KS GOV: LONG-RUNNING AI DISCOVERY AND ADVERSARIAL TESTING FROM A SYSTEM PROMPT

Anonymous authors
Paper under double-blind review

ABSTRACT

Users ask coding agents to make a system faster or more robust and let them run for hours. Such a run outlives the context window, and one untested change can undo a day's work. KS Gov is a coding agent built for such runs: five Python modules and 4,449 significant lines under a 3,851-word system prompt. The prompt states engineering rules and two procedures: an AI discovery and optimization loop that logs tried ideas to files, and adversarial testing by alternating break and fix sub-tasks. One task-prompt sentence adds a read-only reviewer from a second vendor. In two long runs, the agent built a key-value store that sustains 5.50 Mops/s on YCSB-A over 17.2 agent hours, and made a TPC-H engine 34.5× faster than its single-threaded artifact. The first ran under an earlier one-paragraph discovery instruction, with the adversarial steps written in its task prompt; only the second ran under the seven-step loop. In both, the reviewer found defects the benchmark hid. Each case study is one run without a baseline agent. Terminal-Bench 2.0 tests the loop and tools without the two procedures: under a shorter coding prompt on the HarnessTax study's 30 tasks and seven models, the agent solves 75.6% of attempts (74.1% counting solves past the study's 100-turn cap as failures) against 70.0% for Pi, the best published harness, a success gap within sampling error. Against Pi, rerun by us on one model, the paired success gap is +11.1 points, at a cost we cannot distinguish.

1 INTRODUCTION

Large language models write code (Chen et al., 2021) and call tools (Yao et al., 2023), and a use of the resulting agents that did not exist two years ago is now common: a user asks the agent to make a system faster, or a model better, or a program correct on inputs it has not seen, and lets it run for hours through a loop of hypothesis, change, measurement, and keep-or-discard (Karpathy, 2026; Huntley, 2025). FunSearch (Romera-Paredes et al., 2024), AlphaEvolve (Novikov et al., 2025), and the AI Scientist (Lu et al., 2024) run such loops inside purpose-built systems with the loop in code. A general coding agent given the same job has a context window, a shell, a repository, and whatever rules its prompt states.

A run that lasts a day fails in ways a ten-minute task does not. The context window fills, and what the agent tried is lost unless it writes it down. An edit to a file the agent has not read, or a fix it never ran, breaks the artifact hours after the goal was last checked. A measured gain may be an overfit to the one workload the agent tuned on, and the model that made a mistake is the one asked to find it. Claude Code (Anthropic, 2025), Cursor (Anysphere, Inc., 2026), Codex (OpenAI, 2025), Devin (Wu, 2024), SWE-agent (Yang et al., 2024), and OpenHands (Wang et al., 2025) each address some of these failures, and they are compared on benchmarks such as SWE-bench (Jimenez et al., 2024), whose tasks take minutes and whose leaderboard can no longer be ordered statistically (Liu et al., 2026); Ma et al. (2026) find agents scoring near-perfectly against a hidden oracle while the library they were asked to build is unusable. A long run needs a harness that keeps the agent coding in a verified way (every file read before it is edited, every change exercised by a real test, every bug reproduced before it is fixed) and that makes the agent carry out an algorithm stated in prose, step by step, across restarts. This paper asks whether an ordinary coding agent can be brought to do this with rules in a prompt and no loop written in code.

Reviewers: please read the Reviewer Guidelines (iclr.cc/Conferences/2027/ReviewerGuidelines) and the AI Policy for Reviewers (iclr.cc/Conferences/2027/AIPolicyForReviewers). If you used AI to expand, edit, or polish your review, please provide the input text to the LLM. Better still, consider skipping the LLM and submitting your original text: we, and the authors, are much more interested in your unedited thoughts than in what an LLM has to say. AI-assisted or not, you are putting your name and reputation behind your review: LLM-generated falsehoods, hallucinations or misrepresentations are subject to disciplinary action, which may include desk-rejecting all papers you have authored.

KS Gov is five Python modules, 4,449 significant lines, arranged as layers: a ReAct loop, continuation past the context window from a summary the agent writes, tools with a parallel dispatcher and a live model switch, a persistent chat that records every task, and a git worktree per task. A 3,851-word system prompt states the engineering rules and, in the same plain text, two procedures a user can name in a task: an AI discovery and optimization loop whose record of tried and failed ideas lives in two files that outlast the context, and adversarial testing by alternating break and fix sub-tasks. A sentence in the task prompt adds the last piece: one model writes, a second vendor's model reviews read-only. Two case studies test whether prompt, layers, and review together carry a long run: HydraKV, a larger-than-memory key-value store built under an adversarial discovery prompt over 17.2 agent hours and 5 context-window continuations, which sustains 5.50 Mops/s on YCSB-A, 5.9× the FASTER figure its task states, and a TPC-H engine made 34.5× faster under an optimization prompt. HydraKV's starting engine came from Claude Code (Anthropic, 2025) under a plain optimization prompt with no discovery procedure; it reached 1.85 Mops/s by indexing keys in a flat array that holds only because the trace's keys are below 2^{32}. Two runs with no controlled baseline agent say little about other harnesses, so we also ran the same agent loop and tools under a shorter coding prompt on the HarnessTax study's 30 Terminal-Bench 2.0 tasks and seven models, next to its published numbers for Claude Code, Codex CLI, and Pi, and then on the 59 tasks it did not sample. That prompt leaves out the two procedures, so the benchmark tests the loop, tools, and coding rules only.

Our contributions are: (1) a five-layer agent with the discovery and adversarial-testing procedures stated in its system prompt in plain text (Section 2); (2) an open, audited record of two long-running projects carried out from a prompt, HydraKV under the earlier one-paragraph discovery instruction and the TPC-H engine under the seven-step loop, released with their tests, logs, audits, and process records (Section 3); the record documents what each run did and does not measure an effect of the procedures; and (3) a run of the same agent loop and tools on the HarnessTax study's 30 Terminal-Bench 2.0 tasks and seven models, next to the study's published numbers, a run on the rest of Terminal-Bench 2.0 (the 59 tasks the study did not sample), and a same-model run of Pi on the same tasks for a paired comparison (Section 4). Apart from the Claude Code engine HydraKV started from, no other agent worked on our own tasks, and each case study is a single run with no rerun and no ablation of the procedure (Section 6).

2 The System

2.1 The five layers

KS Gov is five agent classes arranged as layers (Figure 3), one per module with its helpers; each layer owns one concern and hands the rest upward, and Appendix A describes each in turn. A line is significant if it is neither blank, a comment, nor part of a docstring (`significant_lines` in the released script): the five modules are 4,449 lines, the code a task executes (with the `core` and agent packages) 25,439, and the whole non-test package 71,152. The KS Agent, innermost, runs a ReAct loop (Yao et al., 2023) over the model's native function-calling API. The Relentless Agent runs a task that outgrows the context window as a sequence of sub-sessions: near the end of its context, the executor calls `finish` with `is_continue=True` and a chronological summary of what it did, and a fresh sub-session starts from the prior summaries. The Chat Agent records every task, its settings, and its result; the process rows of Appendix C are mined from that record. The Worktree Agent runs every task that may edit files on its own branch in a worktree and squash-merges it at the end, so a discarded idea is a revert. The Tool Agent adds `Bash`, `Read`, `Edit`, and `Write` (the last two refuse an existing file the model has not been shown), a browser, `run_parallel` (one sub-agent with its own context per task string), `set_model` (hand the live conversation to another model), `run_agent`, `decide`, and a memory.

2.2 The system prompt

`SYSTEM.md` is 3,851 words. Its rules are the kind a senior engineer gives a new hire, and together they are what we mean by verified coding: read every file before editing or overwriting it; write end-to-end tests only, with no mocks, patches, fakes, or test doubles (mocks couple tests to production code (Spadini et al., 2017)); reproduce a bug with such a test before fixing it, and fix the root cause; and, before calling `finish`, check each requirement against what was delivered. One rule is backed by code, the read-before-edit check in `Edit` and `Write`; the rest are sentences.

