SWE REFACTOR BENCH: CAN CODING AGENTS COMPLETE A LONG-HORIZON, WHOLE-REPOSITORY STACK MIGRATION?

Anonymous authors
Paper under double-blind review

ABSTRACT

Modern software systems accumulate technical debt over decades of development, which makes migration expensive and largely manual. As coding agents become increasingly capable at bug fixing, can they autonomously perform such migrations? Existing benchmarks cannot answer this question because they evaluate only behavioural correctness, not whether the migration actually occurred. This leads to an easy hack: agents copy the original implementation to make tests pass. We call this *Blindness*. To address this problem, we introduce **SWE Refactor Bench**, a benchmark comprising 20 whole-repository migrations, covering 4 kinds of technical debt. A three-stage evaluation protocol measures both migration completeness and behavioural correctness. (1) Migration Audit verifies that the migration occurred. (2) Behavioural Tests measure correctness with a fixed test suite. (3) Agentic Verification uses 6 independent coding agents to generate targeted tests for hidden behavioural differences. Across 520 runs from 8 frontier models and 26 model-effort configurations, only 28 of 520 runs (5.4%) pass all three stages, 13 of the 20 tasks receive no accepted solution, and the best model (claude-opus-5) scores $47.0/100$. Migration completeness and behavioural correctness are distinct abilities: a few runs preserve behaviour by skipping the migration and are stopped at Migration Audit; most attempt it and break behaviour, and are stopped at Behavioural Tests. Agents cannot deliver a perfect migration: among the 340 runs that pass Migration Audit, 58% reach 99% of the fixed checks, yet only 26% reach 100%. Agent capability differs across migration categories: agents score 31.4 on build toolchain rewrites but only 5.6 on language rewrites. Together, these findings position SWE Refactor Bench as a rigorous testbed for developing coding agents for reliable whole-repository migrations.

1 INTRODUCTION

Modern software systems accumulate technical debt over decades, leaving long-lived repositories tied to technology stacks that their teams no longer want to maintain (Cunningham, 1992). Replacing a language, framework, platform, or build toolchain remains expensive and largely manual because engineers must coordinate changes across source code, dependencies, interfaces, and build logic. Examples include rewriting a C library in Rust and replacing Maven with Gradle across an entire repository. A successful migration must both replace the old stack and preserve the system's behaviour. The original system provides a behavioural reference, so correctness can be tested without a separate specification. Coding agents are increasingly reliable at localised bug fixing (Jimenez et al., 2024; Yang et al., 2024; Wang et al., 2025; Zan et al., 2025; Zhang et al., 2025). This progress raises a harder question: can coding agents autonomously complete long-horizon, whole-repository migrations while preserving behaviour? Reliable performance would extend coding agents from local edits to a costly class of system maintenance. Figure 1 contrasts the one-stage evaluation prior benchmarks apply with the three stages SWE Refactor Bench uses, and previews the benchmark's task coverage and model scores.

Existing benchmarks cannot establish whether coding agents can complete whole-repository migrations. They evaluate repository-level code changes with fixed tests, including bug fixes, language-version upgrades, and language migrations (Jimenez et al., 2024; Zan et al., 2025; Zhang et al., 2025;

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.

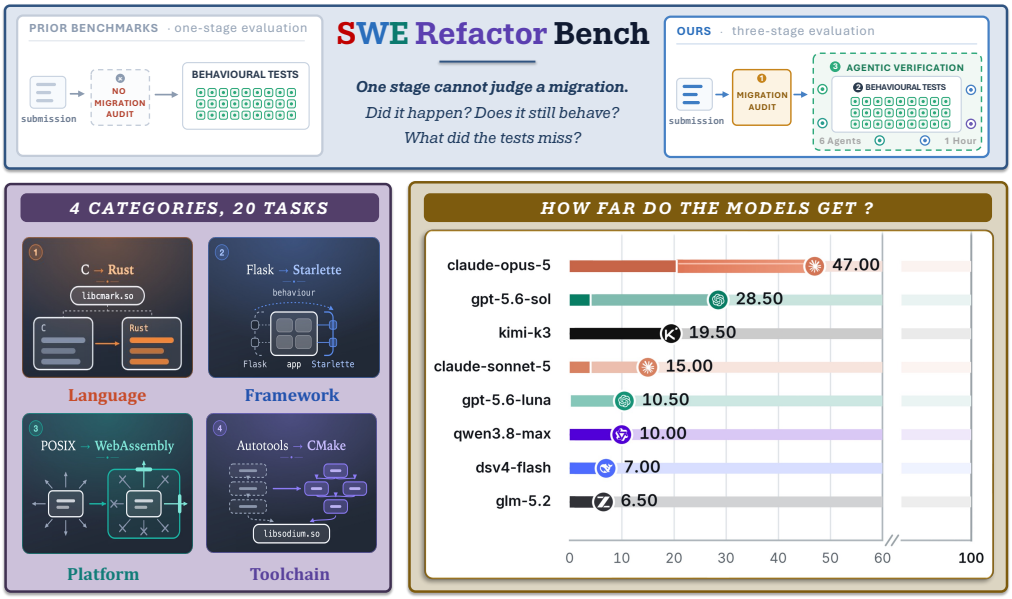

Figure 1: **SWE Refactor Bench at a glance. Top:** how a migration gets scored. **Left,** prior benchmarks: one stage, a behavioural test suite. With no migration audit in front of it, a submission that never migrated anything still turns every check green, so the suite cannot tell a completed migration from an untouched repository. **Right,** SWE Refactor Bench: three stages in series—Migration Audit asks whether the migration happened at all, Behavioural Tests is that same fixed suite, and Agentic Verification sends six coding agents, one hour each, after the differences the suite was never written to catch. **Bottom left:** the 20 tasks span four kinds of technical debt—language, framework, platform, and build toolchain. **Bottom right:** composite scores of 8 frontier models, each in its strongest configuration (out of 100; defined in (4)).

