Software Engineering KISS Sorcar with KISS Sorcar

Koushik Sen
EECS Department, UC Berkeley
ksen@berkeley.edu

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

Abstract

We describe how we built KISS SORCAR—a general-purpose AI assistant and integrated development environment—using KISS SORCAR itself. Over 44 days (April 22 to June 5, 2026), the developer issued 3,099 tasks through the system's own interface. We analyzed every task in the system's SQLite usage log and identified nine recurring patterns: test-first bug fixing, iterative feature construction, periodic code hygiene, paper–code co-evolution, defensive revert, architecture investigation before change, multi-channel general assistance, invariant specification and repair, and cross-model iterative review. The developer used KISS SORCAR not only to write code across Python, TypeScript, JavaScript, CSS, and LaTeX, but also to control smart-home lights via the Govee API, plan trips, send Slack and iMessage messages, check email, query weather and stock prices, build a website, prepare lecture slides, audit security reports, and refuse an unethical hacking request. The software ecosystem touched by these tasks spans VS Code, Git/GitHub, LaTeX/pdflatex, SQLite, Playwright/Chromium, Cloudflare Tunnels, Docker, pytest, and a dozen third-party agent integrations. We find that six established software engineering principles—encoded in a structured system prompt and a five-layer agent hierarchy totaling roughly 2,400 lines of code—explain the patterns and produce an AI assistant capable enough to build itself.

1 Introduction

Large language models generate code and call tools fluently. Deploying them as practical software engineering assistants, however, still exposes persistent gaps: finite context windows, sessions derailed by a single mistake, agents stuck in dead ends, and generated changes that resist review or revert Yang et al. [2024], Wang et al. [2024].

We built KISS SORCAR, an AI assistant and IDE, on top of the KISS Agent Framework Sen [2026]. The name reflects the *Keep It Simple, Stupid* design philosophy: each component does one thing, each concern lives in one place, and the overall system avoids unnecessary abstraction. The entire framework comprises roughly 2,400 lines of core agent code spread across five layers, each adding exactly one concern:

1. **KISS Agent**—budget-tracked ReAct loop Yao et al. [2023].
2. **Relentless Agent**—automatic continuation across sub-sessions via summarization.
3. **Sorcar Agent**—coding tools, browser automation, parallel sub-agents.
4. **Chat Sorcar Agent**—persistent multi-turn chat.
5. **Worktree Sorcar Agent**—git worktree isolation per task.

The defining feature of this project is that we built it with itself. From the second month onward, every code change, every bug fix, every feature addition, and every paper draft was produced by KISS SORCAR operating on its own repository. This *self-hosting* discipline turns the system into a

Table 1: Task categories identified in the 3,099-task usage log. Counts are approximate because tasks often span categories.

Category	Count	%
Bug finding, reproducing, and fixing	∼620	20
Feature implementation	∼520	17
UI/UX refinement	∼360	12
Paper writing and building	∼310	10
Code quality / refactoring	∼250	8
Testing and CI	∼210	7
Git operations	∼155	5
System prompt engineering	∼100	3
General assistance (weather, travel, shopping, stocks, research)	∼210	7
Communication (Slack, iMessage, email)	∼90	3
Smart-home control	∼50	2
Greetings and identity	∼150	5
Other (security audits, installation, architecture investigation)	∼150	5

continuous stress test: any bug the agent introduces impairs its own ability to function, and must be fixed immediately.

To understand how this process works in practice, we analyzed all 3,099 tasks recorded in the system's SQLite usage log (`~/.kiss/sorcar.db`) over 44 days of active development. The tasks fall into nine recurring patterns (Section 3), each grounded in a software engineering principle (Section 5). Table 1 summarizes the task distribution.

The rest of the paper is organized as follows. Section 2 surveys related work. Section 3 presents the nine usage patterns with concrete examples from the log. Section 4 zooms in on a single, complete development episode—redesigning the git-worktree merge workflow—to show the conversational explain-then-revise loop in detail. Section 5 maps each pattern to a software engineering principle. Section 6 catalogs the software ecosystem. Section 7 shows how the principles are encoded in the system prompt. Section 8 discusses lessons learned. Section 10 concludes.

2 Background and Related Work

2.1 AI Coding Agents

SWE-Agent Yang et al. [2024] and OpenHands Wang et al. [2024] provide LLM-based agents for resolving GitHub issues in single sessions. Agentless Xia et al. [2024] shows that a two-phase localize-then-repair pipeline can match agentic approaches on SWE-bench Jimenez et al. [2024]. Industrial products—GitHub Copilot GitHub [2021], Cursor Cursor [2024], Devin Cognition Labs [2024], Claude Code Anthropic [2025], OpenAI Codex OpenAI [2025], and Aider Gauthier [2023]—offer varying degrees of autonomy. Cursor's Composer 2 Cursor [2026] uses large-scale reinforcement learning on a custom fine-tuned model. Multi-agent systems such as ChatDev Qian et al. [2024], MetaGPT Hong et al. [2024], and AutoGen Wu et al. [2023] simulate organizations of specialist roles.

These agents rest on a body of work that taught language models to reason and act. Chain-of-thought prompting Wei et al. [2022] showed that eliciting intermediate reasoning steps improves problem solving, and ReAct Yao et al. [2023] interleaved that reasoning with tool calls—the loop at the core of our KISS Agent. Toolformer Schick et al. [2023] demonstrated that models can learn when and how to invoke external APIs, while Reflexion Shinn et al. [2023] added verbal self-reflection so an agent can learn from its own failed attempts within a session, a behavior our Relentless Agent reproduces through summarized continuation. Voyager Wang et al. [2023] accumulated reusable skills in an open-ended environment, echoing how KISS SORCAR grows its own tooling. The coding ability these agents exploit comes from code-trained foundation models such as Codex Chen et al. [2021] and Code Llama Rozière et al. [2023]. A complementary line of work treats prompting itself as a program to be optimized: DSPy Khattab et al. [2024] compiles declarative language-model pipelines

into tuned prompts. KISS SORCAR instead keeps a single hand-written, human-readable system prompt, trading automatic optimization for transparency and direct editability.

Our approach differs. Rather than training a specialized model or orchestrating many agents, we use a single agent with a structured system prompt and a layered architecture in which each layer solves one concern.

2.2 Software Engineering Principles

The principles we apply are not new. Parnas Parnas [1972] established information hiding and modular decomposition. Dijkstra Dijkstra [1968] argued for structured programming. Brooks Brooks [1995] warned against accidental complexity. Beck Beck [2003] codified test-driven development. Hunt and Thomas Hunt and Thomas [1999] distilled pragmatic engineering heuristics. Martin Martin [2008] catalogued clean-code practices. Gamma et al. Gamma et al. [1994] catalogued reusable design patterns, and the Agile Manifesto Beck et al. [2001] reframed software development around short feedback loops and responsiveness to change—the same rhythm as our task-by-task conversational workflow. Lehman's laws of software evolution Lehman [1980] observed that a system in use must continually change or become progressively less useful, which is precisely the pressure a self-hosting agent places on itself.