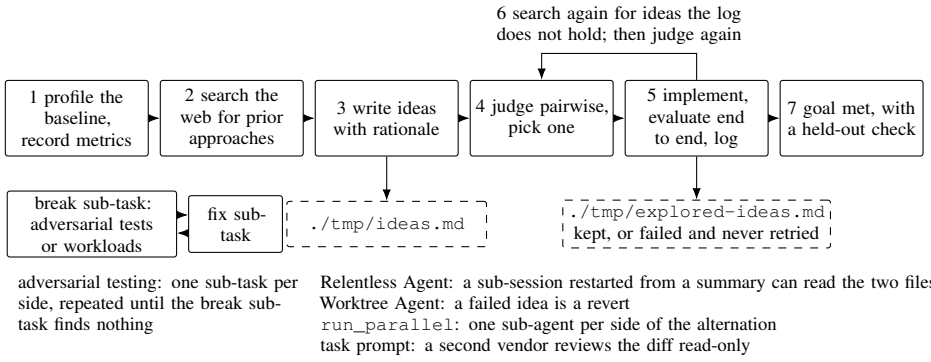


Figure 1: The two procedures the current system prompt states in prose (Appendix B); of the case studies, only the TPC-H task ran the seven steps, in an earlier wording. Top: the seven steps of the discovery and optimization loop with the two files that hold its state. Bottom: the break-and-fix alternation of adversarial testing, and the layer or tool that carries each step across a run of hours.

2.3 Discovery and adversarial testing by instruction

Two paragraphs of the prompt say how to run the long tasks this paper is about. Both are quoted in Appendix B; Figure 1 draws them. The first, headed "AI discovery, auto research, and optimization," is seven numbered steps: profile the baseline; search the web for prior approaches; write candidate ideas with their rationale to `./tmp/ideas.md`; judge them pairwise to pick one; implement it, evaluate end to end, and log the idea and its metrics to `./tmp/explored-ideas.md`, keeping it if better and marking it failed if worse so that it is never retried; search again for ideas the log does not hold; and stop when the user's goal is met, with a held-out check against overfitting. The second, "Adversarial testing," is two sentences: use a sub-task to break the system with adversarial tests or workloads, and another sub-task to fix what breaks.

None of this adds code. A user starts a run by naming the technique in the task text, and the model carries out the steps with the tools of Section 2.1: `Bash` for the profile and the evaluation, the browser for the search, `run_parallel` for the break and fix sub-tasks, `Write` for the two logs. The search state lives in two files, so a sub-session where the Relentless Agent starts from a summary can read what was tried and what failed; the rule that a failed idea is never retried is meant to keep a day-long run from circling. The testing rules of Section 2.2 apply to every implementation step, and the held-out check guards against tuning to the one workload the loop optimized on.

The seven-step text is the current one; the TPC-H task of Section 3.2 ran under an earlier wording of the same steps (Appendix B). The HydraKV tasks ran under an earlier paragraph (Appendix B) that asks the agent to search, implement, measure, and note tried ideas and metrics in a file; it names no files and has no numbered steps, pairwise judging, or adversarial-testing paragraph. HydraKV's adversarial steps came from its task prompt, and its state lived in `PROGRESS.md` and, from the first hardening task on, `DISCOVERY_LOG.md`.

2.4 One task, two vendors

Because `set_model` and `run_parallel` are ordinary tools, a user can ask for a division of labor in the task text. The instruction we settled on, quoted in Appendix B, says: use `claude-fable-5` for all development; use `gpt-5.6-sol` through `run_parallel` "for a thorough read-only review and debugging of the other model's work"; spend at most half the task budget on it; and "ask the model not to invent new problems." The reviewer runs as a separate sub-agent with its own context; the developer judges and fixes what it reports. The budget and read-only clauses are requests in a prompt. We wanted a second vendor because a model favors its own output (Panickssery et al., 2024) and does not correct its own reasoning without external feedback (Huang et al., 2024).

3 TWO LONG-RUNNING RUNS FROM A PROMPT

The procedures of Section 2.3 were written for tasks that take days, and the two projects below are the runs of that kind for which we have a complete record: a task prompt with a numeric goal, `claude-fable-5` developing, `gpt-5.6-sol` reviewing read-only, and the tests, logs, and audit reports released. Table 4 (Appendix C) gives their process record mined from the task database. HydraKV's first task (3.8 hours, $81) ran the adversarial discovery its task prompt describes under the earlier one-paragraph instruction, and five tasks of hardening and audit response followed, 17.2 agent hours in all; the TPC-H engine took one 1.3-hour optimization task under the seven-step loop without its web search, a separate read-only review task, and an audit task that wrote adversarial parameter tests. We read each record for three things: whether the shipped code was tested as the rules require, whether the agent followed the procedure its prompts stated, and what the second-vendor reviewer found that the benchmark did not.

3.1 HYDRAKV: A KEY-VALUE STORE BUILT UNDER AN ADVERSARIAL LOOP

Task. The benchmark description (`benchmark/TASK.md`) fixes everything except the engine: a YCSB-A (Cooper et al., 2010) trace of 250 million 64-bit keys with 100-byte values (50% reads, 50% blind upserts, Zipfian θ=0.95), 16 worker threads, a 30-second scored window from a cold cache, the median of three runs, an 8 GiB resident-memory cap the harness enforces, a validation pass that must find every loaded key, and a read-after-write integrity auditor. The figure to beat was the 0.93 Mops/s reported for Microsoft's FASTER (Chandramouli et al., 2018); the machine was a GCP `n2-standard-64` with eight local NVMe SSDs in RAID0.

The adversarial task. The starting point was an optimized engine that Claude Code (Anthropic, 2025) wrote from a plain optimization prompt, without any AI-driven discovery. It reward-hacked the benchmark: it indexed keys in a flat four-byte-per-key array, which works only because the published trace uses keys below 2^{32}; it reached 1.85 Mops/s. The task prompt asked for adversarial AI discovery: generate workload variants the engine was not tuned for, break it, fix it, and finish on a fresh held-out variant with no tuning, with read-only review on `gpt-5.6-sol` capped at 20% of the budget. The task ran 3.8 hours and $81 across one continuation and received four steering messages: a warning against fitting the trace, a mid-task change of goal to 10 Mops/s, and twice "do not hard-code 250M dense keys". The agent's first move was a trace generator (`gen_variant.cc`) with sparse SplitMix64 keys and scrambled, clustered, drifting, and mixed hot sets. The first variant, sparse keys under the original skew, killed the inherited engine by out-of-memory before it could score; the replacement, a fingerprint hash index sized from the memory budget, scored 1.87 Mops/s on the original trace and 1.91 on the variant. The 10 Mops/s goal then drove a redesign led by the agent's own diagnostics (a 72% cache hit rate; one contended I/O queue): log-append flushes, a doorkeeper admission filter, and 64 sharded I/O queues. Two intermediate builds regressed to 1.10 and 0.47 Mops/s; after profiling them and measuring the device with `fio`, the agent built eight raw `io_uring` rings driving an asynchronous read-miss state machine (5.27 Mops/s), then SLRU-style (Karedla et al., 1994) cache segmentation (5.50). Four further cache designs measured 0.78, 0.95, 1.98, and 4.17 Mops/s and were reverted, with their numbers recorded in the task's `PROGRESS.md`. Two read-only reviews (14 bugs and 10 risks, then 21 line-referenced findings, per the task record) were answered inside the task; the second found that a fingerprint alias, likely among 250 million keys, could abort the process or bury a live key behind a newer slot. The task's final report gives 5.48 Mops/s on the original trace, 5.67 and 5.56 on the sparse and clustered variants, 3.97 on the drifting variant, and 3.87 on the held-out mix, all with validation passing; these five numbers rest on the task record, since the harness logs stayed on the benchmark server. The 10 Mops/s goal was not met. The agent's closing argument was that 50% reads at 10 Mops/s need a hit rate of at least 87% under the measured device ceiling, that every admission policy it tried saturated near 77%, and that getting past it would take memory outside the cap.