Liu et al., 2025; Wang et al., 2024b; Khatry et al., 2025). These tests provide a clear red-to-green signal for bug fixing: at least one test fails before the patch and passes after it. Whole-repository migration, however, starts from a repository whose tests already pass. A correct migration and an untouched repository can therefore receive the same perfect test score. Fixed tests measure behavioural correctness, but they cannot establish whether the migration actually occurred. This creates an easy shortcut: agents can retain or copy the original implementation to make the tests pass. We call this failure mode *Blindness*: a behaviour-only evaluator awards full credit without establishing that the target stack replaced the original implementation. Such an evaluator can detect whether known behaviour was broken, but not whether the migration occurred. Enlarging the suite does not close the gap—the untouched implementation passes any additional check by construction—so evaluating a migration requires separate evidence for two properties: that behaviour survived, and that the target stack replaced the original implementation. Section 2.1 makes this argument precise.

We introduce **SWE Refactor Bench**, which collects three forms of evidence through a three-stage protocol. Migration Audit is a hard gate that checks whether the target stack replaced the old implementation; any failed criterion vetoes the submission. This gate prevents unchanged code and wrapper patches from earning credit. The Behavioural Tests stage measures behavioural correctness with 130,118 fixed checks recorded from the original system and requires every check to pass. Agentic Verification tests behaviours that a fixed suite may miss: six independent coding agents receive the original and migrated codebases and one hour each to generate differential tests. These tests are generated after submission, vary across verifier runs, and remain invisible to the migration agent. A verifier can reject a submission only with an executable counterexample that passes on the original and fails on the migrated system. The three stages prevent migration shortcuts, enforce known behavioural requirements, and probe differences not encoded in the fixed suite.

The benchmark comprises 20 whole-repository migrations of real open-source infrastructure, including SQLite, zlib, libsodium, and GraphHopper. The tasks cover language (7), framework (7), platform (3), and build toolchain (3) migrations and give agents 6 to 30 hours of autonomous work per

task. We evaluate 8 frontier models under 26 model–effort configurations, running each configuration once on all 20 tasks for 520 scored runs. Current agents remain far from reliable whole-repository migration. The best configuration, `claude-opus-5` at xhigh effort, scores $47.0/100$; only 28 of 520 runs (5.4%) pass all three stages, and 13 of the 20 tasks receive no accepted solution. Migration completeness and behavioural correctness are distinct abilities, and agents miss them in opposite directions: 30 runs preserved behaviour by skipping the migration, and were stopped at Migration Audit; 252 completed the migration but broke behaviour, and were stopped at Behavioural Tests. Agents also struggle to close the final correctness gap: among the 340 runs that pass Migration Audit, 58% reach 99% of the fixed checks, yet only 26% reach 100%. These results establish a capability gap on the 20 tasks in SWE Refactor Bench; they do not rank the intrinsic difficulty of all migration projects.

In summary, we make three contributions.

- **Blindness and benchmark.** We identify *Blindness*: behaviour-only evaluation can award full credit without establishing that a migration occurred. And we introduce SWE Refactor Bench, a benchmark of 20 long-horizon, whole-repository migrations of real open-source infrastructure.
- **Three-stage adversarial evaluation.** We combine a hard migration audit and $130{,}118$ fixed behavioural checks with six independent coding agents that search for hidden differences after submission and can reject a migration only with an executable counterexample.
- **Revealing agent capability gaps.** "Getting the code right" is not "getting the migration done", and even doing both is not enough: among the 88 submissions that satisfy both under the fixed suite, agentic verifiers break 60.

2 SWE Refactor Bench: Benchmark Design

SWE Refactor Bench measures one ability: given a working repository built on one technology stack, can an agent deliver the same repository on another stack, with its observable behaviour intact and the old stack gone.

2.1 Task Formulation

A behaviour-preserving migration task is a tuple

$$\tau = (R_A,\ \Sigma_A \to \Sigma_B,\ \mathcal{O},\ \mathcal{I},\ E,\ B). \tag{1}$$

R_A—which we call State A—is a real open-source repository, taken at one commit, in a buildable and working condition. Σ_A is the stack it is built on and Σ_B the stack it must be built on afterwards: a language, an application framework, a host platform, or a build toolchain. $\mathcal{O}$ is the artifact's *observable interface*, the surface available for inspection once the repository has been built—process output and exit status, the symbols an installed library exports, the manifest of an installation tree, the responses of a served endpoint. $\mathcal{I}$ is the instruction given to the agent, E an offline image and B a time budget.

The agent's *input* is $(R_A, \mathcal{I}, E, B)$: a container holding the original repository, the toolchains of both stacks, and an instruction saying what is to be done, with no network. Its *output* is the working tree R_S at the end of the run. Writing $\mathcal{O}(R)$ for the observations produced by the artifact that repository R builds, R_S solves τ exactly when both of the following hold:

$$\Sigma_B \text{ builds the delivered artifact, and } \Sigma_A \text{ is absent from the repository and from the build closure,} \tag{2}$$

$$\mathcal{O}(R_S) = \mathcal{O}(R_A). \tag{3}$$

We call (2) the *migration condition* and (3) the *preservation condition*. The first is a claim about the text of a repository and about what its build compiles; the second is a claim about the behaviour of an artifact. No new functionality has to be designed anywhere in this work: State A already does everything State B must do; the whole difficulty is to rebuild those behaviours exactly, on a stack that expresses them in an entirely different way.

Why behavioural tests fail here. A behavioural test suite T is a finite set of observations drawn from $\mathcal{O}$; the score it gives a repository R is the fraction it passes, $\mathrm{rate}(R;T)$. Since the preservation

condition (3) is defined *relative to* R_A and $T \subseteq \mathcal{O}$, we have $\mathrm{rate}(R_A; T) = 1$ for any T—by construction, and for every task in this family. Setting $R_S = R_A$ (the *empty diff*: the agent hands the repository back untouched) therefore earns full marks on any behavioural suite while satisfying not one clause of the migration condition (2). This is exactly the *blindness* of Section 1: the maximum of the reward sits on a submission with zero work in it, and enlarging T does not help, because every case the migrated repository must pass is a case the original already passes.

2.2 Where the Tasks Come From

Pick the debt first, the repository second. We did not choose repositories and then invent a change for them; we fixed on a migration a maintainer would call overdue, then looked for a project where that migration *is* the whole job. Three admission requirements follow: *the old stack has to be load-bearing rather than incidental*, so that removing it reaches the design and not merely the import statements; *the observable interface has to be one that something outside actually depends on*—a C ABI, an HTTP API, an installation tree—so that "behaviour preserved" is a constraint with content rather than a slogan; and *there has to be a runnable reference*, since State A must be built and executed repeatedly for differential testing to be possible. What this admits is load-bearing infrastructure rather than exercises: SQLite (Gaffney et al., 2022), zlib and the DEFLATE format it implements (Deutsch, 1996), libsodium (Bernstein et al., 2012), and GraphHopper (Geisberger et al., 2008), among others.