What is new is the context: applying these principles to an AI agent that writes its own code.

2.3 Self-Hosting in Software

Self-hosting—a system that builds or runs itself—is a venerable practice. Thompson Thompson [1984] explored the trust implications of self-compiling compilers. Raymond Raymond [1999] observed that eating your own dog food exposes bugs that no amount of external testing reveals. We apply this idea to an AI agent: KISS SORCAR develops itself, and every defect it introduces is a defect it must then live with.

3 Usage Patterns

We examined all 3,099 tasks in the usage log and identified nine recurring patterns. Each pattern is a workflow that the developer repeated dozens or hundreds of times.

Methodology. The data source is the system's own SQLite database, `~/.kiss/sorcar.db`, which the Chat Sorcar Agent (layer 4 of the hierarchy in Section 1) populates automatically after every task. The schema is intentionally small: a `task_history` table stores one row per task (timestamp, the verbatim task description, the result summary, a 32-character `chat_id`, and a JSON `extra` field of metadata such as model name, token count, and dollar cost); an `events` table stores the full per-step trajectory; a `frequent_tasks` table maintains a frequency count per distinct task string; and `model_usage` and `file_usage` tables track which models and files were touched. Because every task the developer issued—whether a coding request, a weather query, or a smart-home command—flows through the same Chat Sorcar Agent, the log is a complete, unfiltered record of how the system was actually used. The 3,099 tasks span 44 days and group into 877 distinct chat sessions, an average of roughly 3.5 tasks per session, confirming the iterative, multi-turn nature of the interaction (Section 3.2). We extracted patterns by three complementary methods: (i) clustering tasks by the `frequent_tasks` frequency table, which surfaces verbatim repeated commands such as "`uv run check --full and fix`" (50 occurrences) and "`git push origin`" (13 occurrences); (ii) full-text `LIKE` queries over the task descriptions to count phrasings (*e.g.* 73 tasks contain "build the paper," 105 contain "`uv run check`"); and (iii) manual reading of representative tasks within each cluster to recover the underlying workflow. The category counts in Table 1 are approximate because a single task often serves more than one purpose—"fix the race condition and run the full suite" is simultaneously a bug fix and a testing task. Task numbers cited below (*e.g.* Task 344) refer to the `task_history` row identifiers, which increase monotonically with time and therefore double as a rough chronological index.

3.1 Pattern 1: Test-First Bug Fixing

The most common pattern (roughly 20% of all tasks) follows a rigid three-step protocol:

1. The developer reports a bug in natural language.
2. The agent writes an integration test that reproduces the bug.
3. The agent fixes the root cause and verifies the test passes.

The canonical phrasing, which appears in hundreds of tasks, is:

> "Can you reproduce the bug by writing an integration test? Then fix it."

This phrasing is not a coincidence; it is the conversational echo of an instruction baked into the system prompt itself (Section 7): "When fixing bugs, issues, or race conditions: write an integration test that reproduces the problem first, then fix the code, then verify the test passes." The developer internalized the agent's own protocol and began phrasing bug reports in the form the agent handles best. The discipline serves two purposes that the developer relied on repeatedly. First, a test that fails before the fix and passes after it is *evidence* that the reported bug was real and that the change addressed it, rather than a plausible-looking edit that touches the wrong code path. Second, writing the test forces the agent to localize the defect precisely—to identify the exact module, function, and input that triggers the failure—before it is allowed to modify anything, which curbs the model's tendency to patch symptoms.

Example. Task 1236: "`ask_user_question` is not showing its window in the web/mobile app." The agent was instructed to write a test first, then fix. The bug reports in the log are strikingly specific, which is what makes the test-first protocol tractable. Representative verbatim examples include: "In the last task, the merge/diff interface was not launched and Auto-commit and merge buttons were not shown. Reproduce the issue . . . then fix it"; "In the last task, you made changes, but they were not reflected in the actual file system. Can you find and reproduce that. . . "; and "the settings are not persisted across tasks. Repro the issue by writing an integration test." This protocol was applied to bugs as varied as missing diff/merge UI triggers for LaTeX files (Task 109), auto-committed changes not appearing in reports (Task 104), thinking blocks leaking outside thought panels (Task 741), and race conditions in concurrent tab management (Task 65).

Concurrency bugs. A distinctive sub-class of test-first tasks targets race conditions that only the self-hosting setting exposes, because the developer runs multiple IDE tabs and a remote web server against the same repository at once. Examples from the log include "Fix the persistence module's SQLite connection handling with a read-write lock on `_load_chat_context` to eliminate concurrent-access errors," "Add integration tests for nested parallel execution where sub-agents themselves invoke `run_parallel`, to verify budget . . . is correct," and a report of errors "when multiple tasks from local extension and from remote web server are using" the database simultaneously. For these, the system prompt prescribes a specific technique the developer invoked by name: "To confirm race conditions: add a random sleep (<0.1s) before the suspected racing statements," which widens the race window so the bug manifests deterministically in a test before the lock or thread-local fix is applied.

Variant: defensive revert. When a fix introduced a regression, the developer's next task was often "can you get rid of the last commit to main?" (Tasks 18, 56, 110, 209, 395, 545, 909, 920, 949, 1877, 1996). The exact string "get rid of the last commit" (with minor variations such as ". . . to main," ". . . from main," ". . . from main and origin," and ". . . in main?") recurs more than 30 times in the log, making it one of the most frequent recovery commands. It functions as a manual rollback mechanism that complements the Worktree Sorcar Agent's branch-per-task isolation: even though each task runs on its own branch and the user reviews the diff before merging, a regression can still slip through review and land on `main`. When that happens, the developer's reflex is to undo first and re-attempt with a fresh approach, rather than to pile a corrective patch on top of a broken commit—a manual application of the defensive-programming principle discussed in Section 5.4.

3.2 Pattern 2: Iterative Feature Construction

Features were never built in a single task. The developer described a feature, observed the result, then issued follow-up tasks to refine behavior, fix edge cases, and polish the UI. A single feature

could span 5–20 tasks. This pattern is the dominant explanation for the 877 chat sessions in the log: a feature is typically developed within one chat session, and the Chat Sorcar Agent's persistence lets each follow-up task reference the prior tasks and results in the same session without the developer re-stating context. The workflow is therefore a tight loop of *describe* → *observe* → *refine*: the developer issues a short natural-language request, watches the agent's result in the IDE, and immediately issues the next task to correct whatever fell short. Crucially, the refinement is interleaved with Pattern 1: roughly half the follow-up tasks in a feature session are themselves bug reports ("the button does not appear on mobile"), so a feature session is usually a chain of small features punctuated by test-first fixes.

