub-session produces better results than a frantic attempt to squeeze everything into the remaining tokens.

“`PWD` in the system prompt and user prompt denotes 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.

4.3 Pre-flight Checks

Before modifying any file, we instruct the agent to read it first. These instructions were motivated by the GitHub project ‘get shit done’ `GetShitDone` [2025]:

```
## Pre-flight Checks

- Read every file you will modify before changing it.
- If the task depends on existing architecture or behavior,
  read the relevant source files first.
- If the task references files, commands, or config that do
  not exist, stop and ask or report instead of guessing.
- **When fixing a bug, an issue, or a race, write tests to
  confirm them. Then fix them.**
```

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

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

reading. By requiring a fresh read immediately before any edit, the instruction ensures that the model operates on the file’s current state, not a stale mental model.

“If the task depends on existing architecture or behavior, read the relevant source files first.”

This extends the previous 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 exist, and what invariants are assumed. The instruction prevents the model from jumping straight to code generation without understanding the broader context.

“If the task references files, commands, or config that do not exist, stop and ask or report instead of guessing.”

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 a bug, an issue, or a race, write tests to confirm them. Then fix them.”

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.

4.4 Code Style Guidelines

The prompt encodes a minimalist code philosophy:

```
## Code Style Guidelines

- Write simple, clean, and readable code with minimal
  indirection
- Avoid unnecessary object attributes, local variables, and
  config variables
- Avoid tight coupling among files and modules.
- Avoid object/struct attribute redirections
- No redundant abstractions or duplicate code
- Public methods MUST have full documentation
- Understand the root cause of an issue or bug, and patch
  the root cause instead of an ad hoc superficial fix.
- Before you write code, wait and think if the code is
  simple, elegant, general, and minimal.
- Once you finish the task, DO NOT write documentations
  unless the task specifically requires it.
```

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

“Write simple, clean, and readable code with minimal indirection.” LLMs tend to over-engineer solutions, introducing unnecessary abstractions, helper classes, and levels of indirection. Simple code is easier to review, test, and maintain.

“Avoid unnecessary object attributes, local variables, and config variables.” 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. Each unnecessary variable adds a name the developer must track and a site where the value could be accidentally mutated.

“Avoid tight coupling among files and modules.” 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. This instruction encourages modular design where each module has a clear, narrow interface.

“Avoid object/struct attribute redirections.” 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. LLMs commonly introduce such redirections when adapting code from one context to another. They make it harder to reason about ownership and lifetime.

“No redundant abstractions or duplicate code.” LLMs sometimes create a new utility function or class that duplicates functionality already present in the codebase, because they did not read the existing code thoroughly. This instruction reminds the model to check for existing implementations before creating new ones, and to factor out duplication when it encounters it.

“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.

“Understand the root cause of an issue or bug, and patch the root cause instead of an ad hoc superficial fix.” LLMs frequently apply symptom-level fixes: adding a null check where the real problem is that a variable should never be null, or catching an exception where the real problem is that the caller passes invalid arguments. This instruction forces the model to trace the causal chain to the root and fix it there.

“Before you write code, wait and think if the code is simple, elegant, general, and minimal.” This is a metacognitive instruction that asks the model to pause and evaluate its plan before committing to an implementation. By explicitly requesting this pause, the prompt encourages the model to spend more inference-time compute on design, reducing the likelihood of producing a first-draft solution that works but is unnecessarily complex.

“Once you finish the task, DO NOT write documentation unless the task specifically requires it.” Claude Opus 4.6 tends to generate many documentation files. These instructions prevent the behavior.

4.5 Deep Work Rules

```
## Deep Work Rules

- When the task says "align", "match", or "make consistent", read the target to determine the exact target state before editing. Never edit based on a vague reference.
- Use concrete values, not indirections. Instead of "update X to match Y", first read Y, then write the specific values into X.
- For multi-part work, list the concrete planned changes before executing them.
- Every meaningful change should have a concrete verification method (test, grep, CLI command).
```

The deep work rules (inspired by ‘get shit done’ GetShitDone [2025]) address a failure mode where the model interprets an instruction loosely and makes changes that are directionally correct but concretely wrong:

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

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

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

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

4.6 Planning for Complex Tasks

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

```
## Planning for Complex Tasks

For tasks involving 3+ files, cross-module changes, or
architectural work:

1. List the files that need to change and why.
1. State the exact intended change in each file.
1. Identify dependencies and execution order.
1. State how each change will be verified.

For simple single-file tasks, skip formal planning and
execute directly.
```

Each planning step targets a specific failure mode:

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

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

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

“State how each change will be verified.” The verification requirement from the Deep Work Rules is reinforced here at the planning stage, ensuring that verification is planned alongside the changes rather than treated as an afterthought.