Four kinds of technical debt. A repository's stack comes down to four things: what language it is written in, what framework it is organised around, what host it assumes, and what builds it. Each class of task moves exactly one of them (Table 5). *Language rewrites* (7 tasks) replace the implementation language while keeping the shape of the artifact: the original's implementation details are deleted along with its source, yet its external behaviour has to be reproduced exactly, and the new language will not make the same choices by itself. *Framework rewrites* (7) keep the language and replace the dependency the application is organised around: most of what has to be reproduced is not code the application wrote but what the framework did on its behalf—how errors are represented, how parameters are coerced, how content is negotiated. *Platform ports* (3) replace the host the code assumes: moving to `wasm32-wasi` (Haas et al., 2017) withdraws process control, the filesystem namespace and memory mapping, leaving only what the host explicitly grants. *Build-toolchain rewrites* (3) move the part that produces the artifact rather than the part that runs, so what is compared is what the program was packaged into.

What a task ships. Once (1) is instantiated, the agent receives a State A that builds cleanly, a State B declaration naming the target stack down to specific versions, an instruction stating the requirements and how the build is invoked, an offline image—which *also* carries State A's own toolchain, since the original is its reference—and an artifact contract. Everything the evaluation uses sits on the other side: the Migration Audit criteria, the Behavioural Tests expectations recorded from a reference build and run of State A, and the further checks the six verifiers of Agentic Verification perform. All three live in a separate image that is never mounted into the agent's container, so those tests are not merely "unread" by it—they do not exist for it. Appendix A instantiates all of this for one task.

2.3 The Three-Stage Protocol

A submission is the working tree at the end of the agent's run; nothing it built is carried into evaluation, and each stage rebuilds from the submitted source whatever it needs. The three stages are applied in order, and each of them can end the run—except that we ran the fixed suite on every submission, vetoed or not, since blindness cannot be counted otherwise; those runs still score zero. Figure 4 separates the agent's workspace and workflow from the hidden evaluation it never touches.

Stage I: Migration Audit. It asks one thing: *did the migration actually happen*—has the old stack disappeared from the repository and from the build. The criteria are written as prompt questions about this repository, answered one by one by a model reading both source trees; every failing verdict must cite re-checkable evidence, and each criterion is judged three times independently, with the majority taken.

Stage II: Behavioural Tests. It asks whether *behaviour survived unchanged*. These tests were not written out of thin air but built from State A: the same calls are run against the original repository and the recorded output becomes the expected answer, so the same inputs must afterwards produce the same outputs and the same exit status. Across the suite there are 130,118 checks, on average more than six thousand per task; a single wrong check scores zero, and only a clean sweep opens Stage III.

Stage III: Agentic Verification. The first two stages check what we thought of in advance; this one checks *what we did not*. A submission that reaches this point has answered the whole fixed suite correctly, so we change the kind of checker—six *coding agents*, one hour each, each holding both source trees and looking for something the original repository does that this submission does not. Five probe one assigned direction each (a C ABI, a set of routes, an installation tree) and the sixth is unrestricted, so they complement one another in coverage. What they hand in cannot be a report, only an executable test case: passing on State A and failing on the submission, going green against the reference first and then reproducing three times, so that a broken test and a flaky test both fail to win. This is differential testing (McKeeman, 1998) in an agentic form: a verifier may work the way a real tester does—generate inputs against a stated property (Claessen & Hughes, 2000), relate one output to another (Chen et al., 2018), or shrink a difference until it is reportable (Zeller & Hildebrandt, 2002)—but what it says does not count; only a difference that runs does.

Scoring. Let g be the Migration Audit verdict, r_i the pass rate of module i in Behavioural Tests, and $s \in \{0, \ldots, 6\}$ the number of verifiers that failed to produce a counterexample. A submission scores

$$S(\tau) \;=\; \underbrace{\mathbf{1}\big[g = \mathsf{pass}\big]}_{\text{Stage I: veto}} \cdot \underbrace{\mathbf{1}\big[r_i = 1 \;\; \forall i\big]}_{\text{Stage II: all or nothing}} \cdot \Big(0.4 \;+\; \underbrace{0.6 \cdot \tfrac{s}{6}}_{\text{Stage III: per verifier}}\Big) \;\in\; \{0\} \cup [0.4,\, 1]. \qquad (4)$$

Every factor corresponds to one condition from Section 2.1. Stage I *multiplies rather than adds*, because without the migration the task was not done at all, and no amount of behavioural correctness should buy points for it. Stage II is *all or nothing*, because what this family of tasks asks is whether this is a *drop-in* replacement, and a library that is wrong once in a thousand calls is not (Section 3.3.2 gives concrete examples). Stage III carries the larger share, 0.6, because it looks at a residue no fixed suite can see; it is scored linearly rather than all-or-nothing because *not broken* is not the same as *no difference*—six verifiers finding nothing is stronger evidence than one verifier finding nothing, but it is still evidence, not a demonstration of equivalence, and a linear score records precisely that degree.

3 Experiments

This section first states the setup and the metrics (Section 3.1), then answers two questions: how far today's agents get on whole-repository behaviour-preserving migration (Sections 3.2–3.3), and whether the evaluation itself holds up (Appendix E).

3.1 Setup and Metrics

We evaluate eight frontier models: `claude-opus-5` and `claude-sonnet-5`; `gpt-5.6-luna` and `gpt-5.6-sol`; `kimi-k3` (Kimi Team, 2025); `qwen3.8-max` (Yang et al., 2025); `dsv4-flash` (DeepSeek-AI, 2024); and `glm-5.2` (GLM-4.5 Team, 2025). The two GPT-series models use Codex as their harness; the other six use Claude Code. Every model ran all 20 tasks, for a total of 8 models, 26 configurations and 520 scored runs. A run is one independent attempt by one model at one task: a fresh container is created from that task's image, the instruction is handed to the agent, and it works on its own within the time budget the task declares, with no network beyond the model endpoint; when the budget runs out or it stops of its own accord, the working tree is collected as a submission and sent into the three-stage evaluation.