Example: remote web server. The remote-access feature started with Task 344:

> "Can you implement a new feature where the user can access KISS Sorcar from the internet from a mobile device or any device with a browser?"

This was followed by 60+ tasks over three days: adding HTTPS/TLS (Task 374), fixing mobile layout (Task 452), adding password authentication (Task 878), displaying the Cloudflare tunnel URL in the welcome page (Tasks 396–415), handling MacBook lid-close disconnections (Task 412), and emailing the URL to the user (Tasks 386–392). The trajectory of this single feature is a microcosm of the whole pattern: the first task delivered a minimal working server, and every subsequent task hardened one concrete dimension—security (TLS, then password auth), reach (mobile layout), discoverability (URL on the welcome page, URL by email), and robustness (surviving a lid-close disconnection). None of these refinements was specified up front; each was prompted by the developer actually using the feature on a phone and hitting its limits. This is iterative construction in the literal sense: the specification emerged from use, not from a design document.

Example: demo mode. Demo mode started with Task 45 and was refined over Tasks 46, 53, 191, 219–220 across multiple days, adding streaming animation, stop-button behavior, and cross-task continuation. A representative refinement task reads: "In the demo mode, I see the welcome page when the demo switches from one task to another. The next task animation MUST" begin without the intervening welcome screen—a single-sentence bug report that triggered a targeted fix to the demo state machine. The multi-day span (Tasks 45 through 220) shows that a feature is not "done" when it first works; it is revisited whenever the developer notices a rough edge during ordinary use.

3.3 Pattern 3: Periodic Code Hygiene

At regular intervals—roughly every 20–30 tasks—the developer ran a sweep:

- "`uv run check -full and fix`" (Tasks 26, 60, 72, 73, 83, 203, 212, ...). The exact string "`uv run check --full and fix`" is the second most frequent command in the entire log (50 occurrences in the `frequent_tasks` table), and 105 tasks in total mention "`uv run check`."
- "Can you thoroughly find all bugs, redundancies, and inconsistencies in `PWD/src/kiss/agents/vscode/`?" (Tasks 1, 6, 15, 17, 19, 23, 153, 425–427, 716, 918, 978, 981).
- "Run all tests and fix tests only if there is no bug in the code. Report the bugs." (Tasks 798, 840, 866, 890–891, 961, 986, 990, 1013, 1126, 1128, 1158). A closely related verbatim phrasing—"can you run all tests and report the cause of failing tests? Do not modify any" code—appears 22 times.

These sweeps function as a periodic "garbage collection" for code quality. They catch drift introduced by feature work, remove dead code left behind by refactors, and ensure the test suite stays green. Three observations are worth drawing out. First, the sweeps are deliberately *decoupled* from feature work: rather than demanding a green lint and a passing suite after every edit (which would slow the describe–observe–refine loop of Pattern 2), the developer lets small inconsistencies accumulate and then clears them in a dedicated pass. This mirrors the system prompt's own division of labor, which asks the agent to run impacted tests after a change but reserves the full `uv run check --full` sweep for the pre-finish verification stage. Second, the "find all bugs, redundancies, and inconsistencies" phrasing invokes the File Browsing protocol: the agent surveys an entire directory, writes a structured inventory of issues, and only then proposes fixes—an explicit collect-then-act discipline that prevents premature, locally-scoped patches. Third, the qualifier "fix tests only if there

is no bug in the code—report the bugs" is a guardrail against a well-known failure mode in which an agent makes a failing test pass by weakening its assertions rather than by fixing the underlying defect. By instructing the agent to *report* rather than silently "fix" a test, the developer keeps the test suite an honest oracle.

3.4 Pattern 4: Paper–Code Co-Evolution

The system prompt and the paper describing it evolved together. A change to the system prompt triggered a paper update, and a paper revision sometimes triggered a code change:

- "`SYSTEM.md` has changed. Update and build the paper." (Tasks 84, 167, 233, 261, 274, 312, 665, 693, 766, 927, 931, 1007).
- "Can you check the consistency and correctness of the paper against itself and the latest code?" (Tasks 169, 694, 824, 985, 1009).
- "I have fixed typos in the system prompt in the paper. Can you update `SYSTEM.md`?" (Task 700).

This bidirectional flow means the paper is always a faithful description of the system, and the system always reflects the principles claimed in the paper. The phrase "build the paper" appears in 73 tasks, making paper maintenance one of the highest-volume non-coding activities in the log; the LaTeX toolchain (`pdflatex`, BibTeX) is consequently one of the most-exercised pieces of the software ecosystem catalogued in Section 6.

The co-evolution is genuinely bidirectional, and the log shows both directions. In the code-to-paper direction, a change to `SYSTEM.md` (the system prompt) triggers a task of the form "`SYSTEM.md` has changed—update and build the paper," so that the paper's system-prompt section never drifts from the deployed prompt. In the paper-to-code direction, Task 700—"I have fixed typos in the system prompt in the paper. Can you update `SYSTEM.md`?"—uses the paper as the source of truth and propagates an edit back into the running system. The consistency-audit tasks ("check the consistency and correctness of the paper against itself and the latest code") close the loop: because the same agent can read both the LaTeX source and the code it describes, a divergence between claim and implementation is mechanically detectable rather than relying on the author's memory. This is the self-hosting principle (Section 5.6) applied to documentation: the artifact that describes the system is maintained by the system, so it stays accurate as a side effect of ordinary development.

3.5 Pattern 5: Architecture Investigation Before Change

Before making non-trivial changes, the developer asked the agent to explain the current architecture:

- "Can you show me the detailed step-by-step workflow of `web_server.py`?" (Task 444).
- "What happens when two tasks are run simultaneously?" (Tasks 3, 5, 7).
- "How does the frontend and backend interact when a task is running?" (Task 1818).
- "Can you create a graph diagram showing how the server-side files are related?" (Tasks 1024–1027).

This "understand before you change" discipline prevents blind modifications. The agent reads the code, produces a step-by-step explanation, and the developer uses that understanding to formulate a precise change request. The pattern is so habitual that several investigation prompts recur verbatim: "can you explain step-by-step in details of the worktree workflow?" and "can you tell me in very details what happens when a task with sub agents is load"ed both appear multiple times in the `frequent_tasks` table.

The investigation step pays off precisely because the agent's explanation and the agent's subsequent edits are produced by the same reader of the same code. When the developer asks "what happens, step by step, with git in `worktree_sorcar_agent.py` when a task finishes?" and then, in the next task, asks for a redesign of that lifecycle, the redesign is grounded in the explanation the agent just generated—there is no translation loss between a human's understanding of a design document and the code that implements it. This is the conversational form of the system prompt's pre-flight "read before you write" rule (Section 5.5): the developer front-loads comprehension into an explicit question-and-answer turn, so the change request that follows can be stated as a precise, local edit rather than a vague aspiration. The graph-diagram requests (Tasks 1024–1027) are a visual variant

of the same impulse: before reorganizing the server-side modules, the developer asks the agent to render their dependency structure, turning an implicit mental model into an inspectable artifact.