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

4.7 Testing Instructions

The testing section is perhaps the most opinionated:

```
## Testing Instructions

- Run lint and typecheckers and fix any lint and typecheck
  errors.
- You MUST achieve 100% branch coverage
- Tests MUST NOT use mocks, patches, fakes, or any form of
  test doubles
- You MUST write integration tests
- Each test should be independent and verify actual behavior
- **Do NOT run all tests after modifications. Only run the
  impacted tests**
- To confirm a race condition, add sleep statements before
  racing statements with delays less than 0.1s
```

Each testing instruction addresses a specific concern:

“Run lint and typecheckers and fix any lint and typecheck errors.” Before committing any change, the agent must ensure it does not introduce lint violations or type errors. This catches a broad class of issues—unused imports, type mismatches, style violations—that would otherwise accumulate across tasks.

“You MUST achieve 100% branch coverage.” Full branch coverage ensures that every conditional path in the code under test has been exercised. LLMs tend to write happy-path tests that cover the

main code path but ignore error handling, edge cases, and early-return branches. The 100% target forces the model to write tests for every branch, including error paths and boundary conditions. Moreover, such tests help with regression — developers can confidently use AI coding agents without fear that changes will break existing program behavior.

“Tests MUST NOT use mocks, patches, fakes, or any form of test doubles.” This is the most opinionated rule. Mock tests that verify code call certain methods in a certain order — these test the implementation, not the behavior. A test suite built on mocks can pass with flying colors while the system is fundamentally broken, because the mocks hide the real dependencies. Integration tests that exercise actual behavior are more expensive to run but provide genuine confidence that the system works. Moreover, writing integration tests forces the model to think more deeply, often enabling the agent to find deeper bugs in the code.

“You MUST write integration tests.” This reinforces the no-mocks rule by explicitly requiring integration tests rather than unit tests. The distinction matters: a unit test in isolation may verify that a function produces the right output for a given input, but an integration test verifies that the function works correctly within the larger system—with real file I/O, real database connections, and real inter-module interactions.

“Each test should be independent and verify actual behavior.” Test independence means that running tests in any order produces the same results. Tests that depend on shared state or execution order are brittle and difficult to debug when they fail. “Verify actual behavior” reiterates that tests should assert on observable outcomes (return values, side effects, system state) rather than implementation details.

“Do NOT run all tests after modifications. Only run the impacted tests.” Running the full test suite after every small change is wasteful when only a few modules are affected. For a large project, a full test run may take minutes, and doing it after every edit adds up to significant wasted time and compute. This instruction directs the model to identify which tests are affected by its changes and run only those, improving iteration speed.

“To confirm a race condition, add sleep statements before racing statements with delays less than 0.1s.” Race conditions are notoriously difficult to reproduce because they depend on precise timing. By inserting small sleep delays at strategic points, the model can widen the race window and make the bug manifest deterministically during testing. The 0.1-second upper bound keeps the test fast while still being sufficient to expose most races.

4.8 Self-Improvement Loop

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

```
## Self-Improvement Loop

- Read the instructions in PWD/USER_PREFS.md at the start
  of each task.
- Then update PWD/USER_PREFS.md to capture the user
  preferences and invariants by analyzing the task. DO NOT
  ADD ANY CODE SNIPPETS OR SYMBOLS.
- You MUST carefully and thoroughly get rid of the user
  preferences and invariants that conflict with the newly
  added ones.
```

Each instruction in this section serves the goal of cross-session learning:

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

“Then update PWD/USER_PREFS.md to capture the user preferences and invariants by analyzing the task.” After completing a task, the agent may have learned new information about the user’s preferences: a preferred naming convention, a disliked pattern, a project-specific invariant. Writing these to the preferences file makes them available to future sessions. This mechanism allows the agent to accumulate project knowledge over time without requiring the user to repeat themselves.

“DO NOT ADD ANY CODE SNIPPETS OR SYMBOLS.” Code snippets in the preferences file are fragile: they become stale as the codebase evolves, and they consume tokens that would be better spent on natural-language descriptions of invariants. This instruction keeps the file compact and robust to code changes.