Six metrics, each answering a different question. Alongside the composite score of (4) we report counts, because a count states *what happened to* a submission, whereas a score only states it after weighting. Used throughout: **Migrated**, runs whose every migration criterion passed by majority; **All tests pass**, runs that passed every fixed check in Behavioural Tests; **Accepted**, runs that did both and survived all six verifiers; **Broken**, the *share* of runs that did both but were then defeated by at

Table 1: **The three-stage funnel: where each of the 520 runs stops.** Each row aggregates 20 runs, one per task. Values are numbers of runs; the five outcome columns are mutually exclusive and exhaustive, so they sum to 20 across a row. *Harness* identifies the coding-agent client: GPT-series models use Codex, and all others use Claude Code. Colour marks the stage at which a run stops: Migration Audit, Behavioural Tests, Agentic Verification, all three passed; within a column, darker means more runs in that cell. *Blindness* is a special kind of Stage I failure: all fixed checks pass, yet the repository was never migrated.

Model	Harness	Effort	Stopped at Stage I: not migrated		Stopped at Stage II	Stopped at Stage III	Passed
			fixed checks also fail	fixed checks pass (Blindness)	fixed checks fail	counterexample found	accepted
Claude Opus 5	Claude Code	low	2	0	13	3	**2**
Claude Opus 5	Claude Code	medium	1	2	10	5	**2**
Claude Opus 5	Claude Code	high	0	2	10	4	**4**
Claude Opus 5	Claude Code	xhigh	0	1	8	6	**5**
Claude Opus 5	Claude Code	max	2	2	9	4	**3**
GPT-5.6 Sol	Codex	none	11	0	8	1	0
GPT-5.6 Sol	Codex	low	11	0	7	2	0
GPT-5.6 Sol	Codex	medium	8	3	7	2	0
GPT-5.6 Sol	Codex	high	8	0	7	4	**1**
GPT-5.6 Sol	Codex	xhigh	5	1	11	3	0
GPT-5.6 Sol	Codex	max	5	0	8	3	**4**
Kimi K3	Claude Code	max	5	0	10	3	**2**
Claude Sonnet 5	Claude Code	low	5	1	13	1	0
Claude Sonnet 5	Claude Code	medium	6	0	10	3	**1**
Claude Sonnet 5	Claude Code	high	3	2	13	2	0
Claude Sonnet 5	Claude Code	xhigh	7	1	10	1	**1**
Claude Sonnet 5	Claude Code	max	5	1	12	1	**1**
GPT-5.6 Luna	Codex	none	10	1	8	1	0
GPT-5.6 Luna	Codex	low	14	1	4	1	0
GPT-5.6 Luna	Codex	medium	7	1	12	0	0
GPT-5.6 Luna	Codex	high	6	2	11	1	0
GPT-5.6 Luna	Codex	xhigh	4	2	12	2	0
GPT-5.6 Luna	Codex	max	2	1	14	3	0
Qwen 3.8 Max	Claude Code	max	8	4	6	0	**2**
DeepSeek V4 Flash	Claude Code	max	6	1	11	2	0
GLM 5.2	Claude Code	max	9	1	8	2	0

least one verifier; **Blindness**, runs that passed every fixed check yet were vetoed at Stage I—the phenomenon of Section 1, counted; and **Score**, the mean of (4) over runs, out of 100.

3.2 Overall Performance

Leaderboard. Comparing strongest configurations, `claude-opus-5` leads on acceptances and score: 5 acceptances in 20 runs and a score of 47.0; behind it are `gpt-5.6-sol` at 28.5, `kimi-k3` at 19.5 and `claude-sonnet-5` at 15.0. Across the eight models in their strongest configurations, 160 runs yield only 14 acceptances, and three models never earned one at all despite producing runs that passed every fixed check. Appendix D.1 gives all 26 configurations with their pass rate, score and cost.

The funnel is narrow, and it narrows for three different reasons. Table 1 decomposes each row's 20 runs into five mutually exclusive outcomes. Of the 520 scored runs, 340 (65.4%) passed Stage I and 118 (22.7%) passed every fixed check, but only 88 did both and thereby reached Stage III, of which 28 (5.4%) survived all six verifiers; 13 of the 20 tasks were never solved by any model. The mean score over all 520 runs is only 13.44/100, but over the 88 that reached Stage III it is 79.43—so the difficulty is in getting to Stage III at all. Moreover the three gates stop *different* submissions; no group is filtered out twice: stages that overlap are redundant, whereas stages that are disjoint are measuring three different things. With the fixed suite as the only instrument, 118 submissions would tie for first place, among them submissions that never migrated anything and submissions that a verifier breaks within the hour.

Outcomes also vary by migration category (Table 8). Build toolchain rewrites have the highest Stage I pass rate (80.8%) and mean score (31.4), whereas framework rewrites contribute 14 of the 28 accepted runs; Appendix D.3 breaks this down stage by stage.

3.3 Analysis of Agent Behaviour

3.3.1 "Getting the code right" and "getting the migration done" are two different abilities

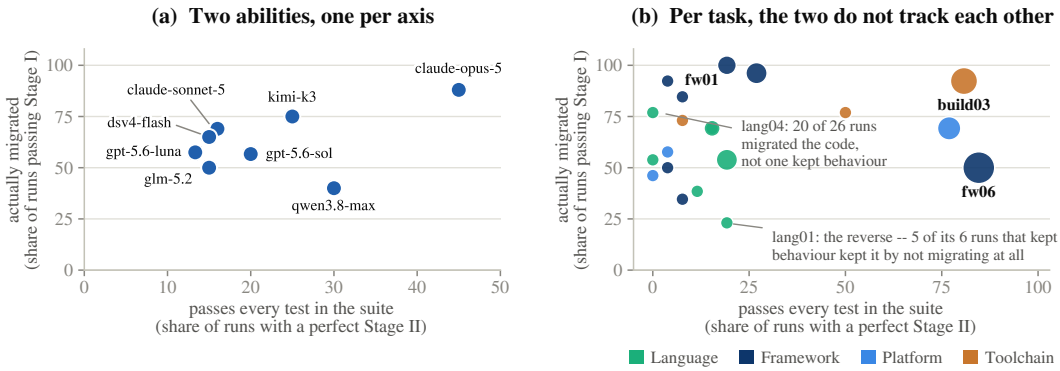

Figure 2: **(a)** Where each model sits on two axes, both of them shares of the same runs: horizontally "does it still behave like the original" (a perfect Stage II), vertically "was the migration done at all" (Stage I). **(b)** The same two axes, per task, with marker area the number of accepted submissions. A migration has to satisfy both conditions, but they are satisfied by *different* runs, and most runs manage at most one.