3.6 Pattern 6: System Prompt Engineering

About 100 tasks were devoted to refining the system prompt itself:

- "Can you check `SYSTEM.md` and tell me if you understand all instructions? How can they be improved?" (Tasks 972–974, 1003–1006).
- "You are not visiting at least 30 websites when you need to search. Fix `SYSTEM.md`." (Task 757).
- "You are creating too many temporary files. Improve `SYSTEM.md`." (Task 1416).
- "Can you look at yesterday's tasks and check which instructions are not being followed? Fix them." (Tasks 1267, 1269, 1271).

The developer treated the system prompt as production code: observing the agent's behavior, identifying deviations from the specification, and patching the prompt accordingly. This is the prompt-engineering equivalent of a test-driven development cycle.

What makes this pattern distinctive is that the test data is the usage log itself. Because every task and its full trajectory are persisted in `sorcar.db`, the developer could ask the agent to mine its own recent history for compliance failures: "Look at yesterday's tasks and check which instructions in `SYSTEM.md` are not being followed." The agent reads the `events` table for the relevant tasks, detects deviations (*e.g.* a research task that fetched pages with `curl` instead of `go_to_url`, or a code change that skipped `uv run check`), writes an integration test with a real LLM call to confirm the deviation reproduces, and only then patches the prompt. This is mutation testing applied to a specification: the developer checks whether the prompt is strong enough to prevent an *observed* failure, not merely a hypothetical one. The verbatim audit prompt—"can you check `PWD/src/kiss/ SYSTEM.md` and tell me if you understand all the instructions"—recurs as a periodic self-review. The fixes that resulted are visible in the deployed prompt today: the "temporary files in `PWD/tmp/`" rule, the explicit `curl`/`wget` prohibition in the web-research section, and the enumerated file extensions that trigger the mandatory `uv run check` were all added reactively after the log revealed the corresponding evasions.

3.7 Pattern 7: Multi-Channel General Assistance

Beyond software engineering, the developer used KISS SORCAR as a general-purpose assistant:

- **Smart home:** "Turn off living room and family room lights" via the Govee API (Tasks 1181–1219).
- **Travel:** "Plan me a trip to Yosemite" with hotel availability checks (Tasks 911, 922–923, 926, 930, 932, 937).
- **Weather:** "How is the weather today?" (Tasks 359, 365, 405, 422, 698, 727–728, 767, 875–876).
- **Shopping:** "Where can I find Darjeeling tea in the Bay Area?" (Tasks 789–790), "Where should I buy Louis Vuitton bags?" (Task 774).
- **Finance:** "Will Microsoft stock increase by August 2026?" (Tasks 333, 337), ETF analysis (Tasks 777–778, 1111–1112, 1123, 1178–1180, 1222).
- **Communication:** Slack messages (Tasks 299–301, 913–915, 1015–1017, 1188–1189), iMessage (Tasks 550–551, 1184–1185, 1210), Gmail (Tasks 1191–1198, 1233).
- **Ethical refusal:** an unethical hacking request, which the agent refused, offering five legitimate alternatives (Task 1240).

This pattern demonstrates that the same system prompt and architecture serve both engineering and non-engineering tasks. The KISS principle—one agent, one prompt, one architecture—scales across domains.

The mechanism that makes this possible is the third-party agent interface (Section 6): smart-home control, Slack, iMessage, and Gmail are all Python modules in `kiss/agents/third_party_agents`, and the Sorcar Agent invokes them through the same uniform tool interface it uses for `Bash`, `Edit`, and `go_to_url`. No special routing, planner, or domain classifier sits in front of them; the model selects the right tool from the task description alone. The log even records the bootstrapping of these integrations as ordinary tasks—"can you authenticate with govee.com so that you can control the

lights?" and "do whatever is necessary to control the lights of govee.com"—after which day-to-day commands such as "use `govee.py` to turn off family room lights" (which appears repeatedly in the `frequent_tasks` table) just work. The system prompt's own override section anticipates this: "Authenticate unauthenticated third-party agents; ask the user only when a page requires human authentication," so the agent provisions its own credentials and only interrupts the developer for a genuine human-in-the-loop login.

The same rigor applies off-task. Notably, the non-coding tasks inherit the engineering discipline. A weather or stock query triggers the Web Research protocol—the agent visits multiple sources, records them in a `tmp/information-{id}.md` file, and is forbidden from answering current-events questions from stale training data. A research request such as "Research these three topics and summarize each…" (a recurring multi-topic prompt in the log) is handled with the same collect-then-synthesize rigor as a code investigation. Even the ethical refusal in Task 1240 is an application of the defensive-programming clause (Section 5.4): the agent declined an unethical hacking request and offered five legitimate alternatives, treating "refuse what you cannot do safely" as a first-class behavior rather than an exception. The lesson of Pattern 7 is therefore not merely that one agent can do many things, but that the *same* engineering safeguards—verify sources, isolate changes, refuse unsafe requests, never fabricate—apply uniformly whether the task is a code fix or a dinner reservation.

3.8 Pattern 8: Invariant Specification and Repair

A distinctive workflow that recurs throughout the log is one in which the developer states a global *invariant*—a property that *every* part of the codebase must satisfy—and then asks the agent to find and fix every place that violates it. The invariant is expressed in a single English sentence; the search and repair span the entire repository. The canonical phrasing has the form "*X must be Y everywhere; find and fix all violations.*"

Representative invariants drawn verbatim from the log include:

- *File-system invariants:* "All temporary files MUST live under `PWD/tmp/`; find and fix every place that writes a temporary file directly to `PWD/` or to the project root."
- *Module-boundary invariants:* "All third-party integrations MUST live in `kiss/agents/third_party_agents`; relocate any that do not, and update all imports."
- *Tool-discipline invariants:* "All file reads in the codebase and in tests MUST go through the `Read()` tool wrapper, never raw `open()` for source inspection; flag every violation."
- *Test-discipline invariants:* "No test in the suite may use mocks, patches, fakes, or test doubles; identify every offending test and rewrite it as a real end-to-end test."
- *Concurrency invariants:* "Every SQLite write to `sorcar.db` MUST be guarded by the persistence module's read-write lock; locate and fix any direct connection that bypasses it."
- *Lifecycle invariants:* "Every background subprocess spawned by an agent MUST be terminated when its parent task finishes; find any code path that can leak a process and fix it."

The verbatim audit prompt that most often opens this pattern is "can you thoroughly find all bugs, redundancies, and inconsistencies" (Pattern 3), but the invariant variant differs in intent: Pattern 3 asks for an open-ended quality sweep, whereas Pattern 8 specifies a single, precise property and demands closure.