“You MUST carefully and thoroughly get rid of the user preferences and invariants that conflict with the newly added ones.” Over time, preferences may become contradictory—for example, an early preference might say “use camelCase for test methods” while a later correction says “use snake_case.” Without explicit conflict resolution, the file would accumulate contradictions, confusing the agent. This instruction mandates that the agent actively resolve conflicts when updating the file to keep it internally consistent. *Note that we do not keep learnings in a database or in various folders because it makes the information stale when lots of code changes are happening. It is impossible for an agent to eliminate stale information across databases and multiple folders.*

4.9 Pre-Finish Verification

Before declaring a task complete, the agent must pass a structured verification checklist (inspired by ‘get shit done’ GetShitDone [2025]):

```
## Pre-Finish Verification

Before calling finish(success=True, ...), you MUST:

1. Re-read every file you modified and verify the changes
   are correct.
1. Run the required checks (lint, typecheck, tests) and
   fix any failures.
1. Explicitly check each user requirement against what was
   delivered.
1. If any check fails, continue working instead of
   finishing.
1. If you have retried the same fix 3 times without
   progress, step back, rethink the approach from scratch,
   and try a different strategy.
```

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

“Re-read every file you modified and verify the changes are correct.” 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 the required checks (lint, typecheck, tests) and fix any 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.

“Explicitly check each user requirement against what was delivered.” 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, continue working instead of finishing.” 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.

“If you have retried the same fix 3 times without progress, step back, rethink the approach from scratch, and try a different strategy.” 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.”

4.10 Web Research Protocol

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

```

## Use web tools when you need to:

- When you need to collect knowledge from the internet,
  visit **AT LEAST 100 WEB SITES** and collect information
  necessary for the task without much thinking in a new
  file PWD/tmp/information-{unique_id}.md. Then go over
  information in PWD/tmp/information-{unique_id}.md, think
  deeply on how to complete the task at hand, and
  complete it.

- If you need to login to a website while browsing for
  information, you MUST ask the user to help you with
  login. You **MUST NOT** ask the user to login to Google.

```

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 “at least 100 web sites 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 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. The Google exception acknowledges that the agent can search the web without a Google account, and asking the user to log in to Google would provide no benefit while creating unnecessary friction.

4.11 File Browsing Protocol

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

```

## Browsing files for a task

- When you need to read files for a task,
  collect information necessary for the task without much
  thinking in a new file
  PWD/tmp/file-information-{unique_id}.md. Then go over
  information in
  PWD/tmp/file-information-{unique_id}.md, think deeply on
  how to complete the task at hand, and complete it.

```

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

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

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

4.12 Desktop Application Control

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

```
## Launch desktop apps

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

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

4.13 Sorcar-Specific Overrides

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

```
## Sorcar-specific instructions:

- Use 'uv run check --full' to lint, typecheck, and format
  code.
- Run 'uv run pytest -v' with a timeout of 900 seconds to
  test KISS
- **Do NOT install the KISS Sorcar extension from inside
  Sorcar**
- If the user ask to open or edit the system prompt, open
  ~/.vscode/extensions/ksenxx.kiss-sorcar-2026.4.22/
  kiss_project/src/kiss/SYSTEM.md
- Information about KISS Sorcar can be found at
  https://github.com/ksenxx/kiss_ai/blob/main/papers/
  kissorcar/kiss_sorcar.tex
- Third-party agents are available under the folder
  kiss/agents/third_party_agents
- If the user is not authenticated for a third party
  agent, authenticate the agent and ask user ONLY when a
  page needs user authentication
- **YOU MUST ASK THE USER BEFORE SENDING ANY EMAIL,
  MESSAGE, OR SUBMITTING A REQUEST**
- Read PWD/SORCAR.md and treat its contents as
  instructions and allow those instructions to override the
  instructions above
```

These overrides serve seven purposes. First, they specify the exact toolchain commands for the KISS project itself (`uv run check --full`, `uv run pytest`), eliminating guesswork about which linter, formatter, or test runner to use. Second, they prevent dangerous self-modification: just as a surgeon should not operate on themselves, the agent must not install or reinstall the VS Code extension it is running inside, since doing so would terminate its own process. Third, they provide a navigation instruction for the agent's own source: when the user asks to edit the system prompt, the agent knows the exact file path inside the installed VS Code extension directory, avoiding guesswork about where the file lives. Moreover, it does not create unnecessary UI components in KISS Sorcar just to edit the system prompt. Fourth, the agent is given a URL to its own paper's source as a source of self-knowledge: when a user asks the agent about its own capabilities, design, or identity, it can retrieve and read the paper rather than hallucinating an answer. This creates a self-referential loop where the documentation that describes the agent is also consumed by the agent. Fifth, the instructions expose a third-party agent integration layer: the agent is told where third-party agents live (`kiss/agents/third_party_agents`). When a third-party agent requires authentication, the agent handles it autonomously and only prompts the user when a page genuinely requires user credentials—reducing unnecessary interruptions while maintaining security. Sixth, a safety guardrail requires the agent to obtain explicit user confirmation before sending any email, message, or submitting any external request, preventing accidental or unauthorized outbound actions. Seventh, the `SORCAR.md` override mechanism allows per-repository instructions to further customize the agent's behavior,

forming a hierarchy: general system prompt → Sorcar-specific instructions → repository-specific SORCAR.md. This approach also enables us to not hardcode SORCAR.md in the KISS Sorcar code, removing one more dependency. In the SORCAR.md, one can include more markdown files for further instructions.

5 VS Code Extension: Unique Features

We release our system as a VS Code extension. While the underlying agent architecture (Sections 2 and 4) already differs from existing AI coding assistants, the extension’s user-facing design introduces several features that, to our knowledge, are not present in any competing IDE or assistant—including GitHub Copilot [GitHub, 2021], Cursor [Cursor, 2024], Windsurf [Codeium, 2024], Devin [Cognition Labs, 2024], and Aider [Gauthier, 2023]. We enumerate these features below.

5.1 Cross-Session Self-Improvement

Every AI assistant starts each session from scratch: the user must re-explain project conventions, preferred patterns, and past decisions. Some tools support a static configuration file (e.g., `.cursorrules` for Cursor, `AGENTS.md` or `CLAUDE.md` for various agents), but these files must be written and maintained by the developer.

Our extension maintains a `USER_PREFS.md` file that the agent reads at the start of each task and *updates* at the end of each task. When the agent discovers a new user preference or project invariant during task execution—a naming convention, an architectural constraint, a disliked pattern—it writes it to the file. Crucially, 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 by the developer.

5.2 Real-Time Budget Accountability

AI coding assistants typically operate on a subscription model (Copilot, Cursor) or a per-seat pricing model (Devin, Windsurf), both of which obscure the per-task cost. The developer has no visibility into how many tokens a task consumed or how much it cost.

Our 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. We enforce both per-task and global budget ceilings. If a task exceeds its budget, the agent raises a hard error rather than silently accumulating charges. This transparency allows developers to make informed decisions about which tasks to delegate to the AI and how to structure prompts for cost efficiency. The KISS Agent also appends the current usage in the context so that the model is fully aware of its limits.

5.3 Integrated Browser Automation

Our extension includes a browser automation tool that allows the agent to navigate to URLs, read accessibility trees, and click elements, type text, press keys, scroll, and take screenshots—all controlled programmatically from within a VS Code task via Playwright. We render a live browser preview in the extension’s sidebar, allowing the developer to watch the agent interact with web applications in real time.