Hardening and audits. Five more tasks followed, each opened by the user with an external audit or a request. The first (5.8 hours, the longest task of the project) answered a production-readiness audit. Its discovery log lists 5 kept idea groups (slot layout, crash recovery, fail-soft I/O, bounded oversized values, log compaction by hole punching) and five rejected with reasons; the tests were written before the fixes; and a second adversarial round found a deleted-key resurrection bug after the

full test matrix had passed. A second external audit (`AUDIT2.md`) reproduced 4 bugs on hardware that the scored benchmark cannot reach: a deadlock between concurrent upserts and deletes, an out-of-memory on a full disk, a recovery that dropped 480K of 976K keys and reported success, and deletes 160 times slower than upserts. A same-day A/B on the reference node gave 5.50 Mops/s for the fixed engine and 5.50 for the pre-fix engine. A workload task then ran the engine over 7 mixes (Table 3, Appendix C), two of them named in steering messages and one steer pointing at FASTER's own test suite; the task fixed 8 bugs. 4 came from the workloads and the test matrix, among them clean-shutdown data loss present on every workload, oversized values with no disk path at all, and a stale tombstone that could delete a key it did not belong to; the other 4 were found by the reviewer, and no workload had triggered them. The last audit found that the fix for slow deletes had introduced crash-resurrection of deleted keys: a delete returned once the key was queued for a background reaper, so a kill before the reaper ran let recovery bring the key back. The repair added a delete-intent log ahead of the acknowledgment; two review rounds on it returned 14 findings and then a linearization race in the round-one fixes, all addressed.

Result, and what it does not show. The released engine is 4,324 non-blank lines of C++17 with 28 end-to-end tests (2,112 lines): an append-only slot log, a fingerprint hash index, and a write-back cache with TinyLFU-style admission (Einziger et al., 2017) and SLRU segmentation. On the reference node, at commit `c7745c61` before the workload fixes above, it sustained 5.50 Mops/s (5.9× the stated FASTER figure) with zero auditor failures, and its Linux test matrix passed under ASan, UBSan, and TSan with 92.5% line coverage; after them (commit `5050e98f`) it measured 5.51 Mops/s and 89.3% coverage in runs whose logs stayed on the node. FASTER was not run on this machine, and no other store was; 5.9× compares the scored YCSB-A mix against a figure in the task text. On other mixes of Table 3, run for correctness without tuning, the engine is far slower: 0.30 Mops/s on `RMW_100`, all read-modify-writes, the update pattern FASTER's in-place updates target, and 0.08 on `TIMESERIES_HD`, which inserts at the tail of a key range and deletes at its head. The drifting variant and the held-out mix ran at 3.97 and 3.87 Mops/s, a third below the scored trace, so the cache policy is sensitive to how the hot set moves. "Production-hardened" is the agent's phrase, and the record argues against it: three external audits each found bugs that the previous round had introduced or missed. The record does support the shape of the loop: variants generated by the agent broke the inherited engine before any human did, four designs were measured and discarded, and each of the 9 review sub-agents returned findings the developer then fixed in the same task.

3.2 BESPOKE OLAP: A TPC-H ENGINE MADE 34.5× FASTER

Task. *Bespoke OLAP* (Wehrstein et al., 2026) synthesizes a one-size-fits-one C++ engine for the 22 TPC-H query templates: the templates are fixed, only their parameters vary, and the engine is allowed to exploit that at build time. Its released artifact runs single-threaded and, on our 32-core Linux machine, needs 3,132.2 ms for the 22 queries at scale factor 10 (sum of per-query medians over five repetitions, seed 42). The task prompt asked for at most 783 ms, a 4× speedup, with validation against DuckDB reference results clean on six (scale factor, seed) configurations, under constraints written to close the obvious shortcuts: modify only `engine/`; keep the stdin protocol, the CSV output, and the timing window; no result memoization across requests; precomputation at ingest may depend on the schema, the data, and the fixed templates but never on a placeholder value; every query must stay correct for arbitrary valid parameters; and aggregates keep the baseline's integer arithmetic. The prompt was directive where the earlier one was open: it named the `AffinityGuard` that pinned the process to one CPU, prescribed OpenMP as the first round and precomputed join artifacts as the second, and listed the heaviest queries. The development task ran 1.3 hours and $43 in one context window, with no continuation and no steering; the loop's web-research step was not asked for and did not run.

What the agent did. The first round removed the pin, added `-fopenmp -march=native`, and parallelized the scan and join kernels with thread-local partial results merged at the end. The second round moved parameter-independent work to ingest, where it is not timed: date-indexed prefix cubes for the queries whose only free parameters are a date window and a fixed dimension (Q1, Q4 to Q8, Q12, Q14, Q15), which reduce to a difference of two prefix entries; partkey join indexes in CSR form for Q9, Q17, and Q20; per-order aggregates for Q10 and Q18; rows pre-filtered by the templates' fixed predicates for Q13, Q16, Q19, Q21, and Q22; per-order segment codes for Q3 and

Table 1: TPC-H at scale factor 10, per-query median execution time in ms over five repetitions on the 32-core machine: the released single-threaded artifact (seed 42), the agent's final engine on seed 42, and the same engine on seed 999, which was not among the three seeds the task named. Speedup is baseline over final on seed 42; the text lists the mechanism behind each query.

Query	Baseline	Final	Seed 999	Speedup
Q1	161.1	0.007	0.007	23,015
Q2	4.2	4.466	4.380	0.9
Q3	137.5	8.624	8.089	15.9
Q4	13.0	0.020	0.019	650
Q5	33.7	0.009	0.009	3,741
Q6	33.0	0.010	0.009	3,303
Q7	74.4	0.006	0.007	12,408
Q8	82.4	0.008	0.010	10,299
Q9	622.2	3.719	2.606	167
Q10	104.5	39.054	33.913	2.7
Q11	16.2	0.428	0.331	37.9
Q12	377.2	0.008	0.008	47,156
Q13	219.5	9.560	9.129	23.0
Q14	43.2	0.007	0.008	6,167
Q15	103.0	0.810	0.782	127
Q16	34.5	10.050	11.712	3.4
Q17	64.1	1.365	1.421	47.0
Q18	372.3	2.376	2.190	157
Q19	216.8	0.050	0.045	4,336
Q20	41.7	5.296	5.431	7.9
Q21	367.9	4.944	5.025	74.4
Q22	9.7	0.014	0.012	691
Total	3,132.2	90.8	85.1	34.5

per-nation value lists for Q11; and Q2 left on its original path. Every fast path checks its assumptions and falls back to the baseline algorithm when they do not hold. Profiles drove the last changes: a serial 60-million-row scan hidden in Q19 and serial radix sorts in Q10. The task ended at 86.8 ms and 36.1×; after the review fixes below, the figure is 90.8 ms and 34.5×. Table 1 gives the per-query picture. 10 of the 22 queries now take under 0.1 ms, which is what a cube lookup costs; the four heaviest baseline queries (Q9, Q12, Q18, Q21, 56% of the baseline total) take 11.0 ms together. Q10, the largest remaining contributor, is now 43% of the total. 1 query, Q2, is slower than baseline, by 0.3 ms on a 4 ms query. The held-out seed 999 tracks seed 42 query by query.