Why this works. The invariant pattern leverages a strength specific to LLM agents that is hard to exploit in a manual workflow. A human enforcing a repository-wide invariant typically reaches for `grep` or a custom lint rule, both of which require the invariant to be expressible as a syntactic pattern. The agent, by contrast, can apply a *semantic* filter: it reads each candidate site, decides whether the site actually violates the invariant in context, and edits only the genuine violations. The system prompt's File-Browsing protocol governs the search phase verbatim: the agent collects every candidate site into a `PWD/tmp/file-information-{id}.md` inventory, reviews the collected material as a single artifact, and only then proposes fixes. This collect-then-act discipline is what prevents the well-known failure mode in which a global search-and-replace silently breaks an unrelated call site that happens to share a syntactic shape.

Example. Task 1416 in the log states an invariant about temporary files: *every* temporary file created by the agent must reside under `PWD/tmp/`, never in the project root. The agent ran the inventory phase, found multiple call sites in tests and in the research protocol that wrote scratch files into `PWD/`, fixed them, and—closing the invariant loop—added a clause to `SYSTEM.md` requiring the property to hold for *future* edits as well. The pattern thereby produces two artifacts: a one-time repair sweep and a permanent specification clause that prevents the next agent session from regressing. Pattern 8 is therefore the natural complement to Pattern 6 (system prompt engineering): the developer first observes a class of violations in the log, then states the invariant that excludes them, then asks the agent to repair the past and amend the prompt against the future.

3.9 Pattern 9: Cross-Model Iterative Review

A second distinctive workflow exploits a feature of the `update_settings` tool exposed by the KISS Agent: the running agent can switch its own underlying LLM mid-session via the `model_name` setting. The developer uses this to perform *cross-model code review*. A task is first executed under the primary model (typically `claude-sonnet`); the developer then issues a follow-up task that switches to a different model—most often `gpt-5.5`—and asks the new model to review the prior change, find bugs, and propose fixes. The loop iterates until the reviewing model reports a clean pass.

The canonical phrasing recurs in the log as variants of:

> "Switch to `gpt-5.5` and review the changes from the last task. Find any bugs, regressions, or violations of `SYSTEM.md`. Write integration tests for any defect you find, then fix it. Iterate until the suite is clean and you have nothing more to report."

The mechanism is a single agent that swaps its weights between tasks, not two agents in a debate: the Chat Sorcar Agent preserves the conversation, the Worktree Sorcar Agent keeps the review on the same branch as the original change, and the KISS Agent tracks the cumulative cost across models in the sidebar so the developer sees the price of the second opinion in real time.

Why two models help. Cross-model review is not redundant when the two models share a common training distribution: it is a practical form of *ensemble disagreement*. A bug that one model introduces—an off-by-one in a slice, an unhandled error path, a stale import left behind after a refactor—is by construction one the model failed to detect; a model with a different training mixture and different inductive biases is statistically more likely to catch it. The pattern repurposes a model-evaluation insight for development: disagreement between models is a noisy but useful signal of latent defects, and the explain-then-revise loop of Section 4 gives the developer a low-cost way to act on that signal without leaving the chat. The discipline also doubles as a guard against the well-known failure mode in which an LLM blesses its own output: when the same model both writes and reviews a change, it is biased to confirm rather than falsify. Forcing the reviewer to be a different model breaks that loop.

Example. Tasks of the form "switch to `gpt-5.5` and audit the worktree merge changes from the previous task" appear repeatedly after major refactors of safety-critical code (*e.g.* the merge-workflow redesign in Section 4). A representative session proceeds as follows: (i) the primary model lands a multi-file edit and reports all tests green; (ii) the developer issues "`set_model gpt-5.5`; now re-read the diff and write a failing test for any bug you can find"; (iii) the reviewer finds, say, an unhandled `None` return on a rarely-taken code path and reproduces it in a new test; (iv) the developer asks the reviewer to fix it, runs the now-larger suite, and either accepts the fix or rolls the round back via the defensive-revert variant of Pattern 1. The loop is typically iterated two or three times before the reviewing model stops finding defects. The cost of a review round is bounded by the budget tracker, so the developer can cap a cross-model audit at a fixed dollar amount, just as one caps a CI run at a fixed wall-clock budget. Pattern 9 thereby brings to a single-developer workflow the adversarial-review discipline that distributed teams obtain through pull-request review—without requiring a second human reviewer.

4 Iterative Development: A Detailed Case Study

Patterns 2 and 5 (Sections 3.2 and 3.5) describe, in the aggregate, how the developer evolves the system through conversation rather than through manual code editing. In this section we zoom in

on a single, complete development episode to make that workflow concrete. The episode is drawn verbatim from the project's own usage log and is representative of how the bulk of KISS SORCAR was built: the developer *never opened a source file, never wrote a line of code, and never ran a test by hand*. Every step—understanding the design, redesigning it, diagnosing an anomaly, and fixing it—was a single natural-language task.

The explain-then-revise loop. The workflow has a characteristic two-move shape that recurs throughout the log. First, the developer asks KISS SORCAR to produce a *detailed, step-by-step description* of a workflow or algorithm that is suspected to be buggy or overly complex. Second, having read that description, the developer asks KISS SORCAR to *revise specific steps*, stated in plain English ("delete the unstage step," "ask only two questions instead of three"). The first move externalizes the agent's understanding of the code into an inspectable artifact; the second move edits that understanding back into the code. The crucial property—and the reason this loop is so much more reliable than editing a design document by hand—is that the same agent, reading the same source, produces both the explanation and the subsequent edits. There is no translation loss between a human's mental model of a design document and the code that implements it: the explanation *is* a faithful read of the code, and the edit is applied to exactly the code that was just explained.

We illustrate the loop with the git-worktree merge workflow implemented in `worktree_sorcar_agent.py`—the outermost layer of the hierarchy in Section 1, which isolates every task on its own branch. This subsystem is a fitting example precisely because it is the one the developer interacts with on *every* task: when a task finishes, the developer must decide whether to merge the agent's branch back into `main` or discard it, so any rough edge in the merge logic is felt hundreds of times a day. The episode comprises six consecutive tasks within a single chat session; we reproduce the prompts verbatim and summarize the agent's responses.