This capability enables use cases that no other IDE assistant supports: verifying a deployed web application after a code change, filling out web forms as part of a testing workflow, scraping documentation to inform a code generation task, or interacting with web-based developer tools (CI dashboards, issue trackers) without leaving the editor.

6 Painless Software Engineering with KISS Sorcar

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 KISS Sorcar, we found a particular workflow to be particularly useful. In the workflow, we first ask KISS Sorcar to

generate a detailed, step-by-step description of a workflow or an algorithm, which we found buggy, and then we ask KISS Sorcar to change some buggy steps in the workflow 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 with git in worktree_sorcar_agent.py when a task finishes?

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

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

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

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

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

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

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

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

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

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

Discussion. The four-step session illustrates the “painless” development loop that our system enables. The developer never opens a source file, never writes a line of code, and never runs a test manually. Instead, the entire cycle—understand the workflow, redesign it, investigate an anomaly, direct a fix—happens through natural-language prompts, with the agent handling code reading, multi-file editing, test updates, and verification. This style of development becomes possible because of the agent hierarchy described in Section 2: the Worktree Sorcar Agent isolates changes on a branch, the Chat Sorcar Agent preserves conversational context across tasks, and the Relentless Agent automatically continues when the context window is exhausted.

7 Related Work

Code-specialized language models. Beyond the general-purpose LLMs that our system can use as backends, a rich line of work has produced models specifically trained for code. Code Llama [Rozière et al., 2023] fine-tunes Llama 2 for code generation and infilling. StarCoder [Li et al., 2023] trains on permissively licensed code from GitHub with a fill-in-the-middle objective. DeepSeek-Coder [Guo et al., 2024] trains on a 2-trillion-token corpus of code and natural language with a repository-level context window. More recently, frontier models have been optimized specifically for agentic software engineering. Claude Opus 4.6 [Anthropic, 2026] advances long-horizon coding and agentic task execution. Cursor released Composer 2 [Cursor Research, 2026], a custom fine-tuned coding model trained with large-scale reinforcement learning. Kimi K2.5 [Kimi Team, 2026] is an open-source multimodal agentic model that jointly optimizes text and vision and introduces Agent Swarm for parallel task decomposition. GLM-5.1 [Z.ai, 2026] is a 754-billion-parameter mixture-of-experts model from Z.ai that sustains autonomous coding over multi-hour sessions, achieving state-of-the-art on SWE-Bench Pro. Our system is model-agnostic and can leverage any of these models through its pluggable LLM backend, benefiting from advances in code-specialized pre-training without architectural changes.

Code generation agents. SWE-Agent [Yang et al., 2024] and OpenHands [Wang et al., 2024b] provide LLM-based agents for resolving software engineering tasks such as GitHub issues. Both use a single-session execution model without automatic continuation. Agentless [Xia et al., 2024] takes the opposite approach, showing that a simple two-phase localize-then-repair pipeline without autonomous agent loops can achieve competitive results on SWE-bench [Jimenez et al., 2024]. Devin [Cognition Labs, 2024], an industrial product marketed as “the first AI software engineer,” operates in a sandboxed environment with shell, browser, and editor access. Aider [Gauthier, 2023] provides a terminal-based pair programming interface with tight git integration, automatically committing each change. Our Relentless Agent layer addresses the single-session limitation common to most of these systems, while our worktree isolation provides stronger safety guarantees than per-change commits do.

ReAct and tool use. The ReAct framework [Yao et al., 2023b] interleaves reasoning and action. Toolformer [Schick et al., 2023] teaches models to use tools via self-supervised learning. We use native function calling provided by modern LLM APIs rather than in-context tool descriptions, thereby reducing prompt overhead and improving reliability.

Reasoning and planning. Chain-of-thought prompting [Wei et al., 2022] demonstrated that eliciting step-by-step reasoning dramatically improves LLM performance on complex tasks. Tree of Thoughts [Yao et al., 2023a] generalizes this to deliberate search over multiple reasoning paths. Reflexion [Shinn et al., 2023] introduces verbal reinforcement learning, where an agent reflects on failed attempts and produces self-critiques that improve subsequent trials. Our continuation protocol

is conceptually related to Reflexion: failed sub-sessions produce summaries that inform subsequent attempts. However, in our case, we use the summaries to continue the task.