Review and audit. A separate task on `gpt-5.6-sol`, told to modify nothing and given 22 steps ($2.55), worked through six checks: timing-window integrity, absence of cross-request memoization, parameter independence of the precomputation, races and numeric boundaries, dispatch completeness, and output fidelity. Five passed with file-and-line evidence; the fourth failed on two defects. In Q20, a parallel loop over matching parts marked qualifying suppliers in a shared byte vector without synchronization, a data race whenever a supplier qualifies through more than one part, which the SF10 references (1,738 to 1,830 result rows) make routine; the fix is an atomic store. In Q1, the new prefix cube accumulated `sum_charge` in `int64_t` where the baseline uses `__int128`; the reviewer placed the overflow near scale factor 8,300, a scale the task never runs but the engine claims to support. Neither defect is visible to the DuckDB validation at SF10, and both were fixed before the final measurements. A third task then audited the fast paths for arbitrary parameters. It added path markers to every guard and fallback, compiled in only under an audit flag, and wrote 265 request cases (boundary dates, out-of-dictionary strings, extreme numbers, empty results, and for every reachable guard a case in which the fallback engages), each compared against the pristine artifact as oracle. All 265 pass; the one finding is a contract note that Q22 truncates phone codes longer than two characters, as the artifact does. The audit's own read-only reviewer found no engine defect but three holes in the test harness, which the developer fixed before the 265-case figure was taken. The audit task took 0.97 hours, $45, and 291 steps.

What the number means. Removing the pin let the engine use 32 cores, but we did not measure the two rounds separately, so the split between parallelism and precomputation is unknown. The project's README quotes 284 ms for the authors' own multithreaded follow-up on the same machine, about 3.1× behind the engine, but no run log of it was kept, so the 34.5× we report is against the single-threaded artifact only. The prefix cubes are what the paper's contract invites, since the templates are fixed by construction, but they move work to ingest (52.5 to 63.0 s, 20%) so the extra ingest is repaid after 4 runs of the query suite, and answer only these 22 templates; nothing here is a general OLAP result. The task prompt supplied a researched plan, and the agent executed it, which is the loop's implement-and-measure step without its search steps; the agent's own contributions are the choice of artifacts per query, the fallbacks, the profile-driven fixes, and the 265-case audit.

	KS Gov		Pi		Codex CLI		Claude Code	
Model	solved	$/att.	solved	$/att.	solved	$/att.	solved	$/att.
Claude Fable 5	**80.0**	1.50	71.1	1.08	72.2	0.98	75.6	1.55
Claude Opus 4.8	**74.4**	1.21	72.2	0.76	72.2	0.85	68.9	0.90
Claude Sonnet 4.6	**76.7**	2.23	65.6	0.61	63.3	0.55	62.2	0.67
Claude Haiku 4.5	**51.1**	0.39	47.8	0.25	31.1	0.21	41.1	0.26
GPT-5.6 Sol	**85.6**	0.69	83.3	0.42	78.9	0.76	71.1	1.35
GPT-5.6 Luna	**78.9**	0.05	76.7	0.04	72.2	0.06	70.0	0.10
Kimi K3	**82.2**	1.14	73.3	0.38	70.0	0.45	66.7	0.52
Mean of 7 models	**75.6**	1.03	70.0	0.51	65.7	0.55	65.1	0.77

Table 2: Terminal-Bench 2.0, the study's 30-task sample, three attempts per task: percentage of attempts solved (mean of task means) and cost per attempt in USD, KS Gov (630 counted attempts, no turn cap, $50 budget) next to the three harnesses of Pan et al. (2026) as published (100-turn cap, high effort). Bold marks the best point estimate per row; intervals are in the text.

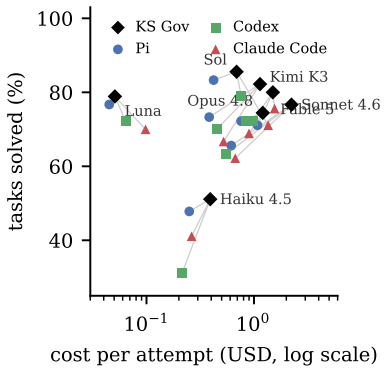


Figure 2: Table 2 as a cost-success plot. One diamond per model for KS Gov; the gray lines join it to the same model under Pi, Codex CLI, and Claude Code.

4 Terminal-Bench 2.0 against three harnesses

The two runs have no controlled baseline agent on the same task. The HarnessTax study of Pan et al. (2026) provides one comparison with other harnesses on the axis this paper covers. It holds the model fixed and varies the harness: seven models (the rows of Table 2) each run under Claude Code (Anthropic, 2025), Codex CLI (OpenAI, 2025), and Pi (Earendil Works, 2026) on the same 30 tasks sampled from Terminal-Bench 2.0 (Merrill et al., 2026) and 30 from SWE-bench Lite, three attempts per task, scored by each benchmark's official verifier. Its conclusion is that the harness moves cost more than success, and that Pi, which gives the model four tools and little else, sits on the cost-success frontier. The study publishes its per-model numbers, task list, and protocol, so we ran our harness on the same 30 tasks under the same scoring, choosing Terminal-Bench 2.0 over SWE-bench Lite because its tasks exercise what our rules are about: long shell sessions, background processes, and a container state the verifier inspects.

The benchmark agent is KS Gov with the general-purpose instructions removed from its system prompt and the coding and software-engineering instructions retained: 727 words in place of the 3,851-word `SYSTEM.md`. A Harbor (Harbor Framework Team, 2026) agent (Harbor 0.20.0) hands each trial's container to the framework, whose tools run inside it: the built-in toolset of Section 2.1 with the browser, memory, and worktree off. The prompt (archived verbatim with the records) names the tools and states six rules of the kind Section 2.2 states, among them: list what "done" means for each item of the task, never modify the task's input files, and before finishing, check every requirement from a fresh shell. Two harness mechanisms back the rules in code. After each shell command, the harness appends a note to the result when processes that were started by the command are still running or when a pre-existing file in the working directory has changed, and a command that would kill every process in the container or delete the workspace is refused with a message. Unlike the three harnesses in the study, we set no turn cap and no wall-clock limit: an attempt ends when the agent calls `finish` or has spent $50. We ran 630 attempts (7 models, 30 tasks, three each) on September 25, 2026, and all of them count. Statistics follow the study: a task's attempts are averaged, then the 30 tasks; pooled figures average the seven per-model results, and intervals are percentile bootstraps over 10,000 resamples of the tasks with the models weighted equally.

Table 2 and Figure 2 give the result. Pooled over the seven models, KS Gov solves 75.6% of attempts ([63.8, 85.9] over tasks) against 70.0% for Pi, 65.7% for Codex CLI, and 65.1% for Claude Code, and its point estimate is the best of the four harnesses on 7 of the 7 models. With 30 tasks a per-model interval spans 27 points on average, so no single row settles its gap; the pooled gap over Pi is 5.6 points and over Claude Code 10.5, and the pooled interval covers all three published means. Counting solved attempts past the study's 100-turn cap as failures, the pooled rate is 74.1%. These statements are about success. The two cost columns rest on different price lists (below), so we draw no cost ratio

from them; on one price list and protocol, the paired Pi run below finds a cost difference of +0.05 dollars per attempt ([-0.71, +0.70]).

The trajectories show where the success and the cost come from. The mean attempt took 27.9 turns (longest 276) and 17.2 minutes; 28 attempts went past the study's 100-turn cap and 9 of those were solved, and 3 cost more than $15 (the most $33.64), 2 of them were solved. 250 of the 630 attempts received a harness note about a still-running process or a changed input file (901 in all), and the harness refused 15 tool calls in 15 attempts: 14 were `run_agent` calls, which have no channel agent to reach in the container, and 1 a shell command the guard misread. The cost comes from the number of turns, the prompt and tool schemas sent with every call, and the notes.

The study ran each harness at high effort; we sent the providers' default request parameters. Its table (September 22, 2026) is priced at its September 1 list and ours at the framework's price table, with two providers for Kimi K3. The seven rows share the 30 tasks, so they are seven views of one sample.