Step 1: Eliciting the detailed workflow. The developer begins not with a change request but with a request for an explanation, applying the "read before you write" discipline (Section 5.5) as an explicit conversational turn:

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

The agent reads the source and returns a structured account of the four-phase post-task lifecycle: (1) during `run()`, a fresh branch and a corresponding worktree directory are created—the branch name encodes a timestamp and a random suffix for uniqueness—and the task executes *inside* the worktree, so the user's main working tree is never touched; when the task completes, *nothing is committed or merged*—the result is returned with merge instructions appended and the worktree is left pending; (2) `merge()` calls `_finalize_worktree()`, which stages all changes, generates a commit message with an LLM call over the staged diff, commits inside the worktree, removes the worktree, checks out the original branch, and replays the change onto it; (3) `discard()` removes the worktree, prunes it, checks out the original branch, and deletes the task branch; (4) in CLI mode an interactive prompt forces the user to choose between `[c]ommit and merge`, a `[m]anual` merge, and `[d]iscard` before exiting. The agent also surfaces two non-obvious invariants that turn out to be central to the redesign that follows: when the user's working tree was dirty at task start, the dirty state is snapshotted into a *baseline commit* so the agent sees the same files as the user, and the merge therefore uses `git cherry-pick --no-commit baseline..branch` (rather than `git merge --squash`) so that only the agent's own commits are replayed and the dirty-state snapshot is excluded. Finally, the agent notes that all worktree state lives in git itself (branch names and git config), so the lifecycle is idempotent and recoverable after a crash.

This single answer is the artifact the rest of the episode operates on. It is not a comment, a design document, or a wiki page that might drift out of date; it is a read of the live source, generated on demand, and it exposes exactly the design decisions (the three-way CLI choice, the baseline-commit detour) that the developer will next decide to simplify.

Step 2: Revising a step in natural language. Having seen the workflow spelled out, the developer judges the three-way choice (auto-merge / manual merge / discard) to be needless complexity and dictates a simpler one:

```
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.
```

This is a single English sentence describing a behavioral change, yet it ranges across two languages and the full stack. The agent applies the Complex-Task-Planning discipline from the system prompt (Section 7)—enumerating the files to touch before editing any of them—and then modifies six files: it rewrites `discard()` to check out the original branch *before* deleting the task branch (deleting a checked-out branch fails); it removes the `manual_merge()` method entirely; it simplifies `merge_instructions()` to present only two options; it replaces the three-button webview toolbar with a two-button "Commit and Merge or Discard?" bar in the TypeScript frontend; it deletes the `manual` action type from the TypeScript type definitions; and it removes the corresponding handler from the Python backend. The agent then updates the test suite to match: three tests covering the now-deleted manual-merge path are removed and one routing test is revised. It runs the impacted tests and reports that all 28 worktree tests pass. The developer reviewed the diff in the worktree and merged it—without ever opening an editor.

Step 3: Diagnosing an anomaly with a second explain turn. After exercising the simplified flow, the developer notices behavior that contradicts the mental model from Step 1: files appear in VS Code's Source Control panel *on the original branch* after pressing "Commit and Merge." Rather than guessing at a fix, the developer issues another *explain* request—the loop's first move applied to a narrower question:

```
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 the squash-merge routine and discovers that the implementation deliberately *unstages* the squash-merged changes with `git reset HEAD`, so that they surface as uncommitted working-tree modifications in the Source Control panel. The original design intent was to let the user review the diff one more time and commit it manually on the original branch. The agent reports this finding, quotes the responsible line, and includes a verified end state showing no *new* commits on the branch—confirming the developer's observation and explaining its cause. Note that this diagnostic turn touched no code: it is purely the collect-then-reason discipline (Section 5.5) applied to a runtime puzzle, and it produces the precise fact—"the unstage is intentional and lives in one `git reset HEAD` call"—that the next task will act on.

Step 4: Directing the fix in one sentence. Armed with the explanation, the developer realizes the design premise no longer holds: review already happened in the worktree branch before the merge, so the post-merge unstage-for-review step is redundant. The fix is dictated in two sentences:

```
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 with a conditional `git commit --no-edit` that reuses the auto-generated squash commit message, and—anticipating an edge case the developer did not mention—adds a `git diff --cached --quiet` guard so that a merge producing no staged changes does not attempt an empty commit. It renames the now-misnamed test `test_merge_leaves_changes_uncommitted` to `test_merge_commits_changes` and rewrites its assertions to verify a *clean* working tree after merge. It runs the suite and reports that all worktree tests pass.

Step 5: Hardening the dirty-state edge case. A subsequent task in the same session asks the agent to re-examine the merge under the dirty-working-tree condition that the baseline commit was designed to handle:

```
What happens to my uncommitted changes if I start a task while the
working tree is dirty and then click "Commit and Merge"?
```

The agent re-reads the merge path and explains the baseline-commit mechanism in operational terms: the dirty files are captured in a baseline commit at task start so the agent operates on the same files the developer sees, and the merge replays only the commits between the baseline and the branch tip via `git cherry-pick --no-commit baseline..branch`. The consequence—which the agent states explicitly—is that the developer's pre-existing uncommitted edits are *not* swept into the agent's commit; they remain as uncommitted changes in the user's tree, exactly as they were before the task ran. This is the explain move again, and it confirms that the simplification in Steps 2–4 preserved the dirty-state invariant rather than silently breaking it—a verification that, in a manual workflow, would have required the developer to construct a dirty-tree scenario and inspect the git history by hand.

Step 6: Serializing concurrent merges. Because the developer routinely runs several IDE tabs and a remote web server against the same repository, a final task probes what happens when two tabs finish and merge at the same time:

```
What happens if two tabs both press "Commit and Merge" on the same repo
at the same time? Can the checkout/merge steps interleave?
```

The agent explains that the checkout–cherry-pick–commit sequence is serialized by a per-repository file lock (`repo_lock`), so concurrent tabs cannot interleave operations on the same repository, and that per-task state (stream buffers, bash output, recording state) is held in thread-local storage so that stopping one task cannot corrupt another. Satisfied that the redesigned, simpler workflow retained the concurrency guarantees of the original, the developer closed the session. This pair of closing explain-only turns (Steps 5 and 6) illustrates a subtle but important habit: after a simplification, the developer uses the same explain move not to drive a change but to *audit* that the change did not regress an invariant the simpler code still needed to honor.

Discussion. The six-task episode is a microcosm of how KISS SORCAR was built. Three properties of the system make this conversational loop possible, and each maps to a layer of the architecture (Section 1). First, the **Worktree Sorcar Agent** ran every one of the six tasks on its own throwaway branch, so the developer could redesign a safety-critical merge routine without any risk to `main`: a botched edit in Step 2 or 4 would simply be discarded. Second, the **Chat Sorcar Agent** preserved the conversation across all six tasks, so the redesign request in Step 2 could refer to "the workflow" explained in Step 1, and the fix in Step 4 could refer to the anomaly diagnosed in Step 3, without the developer restating context. Third, the **Relentless Agent** would have continued any single task that exhausted its context window—a multi-file Python plus TypeScript edit with test updates is large enough that this matters in practice. The **KISS Agent** underneath tracked the dollar cost of each turn, so the developer saw the price of the whole episode accumulate in the sidebar.