Put the two axes together and the failures fall into two groups (Figure 2a). One is the runs that never migrated and kept every fixed check: a submission that barely migrates anything passes any behavioural suite by construction, and 30 runs did exactly that, spread over 7 of the 8 models. Stage II gives all 30 full marks; only Stage I stops them. The other is eight times as large: 252 runs completed the migration and broke behaviour doing it, and only Stage II stops those. Neither stage can stand in for the other, then: Stage II alone would reward doing nothing, Stage I alone would reward doing damage.

Per task, the two axes diverge just as clearly (Figure 2b), and in both directions. On `lang04` (acorn, JavaScript → Rust), 20 of 26 runs passed Stage I, yet *not one* passed every fixed check. `lang01` (cmark, C → Rust) is the mirror image: only 6 runs passed Stage I, yet 5 passed every fixed check—and *all five* of those are blindness. Each cleared seven of the eight criteria and failed only the one asking whether the Rust is the implementation: it reproduces the original's control flow statement for statement, a transliteration in which ownership was never redesigned, so the safety this migration exists to buy was not bought. No behavioural test can express that, because behaviour was never where the problem was; Appendix A shows one such run in full, and Appendix D.2 the four shapes these submissions take.

3.3.2 The last 1%: agents cannot deliver a perfect migration

For a repository migration, any failing unit test is a serious risk: behind that failing test stands a real downstream consumer, and it will not be spared because the other 99.99% of the behaviour is right. And this is precisely the step agents cannot get past. Among the 340 runs that passed Stage I (Figure 3a), 91% get the fixed suite past half, 58% reach 99% and 36% reach 99.9%—but only 26% make no error at all. That final step alone eliminates 35 of the 123 runs that had already reached 99.9%; across the campaign 140 runs land in $[99\%, 100\%)$, missing a median of 12.5 checks, and 18 of them miss exactly one.

Those 18 are not scattered at random; they concentrate on a handful of specific checks. On `fw03` (conduit, Vue → React), four different models all ended at $21768/21769$, failing the same one: the original uses hash routing, so visiting `/` settles at `/#/` while the React version stays at `/`, which

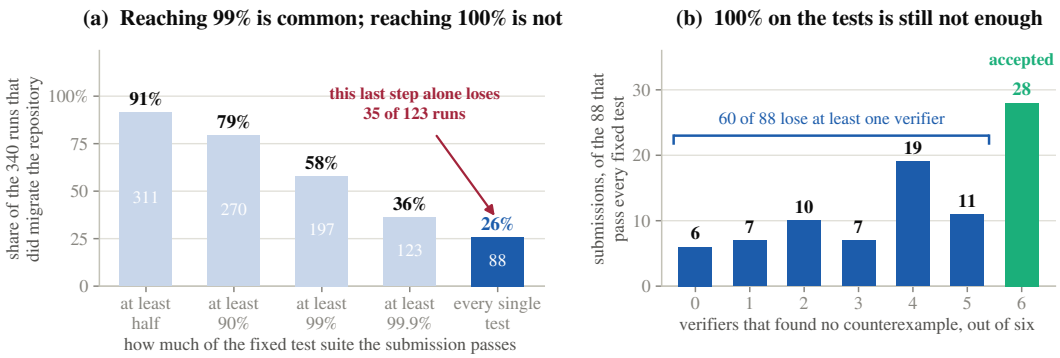

Figure 3: **(a)** Among the 340 runs that *did* complete the migration, the share reaching each level on the fixed suite. **(b)** Among the 88 submissions that passed every fixed check, how many of the six independent verifiers (one hour each) failed to find a behavioural difference. Only the rightmost bar is an accepted migration.

Table 2: **Positioning against related benchmarks.** *Starts failing* means whether the starting state already fails some test in the suite being scored; that is what makes "red to green" an informative verdict—and it is exactly the property SWE Refactor Bench cannot have. *Migration criterion* means whether a non-behavioural instrument can reject a submission that passed every behavioural check; *agentic verifier* means whether, at scoring time, further models actively search for differences beyond the fixed tests that the authors did not anticipate. ✓ is yes, (✓) is partial—MigrationBench moves a language version, Java 8 to 17/21, while the language itself stays—and × marks a dimension the benchmark does not aim at, not a defect. The three marks differ in glyph as well as colour, so the table reads the same in black and white.

Benchmark	Unit of work	Starts failing	Stack changes	Migration criterion	Agentic verifier
SWE-bench (Jimenez et al., 2024)	one issue	✓	×	×	×
SWE-EVO (Le et al., 2025)	one feature	✓	×	×	×
SWE-CI (Chen et al., 2026)	CI history	✓	×	×	×
Terminal-Bench (Merrill et al., 2026)	one task	✓	×	×	×
TransCoder (Rozière et al., 2020)	one function	✓	✓	×	×
MigrationBench (Liu et al., 2025)	whole repository	✓	(✓)	×	×
SWE Refactor Bench (this work)	whole repository	×	✓	✓	✓

breaks every bookmark and every shared link to the site. On `build03` (PyCryptodome, setuptools → Meson), five different models all ended at 380/381: the `METADATA` long description in the built wheel is 0 characters, so once the package is published its PyPI project page is blank. Not one of these checks can be called nitpicking: each corresponds to a regression that would cause trouble in production, and the original repository passes *all* of them. In other words, this is not a problem with the test suite; it is a problem with the migration.

Making no test error is still not enough. Among the 88 submissions that did not miss a single fixed check, only 28 survived all six verifiers; the other 60 (68.2%) had a counterexample found against them within the hour (Figure 3b), with the average submission holding off only 3.94 of the 6 verifiers. In other words, an instrument that looks only at behaviour would have accepted all 88 of these submissions, and Stage III took back two thirds of them. When a break comes, it comes quickly: the median time to a counterexample is 17.0 minutes, against 32.8 minutes for a survival.

