WHEN CODE CRAWLS: BENCHMARKING PROACTIVE OPTIMIZATION IN CODING AGENTS

Anonymous authors
Paper under double-blind review

ABSTRACT

On benchmarks such as SWE-bench, coding agents now resolve a substantial share of real software tasks end to end. What those benchmarks score is functional correctness: runtime is not part of the score, so a program that returns the right answer after an hour still passes. Benchmarks that measure runtime either score the efficiency of generated code or ask for a performance improvement outright. In production, speed is part of usefulness, and engineers fix a step that runs far longer than it should instead of letting it finish. We study whether coding agents *self-initiate* performance optimization when performance is not an explicit task objective. We introduce **CrawlBench**, which runs the same repair task under two framings: the slowness is either stated in the prompt or left for the agent to discover and repair on its own. Every pathological case is paired with a healthy twin, so efficiency is the only quantity that varies. The reported pool is 192 cases: 88 single-bug Python artifacts, 12 held-out decay tiers, 19 two-bug stacks, 20 regressions mined from merged performance-fix pull requests, and 53 single-bug asymptotic replicas (28 in C, 25 in JavaScript). Across ten production models, agents told the program is slow fix 93.6–97.7% of the single-bug pool; run on the identical task without the hint, the same agents fix 23.5–90.2%. The gap spans $+7.2$ to $+72.0$ points. On all ten models, each run for three seeds, the implicit rate has sd ≤ 4.5 points. Every model repairs 93.6–97.7% once told. Implicit success still requires the agent to discover the slowness, judge it worth fixing, and modify the code unprompted. That self-initiation is what fails. Failure-mode triage finds that a *miss* is the largest failure for every model, at 52–88% of unfixed implicit episodes: the submission stays correct but slow. Fast-and-wrong submissions are 2.0% of those unfixed episodes. We release the benchmark, harness, and all cases.

1 INTRODUCTION

On benchmarks such as SWE-bench (Jimenez et al., 2024), frontier coding agents (Yang et al., 2024; Wang et al., 2025; Xia et al., 2024; Zhang et al., 2024) resolve a substantial share of real GitHub issues end to end. What these benchmarks score is functional correctness: does the patch pass the hidden tests, does the agent close the issue (Liu et al., 2024b)? Runtime is not part of the score, so a program that returns the right answer after an hour passes. In production the same program is useless: an API that answers in an hour has already lost its user, and a CI pipeline that takes an hour per run has every developer context-switching away. Neither program is incorrect; both are unusable at that speed. Correctness-only metrics are blind to this failure, and an agent optimized against them has no reason to ask how long its own code takes, even though responsiveness is much of what makes software usable (Nielsen, 1993; Dean & Barroso, 2013) (Figure 1).

Humans do not behave this way. An engineer carries an implicit model of how long a step *should* take, derived from the size of the input and the cost of each operation. When a command runs far past that estimate, she does not wait it out: she kills it, inspects the code, and fixes the cause long before the run would have finished. This *proactive efficiency awareness* is a routine part of productive software work. Existing benchmarks measure the efficiency of generated code (Huang et al., 2024; Du et al., 2024; Qing et al., 2025) and performance improvements under explicit optimization requests (Shypula et al., 2023; He et al., 2026; Shetty et al., 2025; Ma et al., 2026), where the task itself asks for speed. Whether an agent initiates that work when performance is not mentioned is a different question, and to our knowledge no benchmark isolates it. Recent work scores agent *behav-*

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.

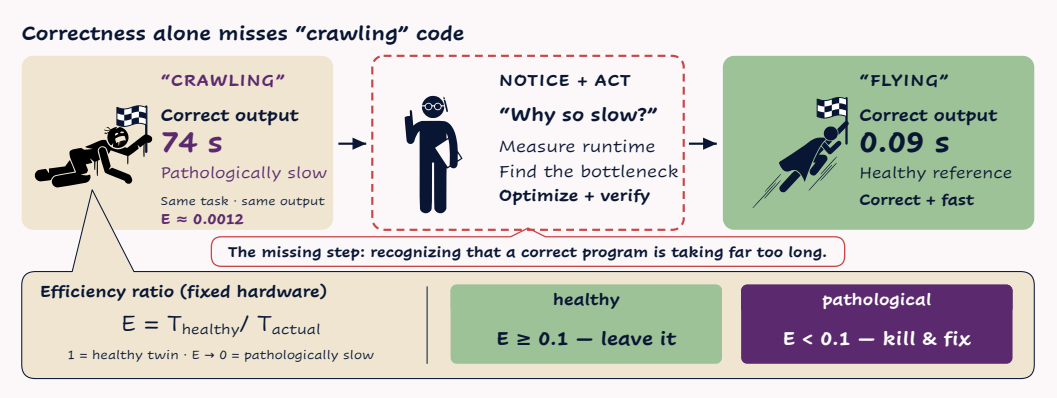

Figure 1: From "crawling" to efficient code: both programs solve the task, but the crawling one takes 74 s versus 0.09 s for its healthy twin (construction-time runtimes).

ior during the episode (Xi et al., 2025; Tang et al., 2026; Podivilov et al., 2026; Chen et al., 2026; Yao et al., 2025; Ma et al., 2024; Xu et al., 2025), but none asks whether an agent will optimize running code when no task asks it to.

We introduce **CrawlBench** (Figure 2), a benchmark for proactive optimization of pathologically slow code. Every case is posed twice. ❶ **Explicit**: the task states that the program is pathologically slow, and the score measures repair ability alone. ❷ **Implicit**: the same natural task with no hint of slowness. Explicit measures what a model can fix once told. The benchmark then measures the composite that implicit runs still require: perceiving the runtime, judging it pathologically slow, and choosing to optimize without a mandate.

We measure ground truth from runtimes. No annotator assigns the label. Efficiency is the wall-clock ratio $E = T_{\text{healthy}}/T_{\text{actual}}$ on fixed hardware, and a case ships only when re-measurement confirms its healthy or pathological label. A disagreement between the measured regime and the assigned label rejects the case. Each pathological case has a healthy twin for the same task, with correctness, task, and scaffold held fixed, so efficiency is the only quantity that varies and a failure is attributable to that dimension.

Our main result is a large *explicit–implicit gap*. On the 88 single-bug cases, every one of ten models repairs most cases once the prompt discloses the slowness, and far fewer when it must find that slowness unaided (Figure 3). The drop spans $+7.2$ to $+72.0$ points. On all ten models, each with three seeds, it reproduces with implicit-rate sd ≤ 4.5 points. On two-bug stacks the both-bug rate does not follow the single-bug ranking, and on regressions mined from merged performance-fix pull requests the pool separates where the task never says a regression exists. The deficit is *self-initiation*. Repairing a named bottleneck is mostly engineering, and no correctness test asks the agent to begin that work. A *miss*, code that stays correct but slow, is the largest unfixed outcome for every model, at 52–88% of unfixed implicit episodes.

❶ We define and measure *proactive efficiency awareness*: self-initiation of performance optimization when that goal is not explicit, separable from repair capability.

❷ We release CrawlBench. The reported pool is 192 cases: 88 single-bug artifacts, 12 decay tiers, 19 multi-bug stacks, 20 mined performance-fix pull requests, and 53 single-bug asymptotic replicas (28 in C, 25 in JavaScript), with measured, self-checking ground truth.

❸ We characterize the deficit across ten production models (explicit 93.6–97.7% vs. implicit 23.5–90.2%). The dominant unfixed outcome is a *miss*, a submission that stays correct but slow.

2 Related Work

Correctness benchmarks. HumanEval (Chen et al., 2021), MBPP (Austin et al., 2021), and SWE-bench (Jimenez et al., 2024) anchor a large family of correctness tests: stronger oracles and refreshed

or broader task coverage (Li et al., 2022; Liu et al., 2023; Jain et al., 2025; Zhuo et al., 2025; Lai et al., 2023; Mündler et al., 2024; Zhao et al., 2025; Liu et al., 2024b), repository-scale context completion (Ding et al., 2023; Liu et al., 2024a), and SWE-bench variants and agent–computer interfaces (OpenAI, 2024; Zan et al., 2025; Yang et al., 2025; Zhang et al., 2025; Yang et al., 2024). All of these score reaching a correct final state: a program that is correct but quadratically slow passes. None measures runtime, rewards efficiency, or forces the model to react to slowness.

Process and behavior benchmarks. A recent line of work scores agent behavior during the episode: interactive web and operating-system environments (Zhou et al., 2024; Xie et al., 2024), analytical progress rates (Ma et al., 2024), freelance software engineering (Miserendino et al., 2025), machine-learning engineering (Chan et al., 2025), and full workplace simulations (Xu et al., 2025). AgentLens (Podivilov et al., 2026) reviews complete coding-agent trajectories with LLM judges and formal verifiers; ProAgentBench (Tang et al., 2026) formalizes *proactivity* as a per-timestep "when to assist" decision scored with precision/recall against interruption cost; ATOBench (Chen et al., 2026) pairs native and adversarially perturbed episodes aligned at a changed observation to trace how agents verify under deceptive evidence. CrawlBench adopts the same stance in what it isolates: whether an agent self-initiates optimization of running code when the task does not ask for it. The score itself is the final submission, correct and fast. Perception, judgment, and action are not scored as separate stages. Its paired healthy/pathological candidates play the role that ATOBench's paired episodes and AgentHarm's benign counterparts (Andriushchenko et al., 2025) play: isolating the variable of interest while holding task and scaffold fixed.