The deeper lesson concerns the *explain-then-revise* loop itself. Conventional wisdom holds that natural language is too imprecise a medium for specifying software changes. This episode suggests the opposite when the explanation and the edit are produced by the same agent over the same source. The developer's change requests in Steps 2 and 4—"ask two questions instead of three," "you don't need the user to review in the original branch"—are vague by the standards of a formal specification, yet they produce correct, multi-file, multi-language edits because they are *grounded in an explanation the agent itself just generated*. The agent does not have to guess what "the unstage step" refers to; it explained that step two tasks earlier and can locate the exact `git reset HEAD` call. This grounding is what distinguishes the loop from prompting a stateless code generator: the explanation turn is not a courtesy to the human, it is a shared, code-faithful referent that makes the subsequent imprecise instruction unambiguous. Finally, the episode shows that the explain move serves two distinct purposes—comprehension before a change (Steps 1 and 3) and verification after one (Steps 5 and 6)—so that the same lightweight conversational primitive both opens and closes the development loop.

5 Principles Behind the Patterns

Each usage pattern maps to one or more established software engineering principles.

5.1 The KISS Principle

Keep it simple. The five-layer hierarchy embodies this: 2,421 lines total, an order of magnitude smaller than comparable systems. Pattern 7 (multi-channel assistance) shows the principle at the usage level: one system handles coding, smart home, travel, and communication without specialized sub-systems.

When the user typed "hi" (Tasks 207, 272, 278, 285, 289, 416, 421, 443, 454, . . . —roughly 150 occurrences), the agent responded with a single friendly sentence. It did not launch into a capabilities description. The KISS principle applied to interaction design says: match the weight of the response to the weight of the input.

5.2 Separation of Concerns

Each module addresses a single concern. Pattern 2 (iterative feature construction) relies on this: the remote web server feature touched the Sorcar Agent (tool integration), the Chat Sorcar Agent (persistence), and the Worktree Sorcar Agent (git isolation), but each change was scoped to one layer.

The smart-home example (Pattern 7) exercised separation at multiple levels: the Chat Sorcar Agent loaded conversation context, the Sorcar Agent selected the Govee tool, and the KISS Agent tracked the budget—each layer unaware of the others' concerns.

5.3 Test-First Development

Write a failing test, then make it pass. Pattern 1 (test-first bug fixing) is the direct application. The phrase "write an integration test to reproduce the issue, then fix it" appears in hundreds of tasks. The system prompt reinforces this with: "Do not use mocks, patches, fakes, or test doubles."

Pattern 3 (periodic code hygiene) extends this to maintenance: running the full test suite after every batch of changes ensures that the cumulative effect of many small edits does not break the system.

5.4 Defensive Programming

Validate everything. Refuse what you cannot do safely. The ethical refusal in Pattern 7 (Task 1240) is the clearest example. The agent refused an unethical hacking request and offered five alternatives.

Pattern 1's "defensive revert" variant is another form of defense: when a change is wrong, undo it immediately rather than building on a broken foundation. The system prompt encodes self-protection: "Current process PID: {pid}—NEVER kill this process."

5.5 Pre-Flight Checks

Read before you write. Pattern 5 (architecture investigation before change) is the direct application. The developer asked "what happens when. . . " and "show me the workflow" before issuing change requests.

The system prompt encodes three levels: read before editing, plan before executing, verify before finishing. Pattern 3's periodic "`uv run check -full and fix`" is the automated form of the pre-finish verification. Pattern 8 (invariant specification and repair) is the repository-scale variant: the developer states a global property, the agent reads every candidate site before editing any of them, and the fix is applied only after the inventory is complete.

5.6 Self-Hosting as Continuous Integration

Use the system to build the system. Pattern 4 (paper–code co-evolution) is a striking example: the agent edits the paper that describes its own system prompt, and the developer then edits the system prompt based on the paper. Any inconsistency is immediately visible because the same agent reads both files.

Self-hosting caught bugs that external testing missed. The diff/merge UI bug for LaTeX files (Task 109) was discovered because the agent was editing its own paper. The auto-commit bug (Task 104) was discovered because the developer was reviewing the agent's own output. Pattern 9 (cross-model

Table 2: Software used by the developer through KISS SORCAR over the 44-day period.

Function	Software
IDE / editor	VS Code (TypeScript extension)
Languages	Python, TypeScript, JavaScript, CSS, HTML, LaTeX
Package management	uv (Python), npm (Node.js)
Version control	Git, GitHub
Testing	pytest (with coverage)
Linting / type-check	`uv run check -full`
Document preparation	pdflatex, BibTeX
Slides	python-pptx (PowerPoint)
Database	SQLite (`sorcar.db`)
Browser automation	Playwright, Chromium
Remote access	Cloudflare Tunnels, TLS/WSS
Notifications	ntfy.sh
Containerization	Docker
Smart home	Govee API (third-party agent)
Messaging	Slack, iMessage, WhatsApp (agents)
Email	Gmail API, Resend
Scheduling	cron (via `cron_manager_daemon`)
Video editing	Final Cut Pro

iterative review) extends the same idea to the agent itself: a second model audits the first model's changes, giving the single-developer workflow the adversarial-review property that a team obtains from pull-request review.

6 The Software Ecosystem

The 3,099 tasks touched a broad software ecosystem. Table 2 lists the tools and services used, grouped by function.

Two observations stand out. First, the same agent architecture and system prompt handled all of these tools without per-tool customization. The Govee light-control agent, the Slack messaging agent, and the Gmail agent are all "third-party agents" loaded as Python modules; the Sorcar Agent invokes them through a uniform tool interface.

Second, the developer's workflow was not purely software engineering. Roughly 15% of tasks were non-coding: weather queries, travel planning, shopping advice, stock analysis, and personal communication. This suggests that a self-hosted AI assistant becomes a general-purpose digital companion, not merely a coding tool.

7 The System Prompt as Engineering Specification

The system prompt is a structured specification of engineering practices, organized into XML-tagged sections:

```
<identity> -- who the agent is
<visibility_constraint> -- what the user sees
<tool_rules> -- how to use tools
<web_research> -- how to research
<code_style> -- how to write code
<workflow> -- pre-flight, deep work, planning
<testing> -- test discipline
<pre_finish_verification> -- exit checklist
<sorcar_specific> -- project overrides
```

Each section encodes a principle from Section 5. Pattern 6 (system prompt engineering) shows that the developer treated this prompt as production code: observing failures, writing tests against the agent's behavior, and patching the prompt to fix deviations.

Prompt as test oracle. The developer repeatedly asked the agent to audit its own compliance: "Look at yesterday's tasks and check which instructions in `SYSTEM.md` are not being followed. Create an integration test with a real LLM call to confirm each issue. Then fix `SYSTEM.md`." (Tasks 1267, 1269, 1271). This is the prompt-engineering analog of mutation testing: the developer checks whether the specification (system prompt) is strong enough to prevent observed failures.