Pi on the same model, paired. To remove the date, price list, and cap from one row of Table 2, we ran Harbor's Pi agent (Pi 0.87.1) on Claude Fable 5 on September 26, 2026: the same 30 tasks, three attempts each, at the study's high-effort setting, under our protocol (no turn cap, no wall-clock limit; Pi has no budget cap, so an attempt ends when the model stops), its token counts priced by the framework's table. All 90 attempts count, 1 of them stopped by us after two hours, while Pi waited on a training command it had started, and counted as failed with its spend. Pi solved 62 attempts, 68.9% ([52.2, 84.4]) at $1.45 and 14.5 turns per attempt; KS Gov on the same model solved 80.0% at $1.50 and 15.3 turns. Both runs attempted every task, so the test is paired: the success gap is +11.1 points ([+3.3, +20.0], tasks resampled once for both harnesses), and the cost difference +0.05 dollars per attempt ([-0.71, +0.70]). KS Gov has the higher task mean on 7 tasks, Pi on 0, and 23 tie, 20 of them solved by both on every attempt (exact sign test, $p = 0.016$). Pi's 3 attempts on `feal-linear-cryptanalysis` all ended in an API error (two in a Pi client bug on a mid-output model fallback, one in a content filter), not a wrong answer; without that task the gap is +8.0 points (6 tasks to 0, $p = 0.031$). Codex CLI and Claude Code were not rerun.

The rest of Terminal-Bench 2.0. We also ran the same agent and prompt on Claude Fable 5 on the 59 tasks the study did not sample, three attempts each, under the same protocol. One attempt, on `extract-moves-from-video`, was stopped before it finished and left no result; as with Pi's stopped attempt, we count it as failed. The agent then solves 79.1% of attempts on the unsampled tasks and 79.4% over all 89 tasks. The intervals below use the 176 counted attempts (140 solved; 4 verifier timeouts count as failed), on which the rates are 79.7% ([70.1, 88.1]) and 79.8% ([71.9, 86.9]), against 80.0% on the study's 30 tasks, a gap of +0.3 points ([-15.8, +15.4], each set resampled on its own). The unsampled tasks cost $1.60 per attempt against $1.50; their attempts are longer (17.2 against 15.3 turns) but cheaper per turn ($0.089 against $0.098, total cost over total turns: $1.54 per attempt over 17.2 turns), and the rest of the gap is task weighting, the 2 counted attempts of `extract-moves-from-video` ($13.27 each) carrying a full task's weight.

5 Related Work

SWE-agent (Yang et al., 2024) runs one session with a purpose-built interface; mini-SWE-agent (SWE-agent team, 2026) is about 100 lines with a bash-only action space. OpenHands (Wang et al., 2025) condenses a session in place (OpenHands, 2026), where the Relentless Agent restarts a fresh session from a summary the agent writes. Claude Code (Anthropic, 2025) compacts context automatically (Anthropic, 2026a), creates a worktree per session on request (Anthropic, 2026b), and refuses to edit a file it has not read; Cursor 2.0 (Cursor Team, 2025) runs parallel agents in worktrees; the Worktree Agent makes the worktree the default.

The discovery loop has older and larger relatives. FunSearch (Romera-Paredes et al., 2024) evolves a population of programs against a fixed evaluator; AlphaEvolve (Novikov et al., 2025) extends it to whole code files with an ensemble of models and evaluators; the AI Scientist (Lu et al., 2024) runs idea, experiment, paper, and review as one pipeline. AIDE (Jiang et al., 2025) searches a tree of candidate solutions, refining the promising ones, and Reflexion (Shinn et al., 2023) keeps a written memory of failed attempts for the next trial. There, the loop is code that calls the model; in ours, it is seven sentences in a prompt, run by the model with a shell and a browser, with the state of the

search in two Markdown files. Huntley (2025)'s Ralph loop restarts a coding agent from a plan file until the plan is done, and Karpathy (2026)'s `autoresearch` drives a keep-or-discard loop from a `program.md` that only the human edits; these are the closest in spirit, and ours adds the held-out check and the adversarial sub-task, which is random and property-based testing (Claessen & Hughes, 2000) with a model-written generator for each system. Anthropic's harness for long-running agents (Young, 2025) has each new context window read a feature list and a progress file first, the same state-in-files idea, and Carlini (2026) ran 16 Claude Code agents in a Ralph-style loop for about 2,000 sessions to build a C compiler that builds Linux 6.9, a long-running case study held on course by tests. StateM (Qin et al., 2026) is the closest system in aim: a runtime that keeps durable state, phase-local context, and versioned runbooks around a fixed model, and it reports same-model gains of eight to nine points on Terminal-Bench 2.1 (GPT-5.6 Luna from 76.7 to 85.4%) on all 89 tasks. Its state and procedures live in a runtime with checked transitions; ours in a prompt.

Several recent studies measure the harness or its prompt. Ben Sghaier et al. (2026) find, across 35 releases of one CLI with the model fixed, that the harness alone moves SWE-bench Verified quality; Fan et al. (2026) ablate planning, action space, and context management; Lin et al. (2026) localize an evolved harness's gain to tools and memory, with none of it in the prompt; Zhang et al. (2026) find that the beneficial rules in a rule file are negative constraints. Sambharia (2026) coined the term for the per-request payload of tool definitions and instructions. Agarwal & Jain (2026) apply natural-language runtime policies and action denials at the harness level on Terminal-Bench 2.1; our harness notes and refused commands are policies of that kind, written by hand. Park et al. (2026) optimize a harness offline from failure traces on Terminal-Bench 2.0; ours is fixed for the run. Wang et al. (2026) find that a harness searched and scored on the same Terminal-Bench 2.1 tasks generalizes poorly to held-out ones and does not consistently beat test-time scaling at a matched budget; our run on the unsampled tasks bears on the first concern only. On review, Kim (2026) report from 375 artifacts of a Claude-and-Codex protocol that heterogeneous reviewer pairs record a defect more often than homogeneous ones (69.8% against 53.1%); our two runs add observations of the same kind and no controlled comparison.

6 Limitations and Threats to Validity

The case studies are two single runs, one per procedure, with no rerun, no controlled baseline agent or person on the same task (HydraKV's starting engine came from one uncontrolled Claude Code session), and no ablation of the procedure. Only the TPC-H task ran the seven-step loop, without its search steps; HydraKV ran the earlier paragraph and never used the two log files. Neither task was run without the procedure or without review, and Terminal-Bench 2.0 does not test the procedures. HydraKV's speedup is against a figure in its task text for the scored mix only; its 10 Mops/s goal was not met, and its held-out mix ran a third slower than the scored trace. The TPC-H speedup carries the caveats of Section 3.2. In both runs, the second-vendor reviewer is also a stronger model than the developer, so nothing here separates "different vendor" from "stronger model" or from a second pass of any kind. The author of the task prompts is one of us. The Terminal-Bench 2.0 comparison with the study's published numbers differs from its protocol in date, effort, price list, cap, and prompt at once; the paired run removes all but effort and prompt, for Pi only, on one model. Memory and the channel agents are not evaluated.

7 Conclusion

Five modules and a system prompt of verified-coding rules and discovery procedures carried two long runs: a key-value store that sustains 5.50 Mops/s on the scored YCSB-A mix over 17.2 agent hours, under an earlier discovery paragraph and an adversarial task prompt, and a TPC-H engine 34.5$\times$ faster than its single-threaded artifact, under the seven-step loop; in both, a second vendor's reviewer found defects the benchmark hid. On the HarnessTax study's 30 Terminal-Bench 2.0 tasks, without the procedures, the agent solved 75.6% of attempts (74.1% with solves past 100 turns counted as failures) against 70.0% for the best published harness, a success gap within sampling error; the paired gap over Pi rerun on one model is +11.1 points ($p = 0.016$).

Reproducibility statement

The supplementary material contains an anonymized copy of the framework source, the case-study process mining script with its JSON output, the benchmark agent with the archived text of its system prompt and the per-attempt records of the Terminal-Bench 2.0 runs, the scripts that recompute every macro in this paper from those files and draw Figure 2, and both case-study projects with their tests, logs, and audit reports. The task database itself contains private prompts and is not released; the mining script runs on any `ksgov.db`. The archive is built by a script that renames the framework, rejects any surviving occurrence of the framework's or the authors' names in a path or text file, extracts the result, and reruns the recomputation script inside it. Rerunning the Terminal-Bench 2.0 runs needs Anthropic, OpenAI, and OpenRouter API keys; the 30-task run cost $649, the run on the other 59 tasks $270, and the Pi run $130.