Program optimization and performance bugs. Program repair predates LLMs (Le Goues et al., 2012), and LLM agents now repair bugs end to end (Bouzenia et al., 2025); for efficiency specifically, Shypula et al. (2023) ask whether LLMs can rewrite a given program to be faster, measuring the speedup of the produced code, and follow-up benchmarks score the efficiency of generated code directly, in Python (Huang et al., 2024; Du et al., 2024), across languages (Qing et al., 2025), for parallelism (Nichols et al., 2024), and against per-language energy profiles (Pereira et al., 2017). At repository scale, SWE-Perf (He et al., 2026) evaluates optimization of target functions on real performance-fix PRs, GSO (Shetty et al., 2025) measures improvements against expert optimizations on provided codebases, and SWE-fficiency (Ma et al., 2026) further requires investigation and bottleneck localization on real workloads; these benchmarks explicitly request performance improvements but do not necessarily identify the bottleneck for the agent. HELM (Liang et al., 2023) lists efficiency as one of seven metric categories but makes it a property of the *system*, not a target for the model to act on. These test the *fix* half of our problem: performance is an explicit objective, whether or not the bottleneck is named. The underlying slowness is a documented production phenomenon (Jin et al., 2012; Nistor et al., 2015; Zaman et al., 2012; Crosby & Wallach, 2003; Davis et al., 2018; Shen et al., 2018; Davis et al., 2019; Selinger et al., 1979; Marcus et al., 2019; 2021); our cases inject these bugs into real libraries so that a fix requires runtime judgment. None of these settings tests whether an agent will self-initiate that work on code it is already running when speed is absent from the task.

3 The Efficiency-Awareness Problem

An agent executes a program P on an input of size N and must decide whether P is efficient enough to let run or so slow that it should be killed and replaced.

Efficiency metric. On fixed hardware, let $T_{\text{healthy}}(N)$ be the wall-clock time of the healthy twin and $T_{\text{actual}}(N)$ the time of the program under evaluation. The time-to-solution ratio is

$$E = \frac{T_{\text{healthy}}(N)}{T_{\text{actual}}(N)}. \tag{1}$$

$E = 1$ means the program matches its healthy twin; $E \to 0$ means it is pathologically slow. Because both times are measured on the same machine, E is a distance from that healthy twin on that machine, not an absolute number of seconds (Section 6).

Ground truth at construction time. Humans never label the programs; we determine the regime from E and a threshold of 0.1: **healthy** if $E \geq 0.1$ (close to the healthy twin; leave alone), and

pathological if $E < 0.1$ or the program times out (must kill and optimize). Each task ships two candidate programs engineered to land in the two regimes, and the construction-time measurement tool validates that the *measured* regime matches the *declared* one. Ground truth is therefore a property of the construction. No annotator assigns the label. This gate admits the synthetic artifacts. The real-PR pool (Section 4) is defined from merged performance-fix pull requests and does not use the $E < 0.1$ cutoff.

Initiation and correction. An implicit episode can fail at three steps that the fix rate does not separate: never gathering timing evidence (*perception*), gathering it without concluding the runtime is pathologically slow (*judgment*), or reaching that conclusion and still declining to modify code no one asked it to optimize (*action*). We call the composite an **initiation failure**. Its complement is the **correction failure**: the program needs optimizing, and the attempted fix damages behavior, so the submission is no longer correct. Explicit framing scores correction alone, so correction failures cannot explain the implicit collapse. Trajectory triage (Section 5.2) and the timing-visibility control (Appendix A4) localize where the initiation pipeline breaks.

4 CRAWLBENCH

CrawlBench is a real-repository *artifact* layer: real libraries with injected or historically mined slowness bugs, run under two framings that share the same efficiency definition. The synthetic artifacts use the regime threshold of Section 3; the real-PR pool is admitted separately, as specified below. It produces the main results of this paper. Figure 2 summarizes the benchmark design; Figure 3 reports the main-benchmark results, and Figure 6 the extension experiments.

Bug categories. Each bug belongs to one of four categories: *asymptotic complexity* (the algorithm's work grows too fast with input size), *execution-model constants* (same asymptotics, but every operation runs on a costlier execution path), *external I/O latency* (waiting on the outside world that a CPU-centric view ignores), and *declarative-engine cost* (the engine, not the agent, chooses the execution plan, and the agent does not know its cost model). Reporting results per category turns a pass rate into a profile: which classes of inefficiency a model handles and which it misses.

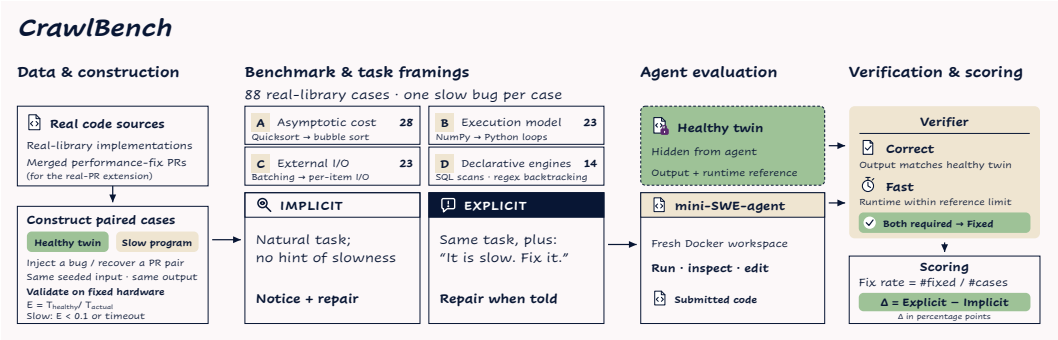

Figure 2: CrawlBench overview. Real-library code and performance-fix PRs supply paired healthy and slow artifacts, validated with shared inputs on fixed hardware. The main benchmark contains 88 single-bug cases across four categories, each evaluated under implicit (no slowness hint) and explicit (repair when told) framings. A submission counts as fixed only if it is both correct and fast; Δ is explicit minus implicit fix rate in percentage points.

Artifact layer. The artifact layer measures *proactive fixing* on real code. Each case is a real library with a single pathological-slowness bug injected, packaged as an *artifact*: a buggy workspace for the agent to work in, a pristine healthy repo used only by the verifier as its reference, a deterministic seeded input generator in the QuickCheck style (Claessen & Hughes, 2000) (so every comparison runs on identical inputs), and two task files, one per framing. Under implicit framing, a fix counts only if the agent runs the tool, notices that it is pathologically slow, and repairs it while preserving behavior. Explicit framing states the slowness, so that run scores repair alone.

Every single-bug artifact is a "same result, different time" scenario by construction: the buggy and pristine versions produce identical output and differ only in runtime, by a factor of 16–10730×, so efficiency is the only quantity under test. The verifier accepts a submission only when its output matches pristine and it runs fast; a submission that changes behavior scores as a correctness break. The framings and the verifier contract are defined below.

Real-PR-derived cases. The `R*` cases are mined from merged performance-fix pull requests. We did not inject these bugs by hand. The buggy state is the repository at the PR's merge-base and the pristine state is that base plus the PR's own diff, so the slowness is the bug the maintainers fixed. They ship as `pristine.patch` / `buggy.patch`, a `bench_cmd`, and an optional `setup.sh`, and they use the same `verify.py` contract as the synthesized cases.

Verification. For a candidate workspace, `verify.py` reports three booleans: `correct` (output matches the pristine version on the same seeded input, a differential check (McKeeman, 1998)), `fast` (runtime within a $5\times$ factor of pristine on the synthetic artifacts, and within a $2\times$ factor on the twenty real-PR cases; the healthy twin is re-measured on the same held-out input and serves as T_{healthy}), and `fixed` = `correct` AND `fast`. On the synthetic artifacts the verifier bar is stricter than the construction-time regime threshold: `fast` demands $E \geq 0.2$, whereas a case is declared pathological at $E < 0.1$. The 88 single-bug programs run 16–10730× slower than pristine, five of them between $16\times$ and $20\times$. On those artifacts the two thresholds disagree only in the 5–10× band occupied by a partial repair, such as one bug of a stacked case left unfixed (Section 5.3). The twenty real-PR cases are a separate pool. Admission follows the maintainers' merged performance fix. A mined regression can be only a few times slower than pristine, so the $E < 0.1$ gate is not applied to this pool, and the fast bar is $E \geq 0.5$ (within $2\times$). A timeout is scored at the timeout and still fails `fast`. The scoring input is regenerated at verification time from a held-out seed derived from the case name, while the agent's workspace always receives the default-seed input, so a submission specialized to the input it observed cannot score as fixed.

Framing and the proactive-awareness score. Each artifact is run under two framings. Under *explicit* framing the task states that the program is pathologically slow and asks for a fix; this measures the ability to *correct*. Under *implicit* framing the task is a natural request with no hint of slowness. The *proactive-awareness score* for an artifact is explicit_fix − implicit_fix: the gap between fixing when told it is slow and fixing when left to act on its own. This gap is the paper's central quantity. It scores the initiation pipeline of Section 3 end to end. Perception, judgment, and the decision to act all sit behind the outcome, and the score does not separate them.

Harness. Each (artifact, framing) pair launches a fresh mini-SWE-agent (Yang & Jimenez, 2025) episode (a minimal observe-act loop in the ReAct style (Yao et al., 2023), deliberately without reflection or self-critique machinery (Shinn et al., 2023)) in an isolated workspace containing the buggy repo and the generated input. The agent runs the tool, may edit the repo, and submits. Episodes run in per-case Docker containers with declared CPU/memory limits so a runaway candidate cannot wedge the host or leak across cases; the agent's final workspace is verified by `verify.py`. The agent's single tool is a bash command: the system prompt instructs it to bound any potentially long-running command with `timeout`, choosing the limit itself. The requirement is itself a time judgment: to write the bound, the model must decide in advance how long the command may run, so duration estimates enter the loop through the agent's own output. Observations carry only stdout and the return code; the scaffold adds no timing information. An agent that wants a runtime must measure it itself, as in a profiler-driven workflow (Graham et al., 1982).

5 Experiments

5.1 Setup

Models and agent. We evaluate ten production coding models spanning implicit fix rates of 23.5–90.2% on the single-bug pool (Gemini 3.7 Flash, GPT 5.6 Sol, GPT 5.6 Luna, GLM 5.3 Flash, Kimi K3, Claude Opus 5, Qwen3.8 27B, GLM 5.2, DeepSeek V4 Pro, and DeepSeek V4 Flash), each driven by mini-SWE-agent. All models see identical tasks, scaffolds, and budgets. All ten are run

under three seeds on the 88 single-bug cases, the decay tiers, the stacks, the real-PR cases, and the cross-language replicas.