Whether the instrument itself holds up—how far the Stage I judge agrees with independent human annotation (89.7%, $\kappa = 0.795$), what Stage III catches that a fixed suite cannot, and whether all six verifiers are needed—is examined in Appendix E.

4 Related Work

Repository-level coding benchmarks: the signal is "red to green". SWE-bench (Jimenez et al., 2024) established the paradigm the field now uses: a real repository, a real issue, and a test that fails before the patch and passes after it. The paradigm has since been extended—to other languages (Zan

et al., 2025), to continuously refreshed task streams that resist contamination (Zhang et al., 2025; Jain et al., 2025), to building a whole library from scratch (Zhao et al., 2025), and to writing tests rather than patches (Mündler et al., 2024). A neighbouring group of suites keeps the same criterion and only stretches the horizon: repository evolution (SWE-EVO) (Le et al., 2025; Li et al., 2024), long continuous-integration histories (SWE-CI) (Chen et al., 2026), and hours of terminal work (Terminal-Bench) (Merrill et al., 2026). The criterion is inherited from function-level suites (Chen et al., 2021) and fits all of them: while the work is unfinished the test is red, when it is finished it turns green, and that jump is the evidence of completion. SWE Refactor Bench's tasks have no such jump, because the starting state is already green—not a defect of those benchmarks but a property of behaviour-preserving evolution.

Code migration and reward hacking: fixed tests are not enough. Behaviour-preserving rewriting is not new: translation between languages stayed at the function level for a long time after TransCoder (Rozière et al., 2020), and the difficulty of evaluating it was documented early (Pan et al., 2024); the unit later grew to a whole repository (Wang et al., 2024b; Ibrahimzada et al., 2024), and C to Rust acquired dedicated benchmarks of its own (Khatry et al., 2025; Emre et al., 2021); other work moves not the language but a language version (Liu et al., 2025) or a single API (Li et al., 2025). The changes differ, the scoring does not: take a test suite, run it, count what passes—reusing the original repository's tests when the language stays the same, and having humans write them on the target side when it does not. For a whole-repository change of stack that scoring is no longer sufficient: a repository handed back as-is still turns any fixed suite green, so fixed tests can say at most "nothing was broken", never "something was changed". It is tempting to read this as reward hacking—agents optimise whatever their reward leaves unconstrained (Skalse et al., 2022; Krakovna et al., 2020; Baker et al., 2025), coding environments have supplied documented cases (Bercovich et al., 2026; Wang et al., 2026), and SWE-bench itself has been audited for leakage and contamination (Liang et al., 2025; Garg et al., 2025; Xu et al., 2024). But a repository handed back as-is circumvents no check: it earns full marks by the rules, and the rules cannot see whether the migration happened. The hole is not in how tight the tests are, so no amount of extra tests will close it (Section 2.1); the only remedy is a second check outside behaviour, holding a veto (Liang et al., 2023).

Model-based judging and differential testing: the tools at the two ends of this work. When what must be decided cannot be written as a scoring script, using a model as the judge is standard practice (Zheng et al., 2023); its known failure modes—position and verbosity bias, self-preference (Wang et al., 2024a; Panickssery et al., 2024)—are why Stage I answers narrow questions with cited evidence over three independent samples rather than producing one global rating, and why we measure its consistency (Appendix E). Stage III is differential testing (McKeeman, 1998), the way compilers have long been checked at scale (Yang et al., 2011; Deng et al., 2023): what a verifier submits is not an opinion but evidence that runs. Formal verification would be stronger, but translation validation (Pnueli et al., 1998), differential symbolic execution (Lahiri et al., 2012; Cadar et al., 2008) and verified compilers (Leroy, 2009) all require two sides whose semantics can be related, which a cross-language whole-repository rewrite does not offer. Agentic verifiers approximate the same effect: if the strongest coding agents, holding both source trees, produce no counterexample, that is the strongest evidence we are able to give. Table 2 places SWE Refactor Bench next to the benchmarks above, dimension by dimension.

5 Conclusion

We have presented SWE Refactor Bench: 20 long-horizon whole-repository migrations of real open-source infrastructure, scored by a three-stage protocol that does not rely on behaviour alone—a migration audit that holds a veto, 130,118 fixed checks recorded from the original, and six coding agents that accept nothing but an executable counterexample. Across 520 evaluations on 8 frontier models, only 28 (5.4%) passed all three stages, and 13 of the 20 tasks were solved by nobody. The failures follow a pattern: "getting the migration done" and "not breaking anything" are two different abilities, and agents missed them in opposite directions. Even when the migration was genuinely done, only 26% of runs passed every fixed check, and two thirds of those still had a counterexample found against them within the hour. There is a long way to go before an agent can deliver a whole-repository migration; and how far along that road we are will be measured not by how much code the agent wrote, but by whether, once it is done, the system is still the same system.

AI USE STATEMENT

In this work, we used generative AI tools for generating synthetic datasets, for implementing methods and for result interpretation, and generative AI is itself a declared component of the evaluation protocol: Migration Audit uses a language model to answer the migration criteria, and Agentic Verification uses six coding agents to construct differential tests (Section 2.3). Task construction was AI-assisted: parts of the Migration Audit criteria and of the fixed Behavioural Tests suites were drafted with generative AI and then reviewed and corrected by the authors. Parts of the evaluation harness, the analysis scripts and the plotting code were written with AI assistance. We have not used generative AI tools for translation assistance, for developing theoretical models, or for formulating or proving mathematical claims, and the remaining tasks requiring disclosure are not applicable to this work. Additionally, we used generative AI tools for editing the manuscript for readability and, in part, for literature identification and summarisation; the text itself was written by the authors. We have reviewed all AI-assisted work: every AI-assisted component was examined by hand and cross-reviewed by at least three authors, and every quantitative claim in this paper is recomputed from the recorded evaluation logs rather than produced by a model. We take responsibility for the final content of this work, including text, claims or artifacts produced with the aid of generative AI.

REPRODUCIBILITY STATEMENT

We release the benchmark in full: the 20 task images, the Migration Audit criteria, the fixed Behavioural Tests expectations, the Agentic Verification harness, the analysis scripts, and the raw logs of all 520 scored runs. An anonymised copy is available for review at `https://anonymous.4open.science/r/SWE-Refactor-Bench/`. The task formulation and the exact acceptance conditions are stated in Section 2.1; the three-stage protocol and the scoring function in Section 2.3 and equation (4); and the construction procedure for a task, end to end, in Appendix A. Section 3.1 gives the 8 models, the 26 model–effort configurations and the one-run-per-task design behind the 520 runs.