AI use statement

The system described here wrote its own code under the prompts recorded in its task database, including the benchmark agent and the scripts that compute the numbers in this paper. We used AI to proofread the text of this paper and fix grammatical and spelling mistakes; every number in the macro block was computed by the released scripts; and a read-only review by a second model was applied to the paper as to the code. We take responsibility for all content.

References

Nikita Agarwal and Nivedit Jain. FIRE: Failure-informed runtime engineering for reliable language-model agents. arXiv preprint arXiv:2609.26048, 2026.

Anthropic. Claude 3.7 Sonnet and Claude Code. `https://www.anthropic.com/news/claude-3-7-sonnet`, February 2025. Announcement of Claude Code, 24 February 2025. Product page: `https://claude.com/product/claude-code`.

Anthropic. Claude Code documentation: Explore the context window. `https://code.claude.com/docs/en/context-window`, 2026a. Accessed 18 September 2026.

Anthropic. Claude Code documentation: Common workflows (scheduling, worktrees, subagents). `https://code.claude.com/docs/en/common-workflows`, 2026b. Accessed 18 September 2026.

Anysphere, Inc. Cursor: AI coding agent for building ambitious software. `https://cursor.com`, 2026. Accessed September 2026.

Oussama Ben Sghaier, Hao Li, Bram Adams, and Ahmed E. Hassan. Don't blame the large language model: How agent harness evolution shapes coding agent quality. arXiv preprint arXiv:2607.03691, 2026.

Nicholas Carlini. Building a C compiler with a team of parallel Claudes. Anthropic engineering blog, `https://www.anthropic.com/engineering/building-c-compiler`, February 2026.

Badrish Chandramouli, Guna Prasaad, Donald Kossmann, Justin Levandoski, James Hunter, and Mike Barnett. FASTER: A concurrent key-value store with in-place updates. In *Proceedings of the 2018 International Conference on Management of Data (SIGMOD)*, pp. 275–290. ACM, 2018. doi: 10.1145/3183713.3196898.

Mark Chen, Jerry Tworek, Heewoo Jun, Qiming Yuan, Henrique Ponde de Oliveira Pinto, Jared Kaplan, Harri Edwards, Yuri Burda, Nicholas Joseph, Greg Brockman, Alex Ray, Raul Puri, Gretchen Krueger, Michael Petrov, Heidy Khlaaf, Girish Sastry, Pamela Mishkin, Brooke Chan, Scott Gray, Nick Ryder, Mikhail Pavlov, Alethea Power, Lukasz Kaiser, Mohammad Bavarian, Clemens Winter, Philippe Tillet, Felipe Petroski Such, Dave Cummings, Matthias Plappert, Fotios Chantzis, Elizabeth Barnes, Ariel Herbert-Voss, William Hebgen Guss, Alex Nichol, Alex Paino, Nikolas Tezak, Jie Tang, Igor Babuschkin, Suchir Balaji, Shantanu Jain, William Saunders, Christopher Hesse, Andrew N. Carr, Jan Leike, Josh Achiam, Vedant Misra, Evan Morikawa,

Alec Radford, Matthew Knight, Miles Brundage, Mira Murati, Katie Mayer, Peter Welinder, Bob McGrew, Dario Amodei, Sam McCandlish, Ilya Sutskever, and Wojciech Zaremba. Evaluating large language models trained on code. arXiv preprint arXiv:2107.03374, 2021.

Koen Claessen and John Hughes. QuickCheck: A lightweight tool for random testing of Haskell programs. In *Proceedings of the Fifth ACM SIGPLAN International Conference on Functional Programming (ICFP)*, pp. 268–279, 2000. doi: 10.1145/351240.351266.

Brian F. Cooper, Adam Silberstein, Erwin Tam, Raghu Ramakrishnan, and Russell Sears. Benchmarking cloud serving systems with YCSB. In *Proceedings of the 1st ACM Symposium on Cloud Computing (SoCC)*, pp. 143–154. ACM, 2010. doi: 10.1145/1807128.1807152.

Cursor Team. Introducing Cursor 2.0 and Composer. `https://cursor.com/blog/2-0`, October 2025.

Earendil Works. Pi: A minimal, extensible AI agent for the terminal. `https://github.com/earendil-works/pi`, 2026. Package `@earendil-works/pi-coding-agent`. Accessed 25 September 2026.

Gil Einziger, Roy Friedman, and Ben Manes. TinyLFU: A highly efficient cache admission policy. *ACM Transactions on Storage*, 13(4):35:1–35:31, 2017. doi: 10.1145/3149371.

Run-Ze Fan, Zihao Zhang, Simin Ma, Yebowen Hu, Shouju Wang, Kaiqiang Song, Fei Liu, Hamed Zamani, and Xiaoyang Wang. An empirical study of harness design for coding agents. arXiv preprint arXiv:2609.20804, 2026.

Harbor Framework Team. Harbor: A framework for evaluating and optimizing agents and models in container environments. `https://github.com/harbor-framework/harbor`, 2026. Version 0.20.0 used here; DOI 10.5281/zenodo.20953922.

Jie Huang, Xinyun Chen, Swaroop Mishra, Huaixiu Steven Zheng, Adams Wei Yu, Xinying Song, and Denny Zhou. Large language models cannot self-correct reasoning yet. In *International Conference on Learning Representations (ICLR)*, 2024. arXiv:2310.01798.

Geoffrey Huntley. Ralph Wiggum as a "software engineer". Blog post, `https://ghuntley.com/ralph/`, July 2025.

Zhengyao Jiang, Dominik Schmidt, Dhruv Srikanth, Dixing Xu, Ian Kaplan, Deniss Jacenko, and Yuxiang Wu. AIDE: AI-driven exploration in the space of code. arXiv:2502.13138, 2025.

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. URL `https://openreview.net/forum?id=VTF8yNQM66`. arXiv:2310.06770.

R. Karedla, J. S. Love, and B. G. Wherry. Caching strategies to improve disk system performance. *Computer*, 27(3):38–46, March 1994. doi: 10.1109/2.268884.

Andrej Karpathy. autoresearch: AI agents running research on single-GPU nanochat training automatically. `https://github.com/karpathy/autoresearch`, March 2026.

Minseo Kim. tap: A file-based protocol for heterogeneous LLM agent collaboration. arXiv preprint arXiv:2606.14445, 2026. Accepted to KCC 2026; 3 pages.

Jiahang Lin, Shichun Liu, Chengjun Pan, Lizhi Lin, Shihan Dou, Zhiheng Xi, Xuanjing Huang, Hang Yan, Zhenhua Han, Tao Gui, and Yu-Gang Jiang. Agentic harness engineering: Observability-driven automatic evolution of coding-agent harnesses. arXiv preprint arXiv:2604.25850, 2026.

Fengshuo Liu, Ying Liu, Ruize Sun, Lie Luo, and Siyuan Guo. Coding agents have converged: Why the SWE-bench leaderboard can no longer order its top entries, and what to measure instead. *arXiv preprint arXiv:2609.17394*, 2026. Accepted at ADMA 2026.

Chris Lu, Cong Lu, Robert Tjarko Lange, Jakob Foerster, Jeff Clune, and David Ha. The AI Scientist: Towards fully automated open-ended scientific discovery. arXiv:2408.06292, 2024.

Yanuo Ma, Ben Kereopa-Yorke, and Ben Schultz. Building to the test: Coding agents deliver what you check, not what you requested. arXiv preprint arXiv:2606.28430, 2026.