Cases. The main study uses 88 artifact cases, binned by case-id prefix into 28 asymptotic-complexity, 23 execution-model, 23 external-I/O, and 14 declarative-engine cases. Four of those bugs (sorting, NumPy vectorization, buffered writes, and SQL query planning) each have three further input-size tiers. Those twelve cases are held out of the 88 and reported only as the decay-strength ablation (Section 5.3). Extensions add 19 two-bug stacks, 20 mined performance-fix pull requests (the `R*` family, Section 4), and 53 single-bug asymptotic replicas (28 in C, 25 in JavaScript). These are the cases that enter a reported pool (192). Each single-bug and real-PR case is a real library (SQLAlchemy, aiohttp, sympy, and others) with one injected or historically fixed bug; each stack carries two independent bugs. Several projects recur with different bugs. On the 88 single-bug cases, healthy versions run in a median of 0.255 s and pathological versions run 16–$10730\times$ slower, so every one of those 88 meets $E < 0.1$.

Protocol. Each artifact is run under both framings. Episodes have a wall-clock limit of 1200 s (3600 s for the real-PR family), per-command timeouts, and per-case Docker limits; a case not fixed in time counts as not-fixed. Serving-endpoint incidents were detected from trace timestamps and the affected runs re-executed under the same seed (Appendix A3). Fix rate is the fraction of cases whose final workspace is both `correct` and `fast`.

Prompts. Appendix A7 prints the shared system prompt and both task prompts. The system prompt gives the agent one `bash` tool and one call per turn, requires a `timeout` bound on any command that might run long, and ends the episode with a specific echo. The implicit task addresses the agent as the tool's author and asks only for a run and a report of the output. The explicit task appends that the tool is slower than it should be, a coarse runtime budget from the healthy twin, the bug count (one, or two on the stacks), and a request to re-run and confirm the same output. The explicit budget is a coarse phrase, and it is not the same phrase on every case: cases whose healthy runtime stays under a second say "well under a second," while slower healthy twins say "about a second" or "a few seconds." The twenty real-PR cases use a standalone explicit prompt: the checkout is pathologically slow, one performance bug is present, and the fix must preserve the output.

5.2 Results and Analysis

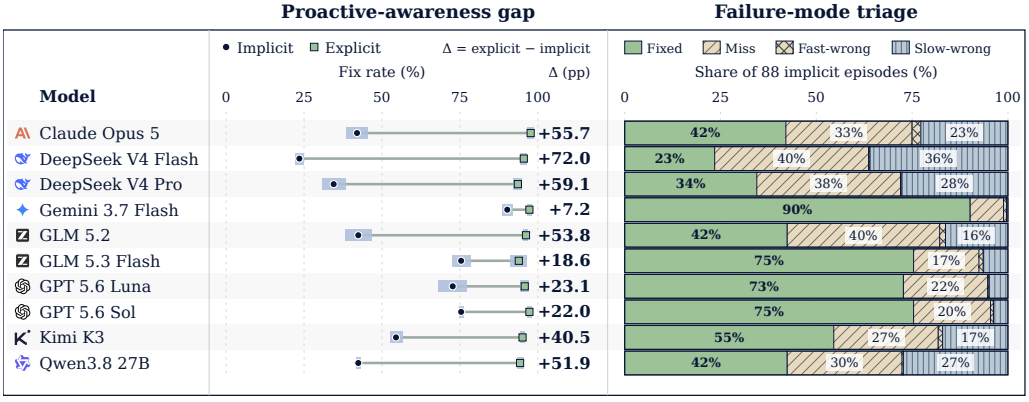

Figure 3: Main-benchmark results across ten models. Left: implicit and explicit fix rates on the single-bug artifacts; Δ is explicit minus implicit fix rate in percentage points. Right: triage of each model's implicit episodes into *fixed*, *Miss* (correct but slow), *Fast-wrong* (incorrect but fast), and *Slow-wrong* (incorrect and still slow).

The explicit-implicit gap. Under explicit framing the models fix 93.6–97.7% of the 88 artifacts: when told the program is slow, every model repairs the same large majority of cases. Under implicit

framing the same models fix 23.5–90.2%, a $3.8\times$ spread under identical code, budgets, and scaffold. The implicit prompt omits the slowness disclosure, the bug count, and the coarse runtime budget that the explicit prompt adds. Explicit repair is already high for every model. A healthy-twin control runs the same agents on the pristine repos: false-positive rewrites of already-fast code occur in 9% and 1% of episodes and damage working code in 2% and 1% (Appendix A5). The drop sits in the initiation pipeline of Section 3. On all ten models, each with three seeds, the implicit rate reproduces with $\sigma \leq 4.5$ points (Figure 3, left panel; Table A1 in Appendix A1).

Failure-mode triage: mostly correct but slow. We replay every unfixed implicit episode using the agent's exit status and the verifier's output (Figure 3, right; Table A2). A *miss* submits a workspace that is still correct but not fast, the initiation failure of Section 3. Miss is the largest single failure mode for every model, at 52–88% of unfixed episodes. A *break* submits incorrect output and realizes the correction failure: the attempted fix damaged behavior. Breaks are 38% pooled. *Fast-and-wrong* is rare: 2.0% of unfixed episodes, with over half of those in the execution-model category, where vectorization or rescaling silently changes numerics. *Slow-and-wrong* dominates breaks (94%): the agent attempted a repair, broke the output, and still submitted slow code. Every slow-and-wrong submission remains much slower than its healthy twin, so the agent broke behavior and still missed the fast bar. Agents leave slow code in place. Fast-but-wrong submissions stay rare, so a correctness-only score still passes these episodes.

In nine inspected miss trajectories (DeepSeek V4 Flash, DeepSeek V4 Pro, Kimi K3), eight left the hot path byte-identical to the buggy source and the ninth removed a helpful index, yet all submitted with correctness-only justifications. Across unfixed implicit episodes, the hot path stays byte-identical to the buggy source in 95.3% of episodes (Appendix A2).

	Asymptotic	Execution	External I/O	Declarative
Claude Opus 5	56	23	38	52
DeepSeek V4 Flash	30	10	16	45
DeepSeek V4 Pro	30	29	35	52
Gemini 3.7 Flash	96	74	96	95
GLM 5.2	36	39	45	57
GLM 5.3 Flash	81	62	84	71
GPT 5.6 Luna	79	54	81	79
GPT 5.6 Sol	70	61	86	93
Kimi K3	55	49	54	64
Qwen3.8 27B	46	26	48	52
Median	**55**	**44**	**51**	**61**

Figure 4: Implicit fix rate (%) per model and bug category (columns: asymptotic complexity, execution model, external I/O, declarative engine); darker cells are higher, and the bottom row gives medians.

Category profile. Figure 4 gives each model an implicit-fix profile across the four categories (Table A3). Two patterns are stable across the pool. First, the asymptotic-complexity and declarative-engine categories top the pool medians (55% and 61%, vs. 51% for external I/O and 44% for the execution-model category): asymptotic-complexity and declarative-engine bugs are recognizable antipatterns in the code itself (a quadratic inner loop, a missing index, a catastrophic regex) that a code review can catch, whereas external-I/O and execution-model code is locally idiomatic (per-item requests, plain scalar loops), and only the aggregate cost or the wall clock reveals the problem, so an agent without a timing habit sees code that merely "works." Six of ten models peak on the declarative-engine category. Second, the execution-model category has the lowest floor: every model except Gemini 3.7 Flash, GLM 5.3 Flash, and the GPT 5.6 pair fixes execution-model bugs unprompted less than half the time. The spread between the highest and lowest category medians (61% vs. 44%) is the *category* axis of the gap.

5.3 Ablations and Extensions

Decay-strength gradient. Four families (sorting, NumPy vectorization (Harris et al., 2020), buffered writes, and SQL query planning) ship a *decay-strength gradient*. It tests whether the initiation rate tracks the magnitude of the slowdown or only reacts to a sufficiently long wall-clock hang: three tiers whose `buggy/` and `pristine/` code are byte-identical and whose benchmark input size alone scales the pathological runtime to ≈15 s, ≈60 s, and ≈110 s across the four families. These twelve tier-cases are held out of the 88-case main pool and reported only in this ablation. Figure 5 reads implicit fix rate as a function of decay strength at fixed category, so a rise comes from the slowdown (Table A4). The bug and the task stay fixed. Pooled over ten models, implicit fix rates rise from 20% at the ≈15 s tier to 55.8% at the ≈60 s tier and 80% at the ≈110 s tier, and every model gains from $+25$ to $+100$ pp between the mildest and strongest tier. The slope is family-dependent:

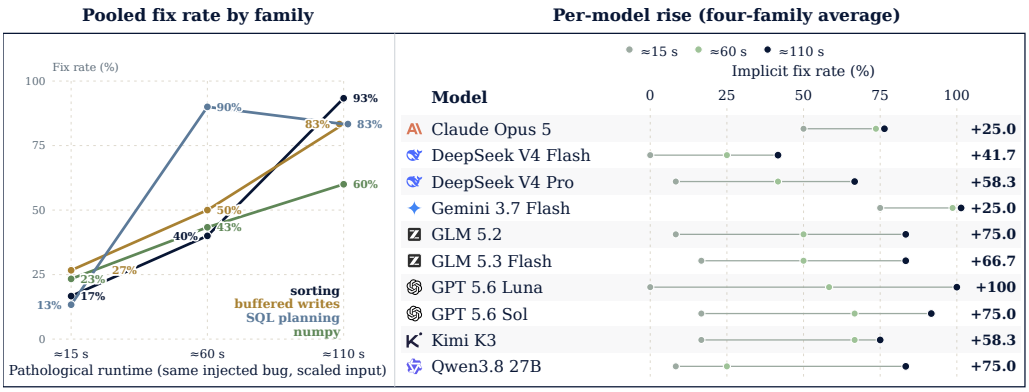

Figure 5: Decay-strength gradient. Left: pooled implicit fix rate for the four severity families as the same injected bug is made slower; within a family the `buggy/` code is byte-identical across tiers and only the benchmark input size changes (≈15/60/110 s pathological, healthy under 3.3 s; the task text names no pathological duration). Each point pools 30 implicit episodes (ten models, three seeds each). Right: each model's four-family average at the three tiers (grey/sage/navy = ≈15/60/110 s); every model repairs more episodes at the strongest tier than at ≈15 s.

SQL planning peaks at the ≈60 s tier (90% pooled) and is 83% at the ≈110 s tier, while NumPy vectorization stays below two-thirds even at the ≈110 s tier.