REFERENCES

Mei Akizuru. go-simple-upload-server v2.2.0. https://github.com/mayth/go-simple-upload-server, 2026. MIT licence.

Andi Albrecht. sqlparse 0.5.3. https://github.com/andialbrecht/sqlparse, 2024. BSD-3-Clause licence.

Automattic. Stylus 0.63.0: an expressive CSS preprocessor. https://github.com/stylus/stylus, 2024. MIT licence.

Bowen Baker, Joost Huizinga, Leo Gao, Zehao Dou, Melody Y. Guan, Aleksander Madry, Wojciech Zaremba, Jakub Pachocki, and David Farhi. Monitoring reasoning models for misbehavior and the risks of promoting obfuscation. *arXiv preprint arXiv:2503.11926*, 2025.

Fabrice Bellard and Charlie Gordon. QuickJS 2020-11-08: a small embeddable JavaScript engine. https://bellard.org/quickjs/, 2020. MIT licence, stated in the header of every source file.

Ivan Bercovich, Ivgeni Segal, Kexun Zhang, Shashwat Saxena, Aditi Raghunathan, and Ziqian Zhong. Terminal wrench: A dataset of 331 reward-hackable environments and 3,632 exploit trajectories. *arXiv preprint arXiv:2604.17596*, 2026.

Daniel J. Bernstein, Tanja Lange, and Peter Schwabe. The security impact of a new cryptographic library. In *International Conference on Cryptology and Information Security in Latin America (LATINCRYPT)*, 2012.

Cristian Cadar, Daniel Dunbar, and Dawson Engler. KLEE: Unassisted and automatic generation of high-coverage tests for complex systems programs. In *USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, 2008.

Canonical Ltd. go-yaml v3.0.1: YAML support for the Go language. https://github.com/go-yaml/yaml, 2022. Apache-2.0 and MIT licences.

Jialong Chen, Xander Xu, Hu Wei, Chuan Chen, and Bing Zhao. SWE-CI: Evaluating agent capabilities in maintaining codebases via continuous integration. *arXiv preprint arXiv:2603.03823*, 2026.

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.

Tsong Yueh Chen, Fei-Ching Kuo, Huai Liu, Pak-Lok Poon, Dave Towey, T. H. Tse, and Zhi Quan Zhou. Metamorphic testing: A review of challenges and opportunities. *ACM Computing Surveys*, 51(1):4:1–4:27, 2018. doi: 10.1145/3143561.

Koen Claessen and John Hughes. QuickCheck: A lightweight tool for random testing of Haskell programs. In *ACM SIGPLAN International Conference on Functional Programming (ICFP)*, 2000.

Ward Cunningham. The WyCash portfolio management system. In *Addendum to the Proceedings on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA Addendum), Experience Report*, pp. 29–30, 1992.

DeepSeek-AI. DeepSeek-V3 technical report. *arXiv preprint arXiv:2412.19437*, 2024.

Yinlin Deng, Chunqiu Steven Xia, Haoran Peng, Chenyuan Yang, and Lingming Zhang. Large language models are zero-shot fuzzers: Fuzzing deep-learning libraries via large language models. In *ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA)*, 2023. arXiv:2212.14834.

Frank Denis. libsodium 1.0.20. https://github.com/jedisct1/libsodium, 2024. ISC licence.

L. Peter Deutsch. DEFLATE compressed data format specification version 1.3. Technical Report RFC 1951, Internet Engineering Task Force, 1996.

Mehmet Emre, Ryan Schroeder, Kyle Dewey, and Ben Hardekopf. Translating C to safer Rust. *Proceedings of the ACM on Programming Languages*, 5(OOPSLA):121, 2021. doi: 10.1145/3485498.

Kevin P. Gaffney, Martin Prammer, Larry Brasfield, D. Richard Hipp, Dan Kennedy, and Jignesh M. Patel. SQLite: Past, present, and future. *Proceedings of the VLDB Endowment*, 15(12):3535–3547, 2022. doi: 10.14778/3554821.3554842.

Jean-loup Gailly and Mark Adler. zlib 1.3.1. https://github.com/madler/zlib, 2024. Zlib licence.

Spandan Garg, Benjamin Steenhoek, and Yufan Huang. Saving SWE-bench: A benchmark mutation approach for realistic agent evaluation. *arXiv preprint arXiv:2510.08996*, 2025.

Robert Geisberger, Peter Sanders, Dominik Schultes, and Daniel Delling. Contraction hierarchies: Faster and simpler hierarchical routing in road networks. In *International Workshop on Experimental Algorithms (WEA)*, 2008.

GLM-4.5 Team. GLM-4.5: Agentic, reasoning, and coding (ARC) foundation models. *arXiv preprint arXiv:2508.06471*, 2025.

Google. Gson gson-parent-2.10.1. https://github.com/google/gson, 2023. Apache-2.0 licence.

Google Inc. Jsonnet 0.20.0: a data templating language. https://github.com/google/jsonnet, 2023. Apache-2.0 licence.

GraphHopper GmbH. GraphHopper 11.0. https://github.com/graphhopper/graphhopper, 2025. Apache-2.0 licence.

Andreas Haas, Andreas Rossberg, Derek L. Schuff, Ben L. Titzer, Michael Holman, Dan Gohman, Luke Wagner, Alon Zakai, and JF Bastien. Bringing the web up to speed with WebAssembly. In *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, 2017.

Sven-Hendrik Haase. miniserve 0.27.1. https://github.com/svenstaro/miniserve, 2024. MIT licence.

IBM Corp. JSONata 2.2.2: a query and transformation language for JSON. https://github.com/jsonata-js/jsonata, 2026. MIT licence.

Ali Reza Ibrahimzada, Kaiyao Ke, Mrigank Pawagi, Muhammad Salman Abid, Rangeet Pan, Saurabh Sinha, and Reyhaneh Jabbarvand. AlphaTrans: A neuro-symbolic compositional approach for repository-level code translation and validation. *arXiv preprint arXiv:2410.24117*, 2024.