Multi-agent software development. ChatDev [Qian et al., 2024] models the software development process as a conversation between role-playing agents (CEO, CTO, programmer, tester) organized in a waterfall-like pipeline. MetaGPT [Hong et al., 2024] takes a meta-programming approach, encoding standard operating procedures as structured outputs that coordinate specialized agents. AutoGen [Wu et al., 2023] provides a general-purpose framework for multi-agent conversation, enabling flexible topologies beyond fixed pipelines. These systems focus on generating entire applications from scratch. We take a different approach: rather than simulating an organization of specialists, we use a single agent with broad access to tools and optional parallel sub-agents for embarrassingly parallel sub-tasks, prioritizing practical utility on real-world codebases over role-playing fidelity.

Multi-turn autonomous agents. AutoGPT [Significant Gravititas, 2023] and BabyAGI [Nakajima, 2023] implement multi-step autonomous agents. These systems typically lack budget controls and safe code isolation. Our layered architecture addresses each of these concerns in a dedicated layer.

Software engineering benchmarks. SWE-bench [Jimenez et al., 2024] evaluates agents on real-world GitHub issues drawn from popular Python repositories, requiring the agent to localize and fix bugs given only the issue description. It has become the de facto standard for measuring agent capabilities in software engineering. HumanEval [Chen et al., 2021] and MBPP [Austin et al., 2021] evaluate function-level code generation from docstrings. LiveCodeBench [Jain et al., 2024] addresses benchmark contamination by continuously collecting fresh competition problems from LeetCode, AtCoder, and CodeForces, and extends evaluation to self-repair, code execution, and test output prediction. These benchmarks focus on isolated coding problems; We target the broader workflow of multi-file, multi-step software engineering tasks that require tool use, testing, and version control.

Agent frameworks and orchestration. LangChain [LangChain, 2022] provides modular abstractions for building LLM applications, including agent loops, tool integration, and memory. It focuses on composability and breadth of integrations rather than on the specific concerns of software development. Our architecture is purpose-built for software engineering, with each layer addressing a specific practical concern (budget tracking, continuation, code safety) rather than providing general-purpose primitives.

LLM agent surveys. Several comprehensive surveys have mapped the rapidly growing landscape of LLM-based agents. Xi et al. [2023] surveys the design space of LLM agents along three dimensions — brain (reasoning), perception (input modalities), and action (tool use) — and catalogs applications across social science, natural science, and engineering. Wang et al. [2024a] propose a systematic framework for autonomous agents built on LLMs, covering profile, memory, planning, and action modules. Our system can be understood through the lens of these frameworks: the KISS Agent implements the action loop, the Relentless Agent addresses the memory and planning concerns through continuation summaries, and the system prompt encodes the profile.

IDE integration. GitHub Copilot [GitHub, 2021], Cursor [Cursor, 2024], and Windsurf [Codeium, 2024] provide AI assistance within editors. Cursor recently released Composer 2 [Cursor Research, 2026], a custom fine-tuned coding model trained with large-scale reinforcement learning that achieves 61.7% on Terminal Bench 2.0. These tools typically operate at the level of code completion or single-turn chat. We operate at the level of autonomous multi-step task execution with full tool access and git-level isolation.

Recent advances in agentic software engineering. The field has matured rapidly since late 2025. Hassan et al. [2025] articulate the foundational pillars of *Agentic Software Engineering* (SE 3.0), defining a research roadmap that emphasizes trust, controllability, and goal-oriented task decomposition. Li et al. [2025] surveys the landscape of autonomous coding agents and characterizes the transition from assistive code completion to full-fledged AI teammates that initiate, plan, and execute development tasks. We instantiate several of the principles advocated in these roadmaps, including layered controllability (via budget tracking and step limits) and safe isolation (via worktrees).

Wang et al. [2025] provides a comprehensive survey of AI agentic programming techniques, cataloging how LLM-based agents decompose goals, interact with compilers and version control systems, and self-correct through feedback loops. Chatlatanagulchai et al. [2025] study *agent context files* — persistent, project-level instruction files that guide agentic coding tools — and find that high-quality context files significantly improve agent performance. Our layered system prompt and SORCAR.md

override mechanisms are instances of this pattern. Mohsenimofidi et al. [2025] further investigates context engineering for AI agents in open-source software, highlighting how curated context improves agent efficacy on repository-level tasks.

Evolving benchmarks for coding agents. SWE-bench Pro [Deng et al., 2025] introduces a substantially more challenging benchmark with 1,865 long-horizon problems drawn from 41 repositories, including commercial codebases, explicitly targeting enterprise-level complexity beyond the original SWE-bench. These long-horizon tasks — often requiring multi-file patches and hours of professional developer effort — directly motivate our continuation mechanism. Prathifkumar et al. [2025] raises the important question of benchmark contamination, showing that overlap between SWE-Bench-Verified problems and LLM training data may inflate scores, suggesting that high benchmark numbers may partly reflect memorization rather than genuine problem-solving ability. Horikawa et al. [2025] provides an empirical study of how AI coding agents handle refactoring tasks, finding that while agents can plan and execute complex refactorings, they still struggle with cross-file dependency analysis — a challenge that our sub-agent parallelism partially addresses.