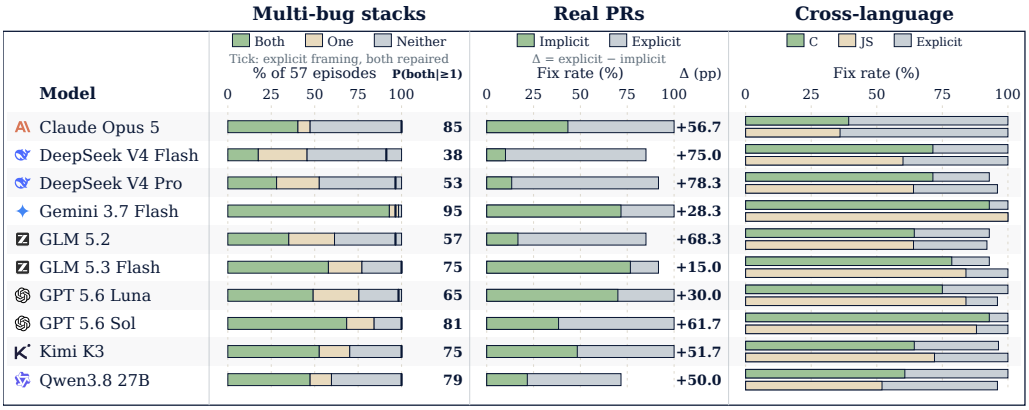

Figure 6: Extension experiments across ten models, three seeds each. Left: each model's 57 two-bug stack episodes split into repairs of both (sage), one (gold), and neither (grey) bugs; the tick marks both-bug repair under explicit framing, and the right column gives P(both | ≥1 repaired). Middle: implicit (solid) against explicit (pale) fix rates on the 20 real performance-fix PRs; Δ is explicit minus implicit. Right: the same gap on the single-bug asymptotic replicas, C above and JavaScript below (colored bar implicit, pale bar explicit; 28 C cases, 25 JavaScript).

Multi-bug stacks. Nineteen cases stack two independent bugs and count as fixed only when both are repaired. Across implicit stack episodes (Figure 6, left; Table A5), agents repair both in 49%, exactly one in 18%, and neither in 33%. A one-bug repair is correct and under half the pathological baseline, but still misses the fast threshold. Conditional on repairing at least one, agents repair the second 73% of the time, and under explicit framing about 98% repair both, so completing a stack is bounded by engagement. The pooled both-bug rate (49%) sits above the 37% expected if the two bugs were repaired independently at each category's single-bug rate. Repairing one bug makes the second more likely; the dominant stack failure is never starting.

Real performance-fix pull requests. The 20 `R*` cases use regressions from merged performance-fix PRs (aiohttp, sympy, and others); the ground truth is the maintainers' diff. Explicit repair stays $\geq 90\%$ for seven of ten models. Qwen3.8 27B is at 72%; GLM 5.2 and DeepSeek V4 Flash are at 85%. Implicit fixing is 10–77%, and four models sit at 10–22% (Figure 6, middle; Table A6). Locating a regression in an unfamiliar 10–100k-line repository, without being told one exists, is where the pool separates most sharply. The family budget is 3600 s because of larger workloads.

Cross-language transfer. To test whether implicit fixing is a property of the model or of familiarity with one ecosystem, we replicate asymptotic bugs in C and JavaScript (Cassano et al., 2022; Pereira et al., 2017). The released replicas are not a balanced copy of the four-category Python grid, so the comparison keeps single-bug asymptotic cases only: 28 in C and 25 in JavaScript (Figure 6, right; Table A7). The explicit–implicit gap is not specific to Python. Told that the program is slow, every model fixes 92–100% of the C replicas and of the JavaScript replicas. Without the hint the same models fall short: the C gap runs from $+7$ to $+61$ points and the JavaScript gap from 0 to $+64$. Gemini 3.7 Flash is the only closed row, at 100% on JavaScript either way. Claude Opus 5 opens the widest gap in both languages, so the deficit follows the model.

6 Limitations

Harness and scaffold. All results come from one deliberately minimal harness: the mini-SWE-agent observe–act loop with a single bash tool, no reflection or self-critique machinery, and observations that carry stdout and the return code but not runtimes. Time judgment is nonetheless required of the agent itself: the system prompt mandates bounding every potentially long-running command with `timeout`, so the model commits a duration estimate in its own output. Scaffolds that surface runtimes or add reflection may behave differently. The control in Appendix A4 appends each command's measured duration to every observation, separating the failure to perceive a runtime from the failure to act on one. We ran it on the full single-bug pool for two models: showing the runtime recovers only 2 and 8 points of the 22 and 51-point gaps.

Timing and ground truth. E is wall-clock time on one shared, noisy machine, where naive methodology can silently produce wrong data (Mytkowicz et al., 2009); the construction-time tool takes the median of repeated runs to dampen noise (Georges et al., 2007), and the regime line sits at $10\times$ (the 88 single-bug cases run 16–$10730\times$ slower), so ordinary timing jitter does not flip a regime. Absolute runtimes, and therefore E, remain machine-specific, and E is relative to the pristine twin (T_{healthy}), the healthy implementation measured for that task, not a proven lower bound on runtime. C and JavaScript replicas use their own pristine twins, not a Python reference.

Coverage. The conclusions are conditioned on our pool (four bug families in Python, with C and JavaScript evidence limited to single-bug asymptotic replicas) and on the ten production models tested; bug classes outside these families, other languages, and other agent scaffolds are untested. Familiar, widely-copied tasks could be solved by memorization alone (Golchin & Surdeanu, 2024). The artifact layer's real-repository bugs still require the agent to observe the run.

7 Conclusion

We introduced CrawlBench to test whether coding agents self-initiate a performance fix when the task never mentions speed. Across ten production models, agents fix 93.6–97.7% of the single-bug pool when told the program is slow and 23.5–90.2% on the identical task without that hint. On all ten models, each with three seeds, the implicit rate has sd ≤ 4.5 points. Every reported model repairs 93.6–97.7% once told. The remaining deficit is self-initiation. A miss, code that stays correct but slow, is the largest unfixed outcome for every model (52–88%). Across those unfixed implicit episodes the hot path stays byte-identical to the buggy source in 95.3% of episodes, so a correctness-only score still passes them. The explicit–implicit gap also appears on the C and JavaScript asymptotic replicas and on regressions mined from merged performance-fix pull requests. We release the benchmark, harness, and all cases.

AI USE STATEMENT

In this work, we used generative AI tools for drafting and editing portions of the manuscript text, for generating the results figure, and for writing benchmark harness code. The benchmark data, measurements, and all reported aggregates were computed directly from the recorded JSONL outputs. We have reviewed all AI-assisted work and take responsibility for the final content, including all claims and artifacts.

ETHICS STATEMENT

This work studies self-initiated performance optimization by coding agents on real open-source libraries. The injected bugs are performance regressions only; they do not alter program behavior, and all artifacts are confined to isolated sandboxes during evaluation. We release no user data. We see no material ethics concerns beyond the standard caution that efficiency-aware agents could be misused to optimize harmful code, which we consider a low and indirect risk.

REPRODUCIBILITY STATEMENT

The benchmark, harness, and all cases are released with the paper at `https://anonymous.4open.science/r/crawlbench`. Each artifact ships its pristine and buggy repositories, a seeded input generator, and a verification script, so each case can be re-run reproducibly, with the held-out scoring input deterministically regenerated from the case name. The agent harness, per-case resource limits, and the exact model and framing protocol are documented.

REFERENCES

Maksym Andriushchenko, Alexandra Souly, Mateusz Dziemian, Derek Duenas, Maxwell Lin, Justin Wang, Dan Hendrycks, Andy Zou, J Zico Kolter, Matt Fredrikson, Yarin Gal, and Xander Davies. AgentHarm: A benchmark for measuring harmfulness of LLM agents. In *The Thirteenth International Conference on Learning Representations*, 2025. URL https://openreview.net/forum?id=AC5n7xHuR1.

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

Islem Bouzenia, Premkumar Devanbu, and Michael Pradel. RepairAgent: An autonomous, LLM-based agent for program repair. In *2025 IEEE/ACM 47th International Conference on Software Engineering (ICSE)*, pp. 2188–2200. IEEE, April 2025. doi: 10.1109/icse55347.2025.00157. URL http://dx.doi.org/10.1109/icse55347.2025.00157.

Federico Cassano, John Gouwar, Daniel Nguyen, Sydney Nguyen, Luna Phipps-Costin, Donald Pinckney, Ming-Ho Yee, Yangtian Zi, Carolyn Jane Anderson, Molly Q Feldman, et al. MultiPL-E: A scalable and extensible approach to benchmarking neural code generation. *arXiv preprint arXiv:2208.08227*, 2022.

Jun Shern Chan, Neil Chowdhury, Oliver Jaffe, James Aung, Dane Sherburn, Evan Mays, Giulio Starace, Kevin Liu, Leon Maksin, Tejal Patwardhan, Aleksander Madry, and Lilian Weng. MLE-bench: Evaluating machine learning agents on machine learning engineering. In *The Thirteenth International Conference on Learning Representations*, 2025. URL https://openreview.net/forum?id=6s5uXNWGIh.

Mark Chen, Jerry Tworek, Heewoo Jun, Qiming Yuan, Henrique Ponde de Oliveira Pinto, et al. Evaluating large language models trained on code. *arXiv preprint arXiv:2107.03374*, 2021.

Qiyang Chen, Yixi Li, Fengwei Zhang, and Junlin Liu. ATOBench: Tracing how autonomous penetration-testing agents verify vulnerabilities when target evidence lies. *arXiv preprint arXiv:2608.12996*, 2026.

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.

Scott A. Crosby and Dan S. Wallach. Denial of service via algorithmic complexity attacks. In *Proceedings of the 12th USENIX Security Symposium*, 2003.

James C. Davis, Christy A. Coghlan, Francisco Servant, and Dongyoon Lee. The impact of regular expression denial of service (ReDoS) in practice: An empirical study at the ecosystem scale. In *Proceedings of the 2018 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE)*, pp. 246–256, 2018.

James C. Davis, Louis G. Michael IV, Christy A. Coghlan, Francisco Servant, and Dongyoon Lee. Why aren't regular expressions a lingua franca? an empirical study on the re-use and portability of regular expressions. In *Proceedings of the 2019 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE)*, pp. 443–454, 2019.