Mike A. Merrill, Alexander G. Shaw, Nicholas Carlini, Boxuan Li, Harsh Raj, Ivan Bercovich, et al. Terminal-bench: Benchmarking agents on hard, realistic tasks in command line interfaces. arXiv preprint arXiv:2601.11868, 2026.

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:2506.13131, 2025.

OpenAI. Introducing Codex. `https://openai.com/index/introducing-codex/`, May 2025. A cloud-based software engineering agent powered by codex-1. Blog post, 16 May 2025; open-source CLI at `https://github.com/openai/codex`.

OpenHands. Condenser: High-level architecture of the conversation history compression system (LLMSummarizingCondenser), OpenHands software agent SDK documentation. `https://docs.openhands.dev/sdk/arch/condenser`, 2026. Accessed 18 September 2026.

Melissa Z. Pan, Shuo Yang, Negar Arabzadeh, Wei-Lin Chiang, Ion Stoica, and Matei Zaharia. HarnessTax: How much does the harness matter for coding agents? `https://harnesstax.github.io/`, 2026. Blog post with per-model tables; 21 model-harness pairs on Terminal-Bench 2.0 and SWE-bench Lite. Accessed 25 September 2026.

Arjun Panickssery, Samuel R. Bowman, and Shi Feng. LLM evaluators recognize and favor their own generations. In *Advances in Neural Information Processing Systems 37 (NeurIPS 2024)*, 2024. doi: 10.52202/079017-2197.

Sungho Park, Wonjoong Kim, Rongyuan Tan, Jue Zhang, Wook-Shin Han, Pengfei Gao, Chanyoung Park, Yongqiang Yao, Rao Fu, Elsie Nallipogu, Qingwei Lin, Saravan Rajmohan, and Dongmei Zhang. AutoSaddler: Automatic harness optimization with durable updates from agent execution traces. arXiv preprint arXiv:2608.23041, 2026.

Ziheng Qin, Yaxin Lu, Zhangyang Atlas Wang, and Kai Wang. StateM: Reaching 95.3% raw accuracy, or a $15 frontier run, on Terminal-Bench 2.1 via harness scaling. arXiv preprint arXiv:2608.15089, 2026.

Bernardino Romera-Paredes, Mohammadamin Barekatain, Alexander Novikov, Matej Balog, M. Pawan Kumar, Emilien Dupont, Francisco J. R. Ruiz, Jordan S. Ellenberg, Pengming Wang, Omar Fawzi, Pushmeet Kohli, and Alhussein Fawzi. Mathematical discoveries from program search with large language models. *Nature*, 625:468–475, 2024. doi: 10.1038/s41586-023-06924-6.

Siddharth Sambharia. The harness tax: The dead weight inside your coding agent. `https://portkey.ai/blog/the-harness-tax/`, April 2026. Blog post, 13 April 2026.

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 36 (NeurIPS 2023)*, pp. 8634–8652, 2023. doi: 10.52202/075280-0377.

Davide Spadini, Mauricio Aniche, Magiel Bruntink, and Alberto Bacchelli. To mock or not to mock? an empirical study on mocking practices. In *2017 IEEE/ACM 14th International Conference on Mining Software Repositories (MSR)*, pp. 402–412, 2017. doi: 10.1109/MSR.2017.61.

SWE-agent team. mini-SWE-agent: The 100 line AI agent that solves GitHub issues. GitHub repository, `https://github.com/SWE-agent/mini-swe-agent`, v2.4.6, 2026. Accessed 19 September 2026.

Xingyao Wang, Boxuan Li, Yufan Song, Frank F. Xu, Xiangru Tang, Mingchen Zhuge, Jiayi Pan, Yueqi Song, Bowen Li, Jaskirat Singh, Hoang H. Tran, Fuqiang Li, Ren Ma, Mingzhang Zheng, Bill Qian, Yanjun Shao, Niklas Muennighoff, Yizhe Zhang, Binyuan Hui, Junyang Lin, Robert Brennan, Hao Peng, Heng Ji, and Graham Neubig. OpenHands: An open platform for AI software developers as generalist agents. In *International Conference on Learning Representations (ICLR)*, 2025. arXiv:2407.16741.

Yike Wang, Huaisheng Zhu, Zhengyu Hu, Yige Yuan, Zhengyu Chen, Shakti Senthil, Hannaneh Hajishirzi, Yulia Tsvetkov, Pradeep Dasigi, and Teng Xiao. Rethinking the evaluation of harness evolution for agents. arXiv preprint arXiv:2607.12227, 2026.

Johannes Wehrstein, Timo Eckmann, Matthias Jasny, and Carsten Binnig. Bespoke OLAP: Synthesizing workload-specific one-size-fits-one database engines. *arXiv preprint arXiv:2603.02001*, 2026. Accepted to VLDB 2026.

Scott Wu. Introducing Devin, the first AI software engineer. Cognition blog, `https://cognition.com/blog/introducing-devin`, March 2024. Product page: `https://devin.ai`.

John Yang, Carlos E. Jimenez, Alexander Wettig, Kilian Lieret, Shunyu Yao, Karthik Narasimhan, and Ofir Press. SWE-agent: Agent-computer interfaces enable automated software engineering. In *Advances in Neural Information Processing Systems 37 (NeurIPS)*, 2024. doi: 10.52202/079017-1601. arXiv:2405.15793.

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. arXiv:2210.03629.

Justin Young. Effective harnesses for long-running agents. Anthropic engineering blog, `https://www.anthropic.com/engineering/effective-harnesses-for-long-running-agents`, November 2025.

Xing Zhang, Guanghui Wang, Yanwei Cui, Wei Qiu, Ziyuan Li, Bing Zhu, and Peiyang He. Guardrails beat guidance: A large-scale study of rules, skills, and persistent configuration for coding agents. arXiv preprint arXiv:2604.11088, 2026.

A THE FIVE LAYERS

Figure 3 draws the five layers of Section 2.1 and the external state each one touches; the paragraphs below describe each layer.

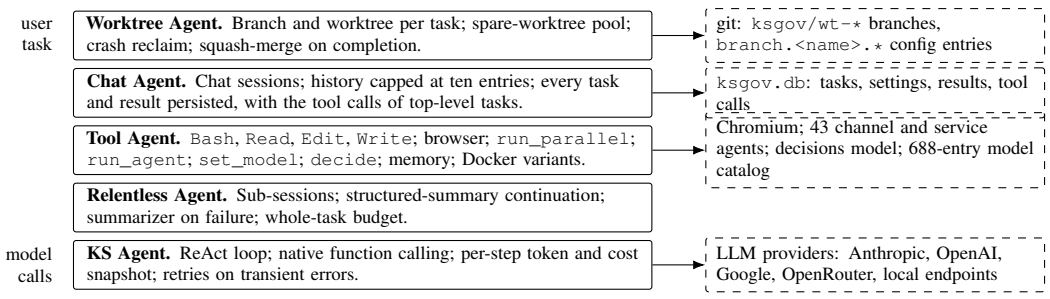


Figure 3: The five layers of KS Gov and the external state each one touches. Each layer wraps the one below it, from the worktree that holds a chat down to the KS agents a relentless agent runs until the task is done.

The KS Agent, innermost, runs a ReAct loop (Yao et al., 2023) over the model's native function-calling API; tools are Python callables and `finish` ends the loop. After every model response, it computes cost from a per-model price table and publishes tokens, cost, and step count; before each step, it checks its budget and calls a hook through which the enclosing layer enforces the task's total budget.

The Relentless Agent exists because our tasks outgrew the context window before they were done. It runs such a task as a sequence of sub-sessions: when the executor judges that it is about to run out of context, it calls `finish` with `is_continue=True` and a chronologically ordered summary of what it did, with reasons and code snippets, and a fresh sub-session starts from the prior summaries (capped at 60,000 characters) under an instruction not to redo completed work.

The Chat Agent gives each task a chat with a stable ID and prepends the chat's prior tasks and results to the prompt, capped at ten entries. After each task, it records the model, working directory, version, tokens, cost, and whether parallel sub-agents or a worktree were used; the process rows of the case studies (Appendix C) are mined from that record.