Self-improvement and test-time scaling. Robeyns et al. [2025] demonstrates a self-improving coding agent that iteratively refines its own scaffolding code, achieving dramatic benchmark improvements through self-generated optimizations. Our self-improvement loop (via `USER_PREFS.md`) captures a lighter-weight form of this idea: rather than rewriting its own code, the agent accumulates learned user preferences and project invariants across sessions. Gao et al. [2025] introduces the Trae Agent, which applies test-time compute scaling to software engineering, dynamically allocating more inference budget to harder problems. Our budget-tracking mechanism provides a complementary perspective: rather than scaling compute adaptively, we enforce hard budget ceilings while the Relentless Agent ensures maximum progress within those bounds.

8 Conclusion

We have shown that a layered, single-concern architecture can address the practical challenges of deploying LLM agents for real-world software development. On Terminal Bench 2.0, a benchmark of 89 diverse terminal-based tasks evaluated across 5 trials each, our system achieves a 62.2% overall pass rate (277/445)—outperforming Claude Code (58%) and Cursor Composer 2 (61.7%) on the same benchmark with the same underlying model (Claude Opus 4.6). These results are achieved without benchmark-specific optimizations, fine-tuning, or reinforcement learning: the entire agent is roughly 1,850 lines of straightforward Python.

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

Finally, our experience building KISS Sorcar suggests that we are in the Golden Age of Experts. If someone is an expert in some particular domain and knows how to use Claude Opus 4.6 or a newer model effectively, they are the King. While implementing KISS and KISS Sorcar, we read the thinking tokens of Claude Opus 4.6, and we found it to be way more intelligent than anything or anyone we have encountered in the past. In fact, we found that Opus 4.6 is so intelligent that we feel that we are the bottleneck. We are exploring and exploiting Opus 4.6 and see "what is possible". We see that several academic research efforts will be obsolete in the future because Opus 4.6 can do them better with the right instructions. We see that people will be writing and customizing their own software to meet their needs rather than relying on large software vendors. We have to forget much conventional wisdom if we want to make big strides. Hopefully, in 3-5 years, this expert knowledge will be incorporated into the best frontier model—the future will be bright for AI experts and bleak for the rest of us.

While using it, we found that we don’t spend much time working. We are mostly getting bored. Although many experts suggest that AI will be harmful if it falls into the wrong hands without proper security and safety guardrails, we believe the biggest problem will be mental health issues, which could end human civilization if not properly addressed in advance.

An Important Advice

We found that researchers are publishing lots of papers on AI and it is hard to keep track of them or validate their claim. We propose to use the following prompt with KISS Sorcar to identify issues in blogs, papers, and code repositories:

Can you read <<url>>, and thoroughly and precisely check for **wrong assumptions**, **cheating**, **irreproducibility issues**, **fraud**, **potential for cheating in evaluation** and **security vulnerabilities**?
Use internet search extensively and do not believe what people say--verify them yourself.
Do not hesitate to download code and run them to validate results.
For security vulnerabilities, create a POC and test it.
Generate an html report in PWD/sorcar_reported_frauds/ and open in the user's default browser.
Thoroughly fact check everything you claim in the report.

In the prompt, replace «url» with the actual link to a blog, a paper, or a code repository.

Acknowledgments

This research is supported in part by gifts from Accenture, Amazon, AMD, Anyscale, Broadcom, Google, IBM, Intel, Intesa Sanpaolo, Lambda, Lightspeed, Mibura, NVIDIA, Samsung SDS, SAP, by the U.S. Department of Energy, Office of Science, Office of Advanced Scientific Computing Research through the X-STACK: Programming Environments for Scientific Computing program (DESC0021982,) and the Defense Advanced Research Projects Agency (DARPA) under Agreement No. HR00112590134.

We would like to thank Marius Momeu for finding critical bugs in the cost computation and in the UI usability, Yogya Mehrotra for testing and fixing bugs in the OpenClaw like third_party_agents in KISS Sorcar; Debabrata Dash, Yiwei Hou, Kaiyao Ke, Muxi Lyu, Anoop Mishra, Manish Shetty, Ion Stoica, Shangyin Tan, Hao Wang, and Matei Zaharia for various useful feedback.

Disclaimer: The paper has been edited in part using KISS Sorcar, an AI tool.

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>, 2026.
- Jacob Austin, Augustus Odena, Maxwell Nye, Maarten Bosma, Henryk Michalewski, David Dohan, Ellen Jiang, Carrie Cai, Michael Terry, Quoc Le, and Charles Sutton. Program synthesis with large language models. *arXiv preprint arXiv:2108.07732*, 2021.
- 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.

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.