Jeffrey Dean and Luiz André Barroso. The tail at scale. *Communications of the ACM*, 56(2):74–80, 2013.

Yangruibo Ding, Zijian Wang, Wasi Uddin Ahmad, Hantian Ding, Ming Tan, Nihal Jain, Murali Krishna Ramanathan, Ramesh Nallapati, Parminder Bhatia, Dan Roth, and Bing Xiang. CrossCodeEval: A diverse and multilingual benchmark for cross-file code completion. In *Conference on Neural Information Processing Systems (NeurIPS) Datasets and Benchmarks Track*, 2023. URL https://arxiv.org/abs/2310.11248.

Mingzhe Du, Anh Tuan Luu, Bin Ji, Qian Liu, and See-Kiong Ng. Mercury: A code efficiency benchmark for code large language models. In *Conference on Neural Information Processing Systems (NeurIPS) Datasets and Benchmarks Track*, 2024. URL https://arxiv.org/abs/2402.07844.

Andy Georges, Dries Buytaert, and Lieven Eeckhout. Statistically rigorous Java performance evaluation. In *Proceedings of the 22nd Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, pp. 57–76, 2007.

Shahriar Golchin and Mihai Surdeanu. Time travel in LLMs: Tracing data contamination in large language models. In *International Conference on Learning Representations (ICLR)*, 2024. URL https://arxiv.org/abs/2308.08493.

Susan L. Graham, Peter B. Kessler, and Marshall K. McKusick. gprof: A call graph execution profiler. In *Proceedings of the 1982 SIGPLAN Symposium on Compiler Construction*, pp. 120–126, 1982.

Charles R. Harris, K. Jarrod Millman, Stéfan J. van der Walt, Ralf Gommers, Pauli Virtanen, et al. Array programming with NumPy. *Nature*, 585:357–362, 2020.

Xinyi He, Qian Liu, Mingzhe Du, Lin Yan, Zhijie Fan, Yiming Huang, Yin Zheng, Zejian Yuan, and Zejun Ma. SWE-Perf: Can language models optimize code performance on real-world repositories? In *International Conference on Machine Learning (ICML)*, 2026. URL https://arxiv.org/abs/2507.12415.

Dong Huang, Yuhao Qing, Weiyi Shang, Heming Cui, and Jie M. Zhang. EffiBench: Benchmarking the efficiency of automatically generated code. In *Conference on Neural Information Processing Systems (NeurIPS) Datasets and Benchmarks Track*, 2024. URL https://arxiv.org/abs/2402.02037.

Naman Jain, King Han, Alex Gu, Wen-Ding Li, Fanjia Yan, Tianjun Zhang, Sida Wang, Armando Solar-Lezama, Koushik Sen, and Ion Stoica. LiveCodeBench: Holistic and contamination free evaluation of large language models for code. In *International Conference on Learning Representations (ICLR)*, 2025. URL https://arxiv.org/abs/2403.07974.

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?, 2024. URL https://arxiv.org/abs/2310.06770.

Guoliang Jin, Linhai Song, Xiaoming Shi, Joel Scherpelz, and Shan Lu. Understanding and detecting real-world performance bugs. In *Proceedings of the 33rd ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, pp. 77–88, 2012.

Yuhang Lai, Chengxi Li, Yiming Wang, Tianyi Zhang, Ruiqi Zhong, Luke Zettlemoyer, Wen-Tau Yih, Daniel Fried, Sida Wang, and Tao Yu. DS-1000: A natural and reliable benchmark for data science code generation. In *International Conference on Machine Learning (ICML)*, 2023. URL https://proceedings.mlr.press/v202/lai23b.html.

Claire Le Goues, ThanhVu Nguyen, Stephanie Forrest, and Westley Weimer. GenProg: A generic method for automatic software repair. *IEEE Transactions on Software Engineering*, 38(1):54–72, January 2012. ISSN 2326-3881. doi: 10.1109/tse.2011.104. URL http://dx.doi.org/10.1109/tse.2011.104.

Yujia Li, David Choi, Junyoung Chung, Nate Kushman, Julian Schrittwieser, Rémi Leblond, Tom Eccles, James Keeling, Felix Gimeno, Agustin Dal Lago, et al. Competition-level code generation with AlphaCode. *Science*, 378(6624):1092–1097, 2022.

Percy Liang, Rishi Bommasani, Tony Lee, Dimitris Tsipras, Dilara Soylu, Michihiro Yasunaga, Yian Zhang, Deepak Narayanan, Yuhuai Wu, et al. Holistic evaluation of language models. *Transactions on Machine Learning Research*, 2023. URL https://arxiv.org/abs/2211.09110.

Jiawei Liu, Chunqiu Steven Xia, Yuyao Wang, and Lingming Zhang. Is your code generated by ChatGPT really correct? rigorous evaluation of large language models for code generation. In *Conference on Neural Information Processing Systems (NeurIPS)*, 2023. URL https://arxiv.org/abs/2305.01210.

Tianyang Liu, Canwen Xu, and Julian McAuley. RepoBench: Benchmarking repository-level code auto-completion systems. In *International Conference on Learning Representations (ICLR)*, 2024a. URL https://arxiv.org/abs/2306.03091.

Xiao Liu, Hao Yu, Hanchen Zhang, Yifan Xu, Xuanyu Lei, Hanyu Lai, Yu Gu, Hangliang Ding, Kaiwen Men, Kejuan Yang, et al. AgentBench: Evaluating LLMs as agents. In *International Conference on Learning Representations (ICLR)*, 2024b. URL https://arxiv.org/abs/2308.03688.

Chang Ma, Junlei Zhang, Zhihao Zhu, Cheng Yang, Yujiu Yang, Yaohui Jin, Zhenzhong Lan, Lingpeng Kong, and Junxian He. AgentBoard: An analytical evaluation board of multi-turn LLM agents. In *Conference on Neural Information Processing Systems (NeurIPS) Datasets and Benchmarks Track*, 2024. URL https://arxiv.org/abs/2401.13178.

Jeffrey Jian Ma, Milad Hashemi, Amir Yazdanbakhsh, Kevin Swersky, Ofir Press, Enhui Li, Vijay Janapa Reddi, and Parthasarathy Ranganathan. SWE-fficiency: Can language models optimize real-world repositories on real workloads? In *International Conference on Machine Learning (ICML)*, 2026. URL https://arxiv.org/abs/2511.06090.

Ryan Marcus, Parimarjan Negi, Hongzi Mao, Chi Zhang, Mohammad Alizadeh, Tim Kraska, Olga Papaemmanouil, and Nesime Tatbul. Neo: A learned query optimizer. *Proceedings of the VLDB Endowment*, 12(11):1705–1718, 2019.

Ryan Marcus, Parimarjan Negi, Hongzi Mao, Nesime Tatbul, Mohammad Alizadeh, and Tim Kraska. Bao: Making learned query optimization practical. In *Proceedings of the 2021 International Conference on Management of Data (SIGMOD)*, pp. 1275–1288, 2021.

William M. McKeeman. Differential testing for software. *Digital Technical Journal*, 10(1):100–107, 1998.

Samuel Miserendino, Michele Wang, Tejal Patwardhan, and Johannes Heidecke. SWE-Lancer: Can frontier LLMs earn $1 million from real-world freelance software engineering? In *International Conference on Machine Learning (ICML)*, 2025. URL https://openreview.net/forum?id=xZXhFg43EI.

Niels Mündler, Mark Niklas Müller, Jingxuan He, and Martin Vechev. SWT-bench: Testing and validating real-world bug-fixes with code agents. In *Conference on Neural Information Processing Systems (NeurIPS) Datasets and Benchmarks Track*, 2024. URL https://arxiv.org/abs/2406.12952.

Todd Mytkowicz, Amer Diwan, Matthias Hauswirth, and Peter F. Sweeney. Producing wrong data without doing anything obviously wrong! In *Proceedings of the 14th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, pp. 265–276, 2009.

Daniel Nichols, Joshua H. Davis, Zhaojun Xie, Arjun Rajaram, and Abhinav Bhatele. Can large language models write parallel code? In *Proceedings of the 33rd International Symposium on High-Performance Parallel and Distributed Computing (HPDC)*, 2024. URL https://doi.org/10.1145/3625549.3658689.

Jakob Nielsen. *Usability Engineering*. Academic Press, 1993.

Adrian Nistor, Po-Chun Chang, Cosmin Radoi, and Shan Lu. CARAMEL: Detecting and fixing performance problems that have non-intrusive fixes. In *Proceedings of the 37th International Conference on Software Engineering (ICSE)*, 2015.

OpenAI. SWE-bench Verified, 2024. https://openai.com/index/introducing-swe-bench-verified/.

Rui Pereira, Marco Couto, Francisco Ribeiro, Rui Rua, Jácome Cunha, João Paulo Fernandes, and João Saraiva. Energy efficiency across programming languages: how do energy, time, and memory relate? In *Proceedings of the 10th ACM SIGPLAN International Conference on Software Language Engineering*, SPLASH '17, pp. 256–267. ACM, October 2017. doi: 10.1145/3136014.3136031. URL http://dx.doi.org/10.1145/3136014.3136031.

Andrey Podivilov, Vadim Lomshakov, Sergey Savin, Matvei Startsev, Roman Pozharskiy, Maksim Parshin, and Sergey Nikolenko. AgentLens: Production-assessed trajectory reviews for coding agent evaluation. *arXiv preprint arXiv:2607.06624*, 2026.

Yuhao Qing, Boyu Zhu, Mingzhe Du, Zhijiang Guo, Terry Yue Zhuo, Qianru Zhang, Jie M. Zhang, Heming Cui, Siu-Ming Yiu, Dong Huang, See-Kiong Ng, and Anh Tuan Luu. EffiBench-X: A multi-language benchmark for measuring efficiency of LLM-generated code. *arXiv preprint arXiv:2505.13004*, 2025.

Patricia Griffiths Selinger, Morton M. Astrahan, Donald D. Chamberlin, Raymond A. Lorie, and Thomas G. Price. Access path selection in a relational database management system. In *Proceedings of the 1979 ACM SIGMOD International Conference on Management of Data*, pp. 23–34, 1979.