The Worktree Agent runs every task that may edit files on its own branch in a worktree under `.ksgov-worktrees/`, so the user's working tree is untouched while the task runs; at merge time the layer stashes the user's changes, checks out the original branch, squash-merges the agent's commits made after a baseline commit of the user's uncommitted changes, and pops the stash. A background pool keeps one spare worktree per repository ready, and a reclaim pass squash-merges worktrees whose owner process has died when they pass its safety guards.

The Tool Agent adds the tools. We kept the names Claude Code uses (`Bash`, `Read`, `Edit`, `Write`) so that models see a vocabulary they already know. `Edit` replaces an exact string that must occur once in the file; `Edit` and `Write` refuse an existing file the tool has not shown the model or written itself. A browser tool drives Chromium through Playwright. `run_parallel` spawns one Chat Agent per task string in a thread pool, each with its own context, and folds their tokens and cost into the parent task; `run_agent` dispatches a sub-task to one of the 43 channel and service agents (Slack, Gmail, GitHub, ...); a memory of Markdown pages with a semantic index has search, read, write, and delete tools. `set_model` hands the live conversation to another model from the 688-entry catalog without restarting the task. `decide(state, questions)` sends a text and typed questions to a non-generative decisions model (`openrouter/~typesafe/jev-latest`) and returns probabilities; one such question classifies a task before it starts, and a task that neither develops software nor searches the Internet runs on `SYSTEM_LITE.md`, a 753-word prompt with the identity and output contract and none of the engineering rules.

B THE PROMPTS

The first HydraKV task prompt of Section 3.1 (model and budget clauses omitted):

```
Can you perform adversarial AI Discovery for the task at ./KV_TASK.md so the task goals are
met? [...] You MUST not stop until the goals are met. Generate adversarial workload variants
to ensure the engine runs fast on the variant workloads. [...] Then generate a new workload
and repeat the process until the engine can achieve the goals on the new test workload on
which AI discovery was not performed.
```

The HydraKV tasks ran under the system prompt's earlier discovery instruction, the same paragraph in all six:

```
Mandatory Instructions (MUST FOLLOW): You will be exploring, implementing, and evaluating
novel ideas while doing AI discovery or auto research or software optimization. Analyze the
data/information provided to you and search the internet extensively to propose the first
few ideas. Implement and experiment with each of your proposals. Record the ideas you used
to achieve user-specified metrics in a file, along with the metric values, so you can avoid
repeating ideas that have already been tried and/or failed. You can also use the file to
combine ideas that have been successful in the past. MAKE SURE THAT YOU DO NOT DO REWARD
HACKING OR CHEATING IN THE MODELS OR AGENTS YOU ARE IMPLEMENTING TO FIT DATA. YOUR SOLUTION
MUST GENERALIZE BEYOND THE DATA PROVIDED. You MUST search the internet at every step to find
new ideas. Use powerful models (if available), such as claude-fable-5, gpt-5.6-sol-xhigh, to
explore diverse kinds of novel ideas.
```

The current discovery-loop instruction of Section 2.3 reads as follows. The TPC-H task ran an earlier wording of the same seven steps ("record baseline metric" and "go to 4" without "not explored before"), with a one-sentence adversarial-testing paragraph.

```
1. read + profile the data/tests/baseline, record baseline metrics
2. web-search for SOTA approaches, papers, repos, issues
3. Write ideas and rationale in ./tmp/ideas.md
4. Pairwise judge the ideas to find a winning idea.
5. Implement -> run real end-to-end evaluation -> log idea, aspect of improvement, and
metrics in ./tmp/explored-ideas.md
if better: keep, and try composing with prior winners on different aspects
if worse: mark as failed so it is never retried
6. search again for fresh ideas not explored before and based on previous experience and
exclude ideas that have been explored in ./tmp/explored-ideas.md; go to step 4
7. stop when the user's metric goal is met, with a held-out / generalization check to prove
it is not overfit
```

The adversarial-testing paragraph that follows it reads:

```
Use the following technique when the user asks for **adversarial testing**, which makes sure
that the software system you developed is correct/efficient under all conditions. Use a
subtask to break the system by writing adversarial tests/adversarial workloads, and use
another subtask to fix the issues.
```

The cross-vendor review instruction of Section 2.4 reads:

```
Use 'claude-fable-5' model for all tasks, including software development. Use 'gpt-5.6-sol'
(not codex) using `run_parallel` tool for a thorough read-only review and debugging of the
other model's work. Thoroughly check whether the other model has missed any code or wiring
or introduced any bugs. Use at most 50% of the task budget in gpt-5.6-sol for reviewing and
debugging, and ask the model not to invent new problems. Use the model names literally
without hallucinating new model names.
```

The read-before-modify rule of Section 2.2, the one rule the tool layer also enforces, reads:

```
Read before modify rule - NON-NEGOTIABLE: You MUST call Read(file_path) on every existing
file BEFORE calling Edit(file_path) on it or overwriting it with Write(file_path). Never
modify a file you have not Read in the current session.
```

C Case-Study Process Record

Table 4 gives the process rows mined from the task database and the results of the two case studies of Section 3; Table 3 gives HydraKV's throughput under the workload mixes of its hardening task.

Table 3: HydraKV under the 7 workload mixes of the hardening task (`WORKLOAD_HARDENING.md`), 250 million keys each. Throughput on the scored mix is the July 21 reference-node re-run; the other mixes were run for correctness without tuning, and their throughput is reported as measured. "Steering message" marks the two mixes the user asked for while the task ran.

Workload mix	Proposed by	Mops/s	Integrity auditor
`A_50_50`	scored	5.50	0 fail
`RMW_100`	agent	0.30	0 fail
`B_95_5`	agent	2.93	0 fail
`C_100_0`	agent	2.78	0 fail
`W_0_100`	steering message	5.09–5.13	0 fail
`TIMESERIES_HD`	agent	0.08	0 fail (1.28M deletes: 0 resurrections, 0 lost keys)
`W95_5_WRITE`	steering message	5.17–5.24	0 fail

Table 4: The two case studies. Process rows come from the task database (development tasks identified by hand; hours are wall-clock; a continuation is one summary-and-restart sub-session). Results: HydraKV is the median of three cold-cache 30-second YCSB-A runs (50% reads, 50% blind upserts, Zipfian θ=0.95, 250 million 64-bit keys with 100-byte values, 16 workers) on a GCP `n2-standard-64` (64 vCPUs, eight local NVMe SSDs in RAID0) under an enforced 8 GiB resident-memory cap; the FASTER figure is the one stated in the task's benchmark description (no FASTER run log exists). TPC-H is the sum of per-query median execution times over five repetitions on a 32-core Linux machine, validated against DuckDB at the paper's tolerance (10^{-2}); seed 999 was held out.

		HydraKV	**TPC-H engine**
Process	top-level tasks; agent hours; model spend	6; 17.2; $352	3; 2.4; $90
	steps; longest task (hours); continuation sub-sessions	1,353; 5.8; 5	475; 1.3; 0
	steering messages; review sub-agents	7; 9	0; 1 (the first review ran as a separate task)
	kept idea groups; workloads tried; bugs fixed in the workload task (from workloads and tests / from the reviewer); e2e tests	5; 7; 8 (4 / 4); 28	4 change groups; 4 seeds; 2 review defects
Result	metric	YCSB-A Mops/s (higher is better)	SF10 seed 42 total, ms (lower is better)
	baseline	0.93 (FASTER, as stated)	3,132.2 (single-threaded artifact)
	agent's engine	5.50	90.8
	other seeds (7 / 123 / 999 held out)		92.5 / 82.1 / 85.1
	SF1 seed 42, baseline to agent		276.8 to 14.2
	side cost	RSS 7.9 GB of 8 GiB	ingest 52.5 to 63.0 s (20%)