GetShitDone. GetShitDone: Prompt engineering for coding agents. <https://github.com/gsd-build/get-shit-done>, 2025.

GitHub. GitHub Copilot: Your AI pair programmer. <https://github.com/features/copilot>, 2021.

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, Xiawu 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.

Kosei Horikawa, Hao Li, Yutaro Kashiwa, Bram Adams, Hajimu Iida, and Ahmed E. Hassan. Agentic refactoring: An empirical study of AI coding agents. *arXiv preprint arXiv:2511.04824*, 2025.

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.

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

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

LangChain. LangChain: Build context-aware reasoning applications. <https://github.com/langchain-ai/langchain>, 2022.

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

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

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

Yohei Nakajima. BabyAGI. <https://github.com/yoheinakajima/babyagi>, 2023.

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

OpenAI. Introducing Codex: A cloud-based software engineering agent. <https://openai.com/index/introducing-codex/>, 2025.

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

- Thanosan Prathifkumar, Noble Saji Mathews, and Meiyappan Nagappan. Does SWE-Bench-Verified test agent ability or model memory? *arXiv preprint arXiv:2512.10218*, 2025.
- Chen Qian, Wei Liu, Hongzhang Liu, Nuo Chen, Yufan Dang, Jiahao Li, Cheng Yang, Weize Chen, Yusheng Su, Xin Cong, Juyuan Xu, Dahai Li, Zhiyuan Liu, and Maosong Sun. ChatDev: Communicative agents for software development. In *Proceedings of the 62nd Annual Meeting of the Association for Computational Linguistics (ACL)*, 2024.
- Maxime Robeyns, Martin Szummer, and Laurence Aitchison. A self-improving coding agent. *arXiv preprint arXiv:2504.15228*, 2025.
- Baptiste Rozière, Jonas Gehring, Fabian Gloeckle, Sten Sootla, Itai Gat, Xiaoqing Ellen Tan, Yossi Adi, Jingyu Liu, Tal Remez, Jérémy Rapin, et al. Code Llama: Open foundation models for code. *arXiv preprint arXiv:2308.12950*, 2023.
- Timo Schick, Jane Dwivedi-Yu, Roberto Dessì, Roberta Raileanu, Maria Lomeli, Luke Zettlemoyer, Nicola Cancedda, and Thomas Scialom. Toolformer: Language models can teach themselves to use tools. In *Advances in Neural Information Processing Systems (NeurIPS)*, 2023.
- Noah Shinn, Federico Cassano, Ashwin Gopinath, Karthik Narasimhan, and Shunyu Yao. Reflexion: Language agents with verbal reinforcement learning. In *Advances in Neural Information Processing Systems (NeurIPS)*, 2023.
- Significant Gravitas. AutoGPT. <https://github.com/Significant-Gravitas/AutoGPT>, 2023.
- Adam Stein, Davis Brown, Hamed Hassani, Mayur Naik, and Eric Wong. Finding widespread cheating on popular agent benchmarks. Blog post, <https://debugml.github.io/cheating-agents/>, 2026a.
- Adam Stein, Davis Brown, Hamed Hassani, Mayur Naik, and Eric Wong. Detecting safety violations across many agent traces. *arXiv preprint arXiv:2604.11806*, 2026b.
- Hao Wang, Qiuyang Mang, Alvin Cheung, Koushik Sen, and Dawn Song. We scored 100% on AI benchmarks without solving a single problem. Blog post, <https://moogician.github.io/blog/2026/trustworthy-benchmarks/>, 2026.
- Huanting Wang, Jingzhi Gong, Huawei Zhang, Jie Xu, and Zheng Wang. AI agentic programming: A survey of techniques, challenges, and opportunities. *arXiv preprint arXiv:2508.11126*, 2025.
- Lei Wang, Chen Ma, Xueyang Feng, Zeyu Zhang, Hao Yang, Jingsen Zhang, Zhiyuan Chen, Jiakai Tang, Xu Chen, Yankai Lin, Wayne Xin Zhao, Zhewei Wei, and Ji-Rong Wen. A survey on large language model based autonomous agents. *Frontiers of Computer Science*, 18(6):186345, 2024a.
- Xingyao Wang, Yangyi Chen, Lifan Yuan, Yizhe Zhang, Yunzhi Li, Hao Peng, and Heng Ji. OpenHands: An open platform for AI software developers as generalist agents. *arXiv preprint arXiv:2407.16741*, 2024b.
- Jason Wei, Xuezhi Wang, Dale Schuurmans, Maarten Bosma, Brian Ichter, Fei Xia, Ed Chi, Quoc Le, and Denny Zhou. Chain-of-thought prompting elicits reasoning in large language models. In *Advances in Neural Information Processing Systems (NeurIPS)*, 2022.
- Qingyun Wu, Gagan Bansal, Jieyu Zhang, Yiran Wu, Beibin Li, Erkang Zhu, Li Jiang, Xiaoyun Zhang, Shaokun Zhang, Jiale Liu, Ahmed Hassan Awadallah, Ryen W. White, Doug Burger, and Chi Wang. AutoGen: Enabling next-gen LLM applications via multi-agent conversation. *arXiv preprint arXiv:2308.08155*, 2023.
- Zhiheng Xi, Wenxiang Chen, Xin Guo, Wei He, Yiwen Ding, Boyang Hong, Ming Zhang, Junzhe Wang, Senjie Jin, Enyu Zhou, et al. The rise and potential of large language model based agents: A survey. *arXiv preprint arXiv:2309.07864*, 2023.
- Chunqiu Steven Xia, Yinlin Deng, Soren Dunn, and Lingming Zhang. Agentless: Demystifying LLM-based software engineering agents. *arXiv preprint arXiv:2407.01489*, 2024.
- John Yang, Carlos E. Jimenez, Alexander Wettig, Kilian Liber, Karthik Narasimhan, and Ofir Press. SWE-agent: Agent-computer interfaces enable automated software engineering. *arXiv preprint arXiv:2405.15793*, 2024.
- Shunyu Yao, Dian Yu, Jeffrey Zhao, Izhak Shafran, Thomas L. Griffiths, Yuan Cao, and Karthik Narasimhan. Tree of thoughts: Deliberate problem solving with large language models. In *Advances in Neural Information Processing Systems (NeurIPS)*, 2023a.
- Shunyu Yao, Jeffrey Zhao, Dian Yu, Nan Du, Izhak Shafran, Karthik Narasimhan, and Yuan Cao. ReAct: Synergizing reasoning and acting in language models. In *International Conference on Learning Representations (ICLR)*, 2023b.
- Z.ai. GLM-5.1: Towards long-horizon tasks. Technical blog, <https://z.ai/blog/glm-5.1>, 2026.