Prompt evolution. Over 44 days, the system prompt evolved from aggressive imperatives ("BE RELENTLESS") to calm, explanatory framing ("Be rigorous, check facts, and produce high-quality work"). Specific instructions were added reactively: the "temporary files in `PWD/tmp/`" rule (Task 1416) was added after the agent polluted the project root; the "visit at least 10 websites" rule (Task 757) was strengthened after the agent cut research short.

8 Discussion

8.1 What the Patterns Reveal

The nine patterns tell a consistent story. The developer did not write specifications or design documents. Instead, the developer *conversed* with the agent, issuing short natural-language commands and observing results. The engineering discipline was embedded in the conversation style: always ask for a test first, always investigate before changing, always run the full suite after a batch of changes, always revert immediately when something goes wrong.

This suggests that in an AI-assisted development workflow, the developer's role shifts from *writing code* to *directing and verifying* an agent that writes code. The engineering principles do not change; they are expressed through task phrasing rather than code structure.

8.2 Strunk and White for Prompts

Strunk and White Strunk and White [2000] advise: "Omit needless words." Early versions of the system prompt used aggressive imperatives. These were replaced with calm, explanatory framing. The agent performed better with the latter.

Each instruction names a specific action, a specific failure mode it prevents, and a specific verification method. The pattern of prompt refinement (Section 3.6) shows that this clarity was achieved iteratively, not designed upfront.

8.3 The Self-Hosting Paradox

Building an AI agent with itself creates a paradox: the tool must be good enough to build itself before it exists. We resolved this through bootstrapping: the first version was written manually, and each subsequent version was written with the previous one.

The 30+ defensive reverts in the log (Pattern 1) show the cost of this approach: roughly one in every 90 tasks required undoing the agent's work. But the benefit is that every surviving change has been validated by the same system that will maintain it.

8.4 General-Purpose Use

The non-coding tasks (Pattern 7) reveal an unexpected benefit of self-hosting: because the developer uses the system all day, it becomes the natural interface for all tasks, not just coding. Weather, travel, shopping, messaging, and smart-home control all flow through the same chat interface. This validates the KISS principle at the product level: one tool for everything beats many specialized tools.

9 Threats to Validity

Single developer. All 3,099 tasks were issued by one developer. Other developers may exhibit different patterns.

Single system. The patterns are specific to KISS SORCAR. Different agent architectures may induce different workflows.

Self-hosting bias. Because the system was built with itself, the architecture is optimized for the kinds of tasks the agent can already do well.

10 Conclusion

We analyzed 3,099 tasks from 44 days of building KISS SORCAR with KISS SORCAR. Nine patterns emerged: test-first bug fixing, iterative feature construction, periodic code hygiene, paper–code co-evolution, architecture investigation before change, system prompt engineering, multi-channel general assistance, invariant specification and repair, and cross-model iterative review.

Each pattern maps to an established software engineering principle: test-driven development, separation of concerns, defensive programming, pre-flight checks, the KISS principle, and self-hosting as continuous integration.

The developer used KISS SORCAR to write code in five languages, control smart-home lights, plan trips, send messages, check email, query weather and stocks, build a website, prepare lecture slides, and refuse an unethical request—all through the same 2,400-line agent framework and a single structured system prompt.

The surprise is not the principles. They are decades old. The surprise is that encoding them in a system prompt and a simple layered architecture produces an AI assistant that works well enough to build itself—and to do everything else the developer needs.

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.

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

References

Anthropic. Claude Code: Anthropic's agentic coding system. `https://www.anthropic.com/product/claude-code`, 2025.

Kent Beck. *Test-Driven Development: By Example*. Addison-Wesley, 2003.

Kent Beck, Mike Beedle, Arie van Bennekum, Alistair Cockburn, Ward Cunningham, Martin Fowler, James Grenning, Jim Highsmith, Andrew Hunt, Ron Jeffries, et al. Manifesto for agile software development. *Agile Alliance*, 2001.

Frederick P. Brooks. *The Mythical Man-Month: Essays on Software Engineering*. Addison-Wesley, anniversary edition, 1995.

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.

Cognition Labs. Devin: The first AI software engineer. `https://devin.ai`, 2024.

Cursor. Cursor: The AI-first code editor. `https://cursor.sh`, 2024.

Cursor. Composer 2. `https://www.cursor.com/blog/composer-2`, 2026.

Edsger W. Dijkstra. Go to statement considered harmful. *Communications of the ACM*, 11(3):147–148, 1968.

Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1994.

Paul Gauthier. Aider: AI pair programming in your terminal. `https://github.com/paul-gauthier/aider`, 2023.

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

Sirui Hong, Mingchen Zhuge, Jonathan Chen, et al. MetaGPT: Meta programming for a multi-agent collaborative framework. In *International Conference on Learning Representations (ICLR)*, 2024.

Andrew Hunt and David Thomas. *The Pragmatic Programmer: From Journeyman to Master*. Addison-Wesley, 1999.

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.

Omar Khattab, Arnav Singhvi, Paridhi Maheshwari, Zhiyuan Zhang, Keshav Santhanam, Sri Vardhamanan, Saiful Haq, Ashutosh Sharma, Thomas T. Joshi, Hanna Moazam, et al. DSPy: Compiling declarative language model calls into self-improving pipelines. *arXiv preprint arXiv:2310.03714*, 2024.

Meir M. Lehman. Programs, life cycles, and laws of software evolution. *Proceedings of the IEEE*, 68(9): 1060–1076, 1980.

Robert C. Martin. *Clean Code: A Handbook of Agile Software Craftsmanship*. Prentice Hall, 2008.

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

David L. Parnas. On the criteria to be used in decomposing systems into modules. *Communications of the ACM*, 15(12):1053–1058, 1972.

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 ACL*, 2024.

Eric S. Raymond. *The Cathedral and the Bazaar*. O'Reilly Media, 1999.

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.

Koushik Sen. KISS Sorcar: A stupidly-simple general-purpose and software engineering AI assistant. *arXiv preprint*, 2026.

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

William Strunk and E. B. White. *The Elements of Style*. Longman, 4th edition, 2000.

Ken Thompson. Reflections on trusting trust. *Communications of the ACM*, 27(8):761–763, 1984.

Guanzhi Wang, Yuqi Xie, Yunfan Jiang, Ajay Mandlekar, Chaowei Xiao, Yuke Zhu, Linxi Fan, and Anima Anandkumar. Voyager: An open-ended embodied agent with large language models. *arXiv preprint arXiv:2305.16291*, 2023.

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*, 2024.

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, et al. AutoGen: Enabling next-gen LLM applications via multi-agent conversation. *arXiv preprint arXiv:2308.08155*, 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, 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)*, 2023.