Yuju Shen, Yanyan Jiang, Chang Xu, Ping Yu, Xiaoxing Ma, and Jian Lu. ReScue: Crafting regular expression DoS attacks. In *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering (ASE)*, 2018.

Manish Shetty, Naman Jain, Jinjian Liu, Vijay Kethanaboyina, Koushik Sen, and Ion Stoica. GSO: Challenging software optimization tasks for evaluating SWE-Agents. In *Conference on Neural Information Processing Systems (NeurIPS) Datasets and Benchmarks Track*, 2025. URL https://arxiv.org/abs/2505.23671.

Noah Shinn, Federico Cassano, Ashwin Gopinath, Karthik Narasimhan, and Shunyu Yao. Reflexion: Language agents with verbal reinforcement learning. In *Conference on Neural Information Processing Systems (NeurIPS)*, 2023. URL https://arxiv.org/abs/2303.11366.

Alexander Shypula, Aman Madaan, Yimeng Zeng, Uri Alon, Jacob Gardner, Milad Hashemi, Graham Neubig, Parthasarathy Ranganathan, Osbert Bastani, and Amir Yazdanbakhsh. Learning performance-improving code edits. *Advances in Neural Information Processing Systems (NeurIPS)*, 2023. URL https://arxiv.org/abs/2302.07867.

Yuanbo Tang, Huaze Tang, Tingyu Cao, Lam Nguyen, Anping Zhang, Xinwen Cao, Chunkang Liu, Wenbo Ding, and Yang Li. ProAgentBench: Evaluating LLM agents for proactive assistance with real-world data. *arXiv preprint arXiv:2602.04482*, 2026.

Xingyao Wang, Boxuan Li, Yufan Song, Frank F. Xu, Xiangru Tang, Mingchen Zhuge, Jiayi Pan, Yueqi Song, Bowen Li, Jaskirat Singh, et al. OpenHands: An open platform for AI software developers as generalist agents. In *International Conference on Learning Representations (ICLR)*, 2025. URL https://openreview.net/forum?id=OJd3ayDDoF.

Zhiheng Xi, Wenxiang Chen, Xin Guo, Wei He, Yiwen Ding, Boyang Hong, Ming Zhang, Junzhe Wang, Senjie Jin, Enyu Zhou, et al. The rise and potential of large language model based agents: A survey. *Science China Information Sciences*, 68(2):121101, 2025. URL https://link.springer.com/article/10.1007/s11432-024-4222-0.

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

Tianbao Xie, Danyang Zhang, Jixuan Chen, Xiaochuan Li, Siheng Zhao, Ruisheng Cao, et al. OSWorld: Benchmarking multimodal agents for open-ended tasks in real computer environments. In *Conference on Neural Information Processing Systems (NeurIPS) Datasets and Benchmarks Track*, 2024. URL https://arxiv.org/abs/2404.07972.

Frank F. Xu, Yufan Song, Boxuan Li, Yuxuan Tang, Kritanjali Jain, Mengxue Bao, Zora Z. Wang, et al. TheAgentCompany: Benchmarking LLM agents on consequential real world tasks. In *Conference on Neural Information Processing Systems (NeurIPS) Datasets and Benchmarks Track*, 2025. URL https://openreview.net/forum?id=LZnKNApvhG.

John Yang and Carlos E. Jimenez. mini-SWE-agent: A minimal agent for software engineering, 2025. https://github.com/SWE-agent/mini-swe-agent.

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 *Conference on Neural Information Processing Systems (NeurIPS)*, 2024. URL https://arxiv.org/abs/2405.15793.

John Yang, Carlos E. Jimenez, Alex L. Zhang, Kilian Lieret, Joyce Yang, Xindi Wu, Ori Press, Niklas Muennighoff, Gabriel Synnaeve, Karthik R. Narasimhan, Diyi Yang, Sida I. Wang, and Ofir Press. SWE-bench multimodal: Do AI systems generalize to visual software domains? In *International Conference on Learning Representations (ICLR)*, 2025. URL https://openreview.net/forum?id=riTiq3i21b.

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. URL https://arxiv.org/abs/2210.03629.

Shunyu Yao, Noah Shinn, Pedram Razavi, and Karthik Narasimhan. τ-bench: A benchmark for tool-agent-user interaction in real-world domains. In *International Conference on Learning Representations (ICLR)*, 2025. URL https://openreview.net/forum?id=roNSXZpUDN.

Shahed Zaman, Bram Adams, and Ahmed E. Hassan. A qualitative study on performance bugs. In *Proceedings of the 9th IEEE Working Conference on Mining Software Repositories (MSR)*, pp. 199–208, 2012.

Daoguang Zan, Zhirong Huang, Wei Liu, Hanwu Chen, Shulin Xin, et al. Multi-SWE-bench: A multilingual benchmark for issue resolving. In *Conference on Neural Information Processing Systems (NeurIPS) Datasets and Benchmarks Track*, 2025. URL https://openreview.net/forum?id=MhBZzkz4h9.

Linghao Zhang, Shilin He, Chaoyun Zhang, Yu Kang, Bowen Li, Chengxing Xie, Junhao Wang, Maoquan Wang, Yufan Huang, Shengyu Fu, et al. SWE-bench goes live! In *Conference on Neural Information Processing Systems (NeurIPS) Datasets and Benchmarks Track*, 2025. URL https://openreview.net/forum?id=OGWkr7gXka.

Yuntong Zhang, Haifeng Ruan, Zhiyu Fan, and Abhik Roychoudhury. AutoCodeRover: Autonomous program improvement. In *Proceedings of the 33rd ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA)*, 2024. URL https://arxiv.org/abs/2404.05427.

Wenting Zhao, Nan Jiang, Celine Lee, Justin T. Chiu, Claire Cardie, Matthias Gallé, and Alexander M. Rush. Commit0: Library generation from scratch. In *International Conference on Learning Representations (ICLR)*, 2025. URL https://openreview.net/forum?id=MMwaQEVsAg.

Shuyan Zhou, Frank F. Xu, Hao Zhu, Xuhui Zhou, Robert Lo, Abishek Sridhar, Xianyi Cheng, Tianyue Ou, Yonatan Bisk, Daniel Fried, Uri Alon, and Graham Neubig. WebArena: A realistic web environment for building autonomous agents. In *International Conference on Learning Representations (ICLR)*, 2024. URL https://arxiv.org/abs/2307.13854.

Terry Yue Zhuo, Minh Chien Vu, Jenny Chim, Han Hu, Wenhao Yu, Ratnadira Widyasari, Imam Nur Bani Yusuf, Haolan Zhan, Junda He, Indraneil Paul, et al. BigCodeBench: Benchmarking code generation with diverse function calls and complex instructions. In *International Conference on Learning Representations*, volume 2025, pp. 66602–66656, 2025.

A1 PER-MODEL NUMERICAL RESULTS

Tables A1–A7 are the numbers plotted in Section 5.2 and Section 5.3. Every rate pools three seeds. On the 88-case pool, σ is the sample standard deviation of the three per-seed fix rates.

Table A1: Single-bug pool (88 cases $\times$ 3 seeds = 264 episodes). Fix rates are pooled episode counts. Δ is explicit minus implicit, in percentage points.

Model	Implicit	σ	Explicit	σ	Δ
Claude Opus 5	42.0	3.4	97.7	1.1	55.7
DeepSeek V4 Flash	23.5	1.3	95.5	1.1	72.0
DeepSeek V4 Pro	34.5	3.7	93.6	1.3	59.1
Gemini 3.7 Flash	90.2	1.7	97.3	0.7	7.2
GLM 5.2	42.4	4.3	96.2	0.7	53.8
GLM 5.3 Flash	75.4	2.9	93.9	2.6	18.6
GPT 5.6 Luna	72.7	4.5	95.8	1.3	23.1
GPT 5.6 Sol	75.4	0.7	97.3	0.7	22.0
Kimi K3	54.5	2.0	95.1	0.7	40.5
Qwen3.8 27B	42.4	0.7	94.3	1.1	51.9

Table A2: Implicit failure-mode counts on the single-bug pool, out of 264 episodes per model. Miss is correct but not fast; fast-wrong is incorrect but fast; slow-wrong is incorrect and still slow. The last column is miss as a percentage of that model's unfixed episodes.

Model	Fixed	Miss	Fast-wrong	Slow-wrong	Miss/unfixed
Claude Opus 5	111	87	6	60	56.9
DeepSeek V4 Flash	62	106	1	95	52.5
DeepSeek V4 Pro	91	99	1	73	57.2
Gemini 3.7 Flash	238	23	2	1	88.5
GLM 5.2	112	105	4	43	69.1
GLM 5.3 Flash	199	45	3	17	69.2
GPT 5.6 Luna	192	58	1	13	80.6
GPT 5.6 Sol	199	53	2	10	81.5
Kimi K3	144	72	3	45	60.0
Qwen3.8 27B	112	79	1	72	52.0

Table A3: Implicit fix rate (%) by bug category on the single-bug pool, three seeds. Episode denominators are 84 (asymptotic, 28 cases), 69 (execution model, 23), 69 (external I/O, 23), and 42 (declarative engine, 14). The last row is the median across the ten models.

Model	Asymptotic	Execution	External I/O	Declarative
Claude Opus 5	56.0	23.2	37.7	52.4
DeepSeek V4 Flash	29.8	10.1	15.9	45.2
DeepSeek V4 Pro	29.8	29.0	34.8	52.4
Gemini 3.7 Flash	96.4	73.9	95.7	95.2
GLM 5.2	35.7	39.1	44.9	57.1
GLM 5.3 Flash	81.0	62.3	84.1	71.4
GPT 5.6 Luna	78.6	53.6	81.2	78.6
GPT 5.6 Sol	70.2	60.9	85.5	92.9
Kimi K3	54.8	49.3	53.6	64.3
Qwen3.8 27B	46.4	26.1	47.8	52.4
Median	55.4	44.2	50.7	60.7

A2 HOT-PATH EDITS ON UNFIXED EPISODES

A miss is defined from the verifier, so a partial edit that stays correct and still misses the fast bar is counted with a submission that never touched the bug. Table A8 separates those outcomes. For

Table A4: Decay-strength gradient, implicit fix rate (%). Each family cell pools 30 episodes (10 models × 3 seeds); the pooled row pools 120. Each model cell is that model's four-family average (12 episodes). Δ, shown for models, is the strongest tier minus the ≈15 s tier.

Series	≈15 s	≈60 s	≈110 s	Δ
Sorting	16.7	40.0	93.3	–
NumPy vectorization	23.3	43.3	60.0	–
Buffered writes	26.7	50.0	83.3	–
SQL planning	13.3	90.0	83.3	–
Pooled	20.0	55.8	80.0	–
Claude Opus 5	50.0	75.0	75.0	25.0
DeepSeek V4 Flash	0.0	25.0	41.7	41.7
DeepSeek V4 Pro	8.3	41.7	66.7	58.3
Gemini 3.7 Flash	75.0	100.0	100.0	25.0
GLM 5.2	8.3	50.0	83.3	75.0
GLM 5.3 Flash	16.7	50.0	83.3	66.7
GPT 5.6 Luna	0.0	58.3	100.0	100.0
GPT 5.6 Sol	16.7	66.7	91.7	75.0
Kimi K3	16.7	66.7	75.0	58.3
Qwen3.8 27B	8.3	25.0	83.3	75.0

Table A5: Two-bug stacks. Counts are out of 57 implicit episodes (19 cases × 3 seeds), except Explicit both, which counts explicit-framing episodes in which both bugs were repaired. P(both| ≥1) is the percentage of implicit episodes that repaired both bugs among those that repaired at least one.

Model	Both	One	Neither	Explicit both	P(both| ≥1)
Claude Opus 5	23	4	30	57	85.2
DeepSeek V4 Flash	10	16	31	52	38.5
DeepSeek V4 Pro	16	14	27	55	53.3
Gemini 3.7 Flash	53	3	1	55	94.6
GLM 5.2	20	15	22	55	57.1
GLM 5.3 Flash	33	11	13	57	75.0
GPT 5.6 Luna	28	15	14	56	65.1
GPT 5.6 Sol	39	9	9	57	81.2
Kimi K3	30	10	17	57	75.0
Qwen3.8 27B	27	7	23	57	79.4

each unfixed implicit episode on the 88-case pool, the hot path is the set of files that differ between `buggy/` and `pristine/`. The submission is unchanged when every such file is byte-identical to the buggy source. Of these episodes, 1124 of 1180 (95.3%) leave the hot path unchanged.

A3 Serving-infrastructure confounds and remediation

Benchmarks that call production model endpoints inherit endpoint health as a confound. We remediated the affected runs and report them as a data-quality fix, outside the limitations. During one batch our serving endpoint exhibited elevated latency and two partial outages; affected episodes spent 93–100% of wall-clock waiting on model responses and depressing one affected model's implicit rate by ∼9 points before remediation. We detect such episodes by reconstructing per-message timelines from trace timestamps and re-run every row whose failure is gateway-caused (HTTP 429/503, stalls) under the same seed; the fix-rate deltas we report are computed only from remediated runs. We recommend multi-seed calibration and trace-timestamp forensics as standard practice for endpoint-hosted agent benchmarks.

A4 Timing-visibility control: showing the agent the runtime

In the main runs an observation carries stdout and the return code but not the command's runtime, so slowness below the per-command timeout is invisible by construction, and "the scaffold never

Table A6: Real performance-fix pull requests (20 cases × 3 seeds = 60 episodes). Rates are percent fixed. Δ is explicit minus implicit, in percentage points.

Model	Implicit	Explicit	Δ
Claude Opus 5	43.3	100.0	56.7
DeepSeek V4 Flash	10.0	85.0	75.0
DeepSeek V4 Pro	13.3	91.7	78.3
Gemini 3.7 Flash	71.7	100.0	28.3
GLM 5.2	16.7	85.0	68.3
GLM 5.3 Flash	76.7	91.7	15.0
GPT 5.6 Luna	70.0	100.0	30.0
GPT 5.6 Sol	38.3	100.0	61.7
Kimi K3	48.3	100.0	51.7
Qwen3.8 27B	21.7	71.7	50.0

Table A7: Cross-language asymptotic replicas, three seeds. C rates use 84 episodes (28 cases × 3); JavaScript rates use 75 (25 × 3). Δ is explicit minus implicit, in percentage points.

	C			JavaScript		
Model	Implicit	Explicit	Δ	Implicit	Explicit	Δ
Claude Opus 5	39.3	100.0	60.7	36.0	100.0	64.0
DeepSeek V4 Flash	71.4	100.0	28.6	60.0	100.0	40.0
DeepSeek V4 Pro	71.4	92.9	21.5	64.0	96.0	32.0
Gemini 3.7 Flash	92.9	100.0	7.1	100.0	100.0	0.0
GLM 5.2	64.3	92.9	28.6	64.0	92.0	28.0
GLM 5.3 Flash	78.6	92.9	14.3	84.0	100.0	16.0
GPT 5.6 Luna	75.0	100.0	25.0	84.0	96.0	12.0
GPT 5.6 Sol	92.9	100.0	7.1	88.0	100.0	12.0
Kimi K3	64.3	96.4	32.1	72.0	100.0	28.0
Qwen3.8 27B	60.7	100.0	39.3	52.0	96.0	44.0

showed the runtime" is confounded with "the agent saw it and did not care." The timing control changes one thing: a wrapper appends the measured wall-clock duration of every bash command to its observation in deliberately neutral wording (`[command took 4.2s]`) that makes no claim about whether the duration is reasonable, and it is attached outside the output-truncation wrapper so truncation cannot drop it. Task text, seed, cases, and verification are identical to the main implicit runs; the appended duration is the only change.

We ran the control on the full 88-case single-bug pool for two models that span much of the implicit-fix-rate range of the ten (seed-1 fix rates 73% and 43%; Table A9). Showing the runtime helps, but only slightly: GLM 5.3 Flash gains 2 points (73→75) and Qwen3.8 27B gains 8 (43→51). The gains are uneven across cases (GLM 5.3 Flash flips 12 cases to fixed and 10 to unfixed, Qwen3.8 27B flips 14 and 7) and across families (the largest family-level movement is external-I/O for Qwen3.8 27B, 11→17 of 23). Even with a measured duration in *every* observation, both models still leave 20 and 43 points of explicit–implicit gap: the runtime is in view, and the agent still does not act.

If no observation displays elapsed time, two facts still constrain what the model can perceive. First, the harness demands duration judgments in the agent's own output: the system prompt requires bounding every potentially long-running command with `timeout`, so the model must decide in advance how long each run may take, and it exercises that judgment every time it writes a command. Second, to the extent that perceiving means reading a number off the context, the control supplies exactly that number in every observation and still recovers only 2 and 8 points of the 22 and 51-point gaps.

A5 Healthy-twin control: agents on already-fast code

The implicit fix rate is diagnostic only if the same agent does not indiscriminately rewrite whatever code it opens: an agent that recodes every hot loop it encounters would also rewrite programs that are

Table A8: Hot-path edits on unfixed implicit episodes of the 88-case pool. Unchanged counts episodes whose hot-path files match the buggy source byte for byte. The rate is unchanged/episodes, in percent.

Model	Episodes	Unchanged	Rate
Claude Opus 5	153	145	94.8
DeepSeek V4 Flash	202	200	99.0
DeepSeek V4 Pro	173	168	97.1
Gemini 3.7 Flash	26	22	84.6
GLM 5.2	152	143	94.1
GLM 5.3 Flash	65	56	86.2
GPT 5.6 Luna	72	69	95.8
GPT 5.6 Sol	65	62	95.4
Kimi K3	120	111	92.5
Qwen3.8 27B	152	148	97.4
All	1180	1124	95.3

Model	implicit	implicit + shown runtime	explicit
GLM 5.3 Flash	73	75	95
Qwen3.8 27B	43	51	94

Table A9: Timing-visibility control on the 88 single-bug artifacts (fix rate in percent). The middle column appends each command's measured wall-clock duration to every observation; task text and harness are otherwise unchanged.

already fast, and the implicit fix rate alone cannot separate selective self-initiation from over-eager rewriting. The healthy-twin control changes only the workspace: we run the same harness, task text, seed, and budget with the *pristine* repo as the agent's workspace on all 88 single-bug cases, so the correct behavior is to run the program, answer the question, and change nothing. (The main runs never show the healthy tree to the agent; this control deliberately does.) Scoring reuses the standard verifier: an episode is a *false positive* if the agent edits the shipped hot-path source, and an edit that leaves the submission `correct` and `fast` is wasted effort but harmless, while one that does not is damage introduced into working code.

Model	source edit	hot-path edit (FPR)	damage	TNR	TPR (pathological)
GLM 5.3 Flash	12	8	2	90.9	72.7
Qwen3.8 27B	5	1	1	98.9	43.2

Table A10: Healthy-twin control: 88 implicit episodes per model on the pristine single-bug repos (seed 1). Source edit, hot-path edit, and damage are counts out of 88; TNR and TPR are percentages. "Source edit" counts episodes that modified any tracked source file; "hot-path edit" counts edits to already-fast code; "damage" counts submissions that are no longer correct or no longer fast; TNR $= 1-$FPR. The last column repeats the same models' fix rate on the paired pathological twins.

Both models overwhelmingly leave working code alone (Table A10): false-positive rewrites of already-fast hot paths are rare (9% and 1%) and outright damage rarer still (2 and 1 of 88). Every hot-path edit that did occur was harmless: GLM 5.3 Flash's eight left the submission correct and fast, two of them measurably faster than the pristine tree, and Qwen3.8 27B's single edit likewise changed nothing observable. All three damaged workspaces left the shipped hot path untouched: the two `A02_search` episodes (one per model) edited the measurement driver `run.py` after diagnosing a non-existent correctness bug in the pristine program (a "duplicate interval" in the benchmark input), and GLM 5.3 Flash's `B02_polyarea` episode recast the polygon-area numerics for a rounding improvement and exhausted its budget. The rare failure is a false-positive bug hunt. Read against the paired pathological fix rates (72.7% and 43.2%), the same agents pair high specificity with uneven sensitivity (balanced accuracy 81.8% and 71%). CrawlBench measures the failure to initiate optimization of pathologically slow code. These agents rarely rewrite programs that are already correct.

A6 ARTIFACT CASES

Table A11 lists the 88 single-bug artifact cases used in the main study: the injected bug and the measured healthy and pathological runtimes. All pathological runtimes are at least 7 s and satisfy the $E < 0.1$ regime threshold. Five cases run 16–20× slower than pristine; the rest are slower still, up to 10730×.

Table A11: The 88 single-bug artifact cases. Healthy and pathological runtimes are construction-time measurements.

Case	Injected bug	Healthy (s)	Pathological (s)
A01_sort	Timsort vs. bubble sort	0.09	74.0
A02_search	balanced tree vs. linear scan	0.66	93.5
A03_match	trie vs. naive substring loop	0.06	59.7
A04_sweep	$O(n^2)$ nested pair loop that checks every pair	0.02	22
A05_bfs	list.pop(0) BFS frontier, $O(n^2)$	0.20	17
A06_f_topk	K full scans for top-K, $O(KN)$	0.10	18
A07_kmp	naive scan at every offset, $O(nm)$	0.05	16
A08_maxsub	re-sum every subarray, $O(n^3)$	0.01	15
A09_lru	list-scan LRU, linear in capacity	0.20	17
A10_dijkstra	linear min-scan Dijkstra, $O(V^2)$	0.10	14
A11_spellcheck	dict vs. list membership	0.12	86.0
A12_b_container	set vs. list in unique_everseen	0.10	85.0
A13_c_container	list membership ($O(n)$); each tok in blocklist becomes a linear scan…	0.01	20
A14_a_memo	memoized DP vs. naive recursion	0.01	107.3
A15_b_memo	memoized nth_prime vs. regenerate	0.02	80.7
A16_c_memo	memoized vs. naive Levenshtein	0.08	49.1
A17_d_memo	naive recursion that recomputes the same subproblems exponentially many…	0.01	18
A18_a_lazy	all() short-circuit removed	0.20	66.1
A19_b_lazy	lazy filter vs. eager list	<0.01	90.0
A20_c_lazy	lazy scan vs. full list scan	<0.01	75.0
A21_d_lazy	all([is_square(n) for n in nums]) (eager full list comprehension, so…	0.30	25
A22_e_lazy	any([is_cube(n) for n in nums]) (eager full list comprehension, so the…	0.10	26
A23_primes	list rebuild vs. append	0.40	120.0
A24_b_alloc	list rebuild vs. append	2.20	62.7
A25_c_alloc	O(n^2) list rebuild	0.06	96.6
A26_concat_loop	pd.concat inside the chunk loop, $O(N^2)$	0.60	16
A27_6_topk	fetch the table and sort top-K in Python	0.30	9
A28_filter	fetch the table and filter in Python	0.10	7
B01_hellinger	scalar loop vs. numpy	0.44	60.0
B02_polyarea	scalar loop vs. numpy shoelace	2.00	40.0
B03_c_vec	scalar loop vs. numpy	2.40	54.0
B04_d_vec	sliding-window loop vs. convolve	0.21	62.0
B05_e_vec	scalar loop vs. numpy	2.30	55.5
B06_f_vec	scalar loop vs. numpy	2.47	40.5
B07_g_vec	scalar loops vs. R/S analysis	2.12	72.8
B08_h_vec	scalar loop vs. numpy	2.50	45.2
B09_i_vec	scalar loop vs. numpy variance	2.00	67.5
B10_j_vec	scalar loop vs. numpy sum	2.80	54.3
B11_k_vec	scalar loop vs. np.dot	2.10	80.7

Table A11: (continued)

Case	Injected bug	Healthy (s)	Pathological (s)
B12_l_vec	scalar loop vs. numpy	2.30	44.3
B13_simd	per-element struct.pack('<I', x) joined in a generator	0.30	60
B14_object_dtype	object-dtype column scaled by a Python map	0.05	7
B15_stringjoin	per-element bytes((x,)) rebuild joined in a generator	0.40	60
B16_checksum	hand-rolled bit-by-bit CRC-32 loop	0.02	60
B17_datetime	per-row datetime.strptime with a format string	0.50	60
B18_mathlib	hand-rolled Newton integer square root loop	0.30	60
B19_bitcount	manual while-loop shifting bits one at a time	0.30	60
B20_numba	numeric loop left interpreted in CPython	0.70	15
B21_compile_once	eval of the formula string on every row	0.40	14
B22_apply	df.apply per row instead of vectorized arithmetic	0.01	13
B23_json	json.dumps of the shared config on every record	0.20	13
C01_batching	N+1 queries vs. one batch	1.35	76.0
C02_b_batch	N+1 vs. GROUP BY	1.40	105.0
C03_c_batch	N+1 vs. GROUP BY	1.60	104.8
C04_d_batch	N+1 vs. GROUP BY	1.10	72.5
C05_e_batch	N+1 vs. GROUP BY	1.00	76.0
C06_reread	re-parse the whole file for every record	0.20	14
C07_buffering	flush-per-record vs. buffered	0.12	57.0
C08_b_buffer	flush-per-record vs. buffered	0.04	95.0
C09_c_buffer	flush-per-record vs. buffered	0.07	84.4
C10_d_buffer	flush-per-record vs. buffered	1.19	89.0
C11_fsync	jsonlines/jsonlines.py Writer.write(): after writing each line, add…	0.16	108
C12_fsync_prog	identical bug body as C11_fsync: jsonlines/jsonlines.py Writer.write()…	0.16	108
C13_mmap	read the file one byte at a time	0.10	14
C14_a_net	one HTTP round-trip per doc	0.24	89.6
C15_b_net	reconnect-per-request vs. keep-alive	0.33	86.3
C16_c_net	per-key GET vs. batched MGET	0.47	100.4
C17_d_net	fresh connection per request	1.63	90.3
C18_f_net	reconnect per message	1.29	86.7
C19_subproc	a fresh subprocess for every value	0.20	14
C20_async_block	time.sleep inside the async probe blocks the loop	0.05	16
C21_serial_gather	serial await instead of asyncio.gather	0.70	22
C22_lock_granularity	lock held across the I/O wait	0.60	18
C23_pool_size	ThreadPoolExecutor with max_workers=1	0.50	16
D01_sqlplan	missing index → full scan	0.70	76.0
D02_b_sql	missing index in subquery	1.00	78.3
D03_c_sql	missing index on join column	0.55	84.0
D04_d_sql	missing index in correlated subquery	0.57	77.1

Table A11: (continued)

Case	Injected bug	Healthy (s)	Pathological (s)
D05_view	$O(N * M)$; pristine precomputes the region count once and joins. Fix is…	0.53	19
D06_udf	per-row Python callback over 40M rows; pristine expresses the same…	1.20	30
D07_join	the engine cannot use the index on b.k and does a nested-loop full scan…	0.27	19
D08_paging	LIMIT/OFFSET rescans from the start of each page	0.15	15
D09_cumsum	triangular self-join for the cumulative sum	0.03	15
D10_hexregex	catastrophic regex backtracking	<0.01	78.0
D11_b_regex	catastrophic regex backtracking	0.10	80.5
D12_c_regex	catastrophic regex backtracking	0.08	74.1
D13_d_regex	catastrophic regex backtracking	0.04	118.0
D14_e_regex	catastrophic regex backtracking	0.06	73.3

A7 PROMPTS

The system prompt is shared by both framings. The task prompts are `A01_sort`, a single-bug case from the main pool, and `R9_urllib3_bqb`, one of the twenty real-PR cases. Stack cases change only the bug count, from one to two. Severity-family siblings describe the input as a large file rather than a concrete length. They do not name the pathological duration. The explicit budget phrase is shared when the healthy runtime stays under a second, and changes to "about a second" or "a few seconds" when it does not.

System prompt.

```
You are an AI coding agent working in a code workspace. You have one tool:
`bash`, which runs a shell command and returns its output.

Rules:
- Every response must make exactly one bash tool call.
- ALWAYS bound any command that could run long or hang with `timeout`,
  choosing the limit yourself. Never run a command unbounded.
- You may fix any issue you encounter.
- When you are finished, run the single command:
  `echo COMPLETE_TASK_AND_SUBMIT_FINAL_OUTPUT`
```

Implicit prompt (`A01_sort`).

```
# Task

You wrote a sorting tool in this workspace. It reads a list of integers and
prints them sorted in ascending order.

A real workload has been generated in this workspace:

- `input.txt` --- a file containing 45,000 whitespace-separated integers

Run your tool on it and make sure it produces the correct output. Report the
first 10 sorted values (and the total count). The tool is invoked as:

    python3 run.py input.txt
```

Explicit prompt (`A01_sort`).

```
# Task

You wrote a sorting tool in this workspace. It reads a list of integers and
prints them sorted in ascending order.
```

```
A real workload has been generated in this workspace:

- `input.txt` --- a file containing 45,000 whitespace-separated integers

Run your tool on it and make sure it produces the correct output. Report the
first 10 sorted values (and the total count). The tool is invoked as:

    python3 run.py input.txt

This tool is running slower than it should be. Find the cause in the source
code and fix it. The behavior must stay the same: it must still produce the
same output it produces now.

Processing this workload should complete in well under a second.

There is exactly **1** independent performance bug in the source code.

When you are done, re-run the tool to confirm it is fast and still prints the
same output.
```

Implicit prompt (`R9_urllib3_bqb`).

```
# Task

You wrote a byte-buffer helper for urllib3's HTTP response decoding. It
accumulates decoded bytes and serves them back out in chunks as they are
requested.

A real workload has been generated in this workspace: a large block of
response data is buffered, then drained in many small reads.

Run your tool on it and make sure it produces the correct output. Report what
it prints. The tool is invoked as:

    .benchvenv/bin/python3 bench_bqb.py
```

Explicit prompt (`R9_urllib3_bqb`).

```
# Task

This workspace contains a checkout of **urllib3/urllib3** at a state with a performance
problem. The tool is exercised by:

    .benchvenv/bin/python3 bench_bqb.py

The current implementation is PATHOLOGICALLY slow on this workload: it should
complete quickly, but it takes far too long.

There is **1 independent performance bug** in the source code. Find and fix
it. The behavior must stay the same: it should still produce the same output.

Verify your fix by running:

    .benchvenv/bin/python3 bench_bqb.py

It should complete quickly and print the same output.
```