Naman Jain, King Han, Alex Gu, Wen-Ding Li, Fanjia Yan, Tianjun Zhang, Sida Wang, Armando Solar-Lezama, Koushik Sen, and Ion Stoica. LiveCodeBench: Holistic and contamination free evaluation of large language models for code. In *International Conference on Learning Representations (ICLR)*, 2025. arXiv: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? In *International Conference on Learning Representations (ICLR)*, 2024. arXiv:2310.06770.

Anirudh Khatry, Robert Zhang, Jia Pan, Ziteng Wang, Qiaochu Chen, Greg Durrett, and Isil Dillig. CRUST-bench: A comprehensive benchmark for C-to-safe-Rust transpilation. *arXiv preprint arXiv:2504.15254*, 2025.

Kimi Team. Kimi K2: Open agentic intelligence. *arXiv preprint arXiv:2507.20534*, 2025.

Victoria Krakovna, Jonathan Uesato, Vladimir Mikulik, Matthew Rahtz, Tom Everitt, Ramana Kumar, Zac Kenton, Jan Leike, and Shane Legg. Specification gaming: The flip side of AI ingenuity. DeepMind blog. https://deepmind.google/blog/specification-gaming-the-flip-side-of-ai-ingenuity/, 2020.

Shuvendu K. Lahiri, Chris Hawblitzel, Ming Kawaguchi, and Henrique Rebêlo. SYMDIFF: A language-agnostic semantic diff tool for imperative programs. In *International Conference on Computer Aided Verification (CAV)*, volume 7358 of *Lecture Notes in Computer Science*, pp. 712–717. Springer, 2012. doi: 10.1007/978-3-642-31424-7_54.

Tue Le, Minh V. T. Thai, Dung Nguyen Manh, Huy Phan Nhat, and Nghi D. Q. Bui. SWE-EVO: Benchmarking coding agents in long-horizon software evolution scenarios. *arXiv preprint arXiv:2512.18470*, 2025.

Xavier Leroy. Formal verification of a realistic compiler. *Communications of the ACM*, 52(7): 107–115, 2009. doi: 10.1145/1538788.1538814.

Jia Li, Ge Li, Xuanming Zhang, Yunfei Zhao, Yihong Dong, Zhi Jin, Binhua Li, Fei Huang, and Yongbin Li. EvoCodeBench: An evolving code generation benchmark with domain-specific evaluations. In *Advances in Neural Information Processing Systems (NeurIPS), Datasets and Benchmarks Track*, 2024. arXiv:2410.22821.

Tianyu Li, Ruishi Li, Bo Wang, Brandon Paulsen, Umang Mathur, and Prateek Saxena. Adversarial agent collaboration for correctness improvements of C to safe Rust translation. *arXiv preprint arXiv:2510.03879*, 2025.

Percy Liang, Rishi Bommasani, Tony Lee, Dimitris Tsipras, Dilara Soylu, Michihiro Yasunaga, Yian Zhang, Deepak Narayanan, Yuhuai Wu, Ananya Kumar, Benjamin Newman, Binhang Yuan, Bobby Yan, Ce Zhang, Christian Cosgrove, Christopher D. Manning, Christopher Ré, Diana Acosta-Navas, Drew A. Hudson, Eric Zelikman, Esin Durmus, Faisal Ladhak, Frieda Rong, Hongyu Ren, Huaxiu Yao, Jue Wang, Keshav Santhanam, Laurel Orr, Lucia Zheng, Mert Yuksekgonul, Mirac Suzgun, Nathan Kim, Neel Guha, Niladri S. Chatterji, Omar Khattab, Peter Henderson, Qian Huang, Ryan Andrew Chi, Sang Michael Xie, Shibani Santurkar, Surya Ganguli, Tatsunori Hashimoto, Thomas Icard, Tianyi Zhang, Vishrav Chaudhary, William Wang, Xuechen Li, Yifan Mai, Yuhui Zhang, and Yuta Koreeda. Holistic evaluation of language models. *Transactions on Machine Learning Research*, 2023. arXiv:2211.09110.

Shanchao Liang, Spandan Garg, and Roshanak Zilouchian Moghaddam. The SWE-bench illusion: When state-of-the-art LLMs remember instead of reason. *arXiv preprint arXiv:2506.12286*, 2025.

Linbo Liu, Xinle Liu, Qiang Zhou, Lin Chen, Yihan Liu, Hoan Nguyen, Behrooz Omidvar-Tehrani, Xi Shen, Jun Huan, Omer Tripp, and Anoop Deoras. MigrationBench: Repository-level code migration benchmark from Java 8. *arXiv preprint arXiv:2505.09569*, 2025.

John MacFarlane. cmark 0.31.1: the CommonMark reference implementation. https://github.com/commonmark/cmark, 2024. BSD-2-Clause, MIT and CC-BY-SA-4.0.

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

Mike A. Merrill et al. Terminal-Bench: Benchmarking agents on hard, realistic tasks in command line interfaces. *arXiv preprint arXiv:2601.11868*, 2026.

Niels Mündler, Mark Niklas Müller, Jingxuan He, and Martin Vechev. SWT-bench: Testing and validating real-world bug-fixes with code agents. In *Advances in Neural Information Processing Systems (NeurIPS)*, 2024. arXiv:2406.12952.

Rangeet Pan, Ali Reza Ibrahimzada, Rahul Krishna, Divya Sankar, Lambert Pouguem Wassi, Michele Merler, Boris Sobolev, Raju Pavuluri, Saurabh Sinha, and Reyhaneh Jabbarvand. Lost in translation: A study of bugs introduced by large language models while translating code. In *IEEE/ACM International Conference on Software Engineering (ICSE)*, 2024. arXiv:2308.03109.

Arjun Panickssery, Samuel R. Bowman, and Shi Feng. LLM evaluators recognize and favor their own generations. *arXiv preprint arXiv:2404.13076*, 2024.

Amir Pnueli, Michael Siegel, and Eli Singerman. Translation validation. In *International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, volume 1384 of *Lecture Notes in Computer Science*, pp. 151–166. Springer, 1998. doi: 10.1007/BFb0054170.

Kenneth Reitz. httpbin 0.10.2. https://github.com/psf/httpbin, 2024. ISC or MIT licence.

Baptiste Rozière, Marie-Anne Lachaux, Lowik Chanussot, and Guillaume Lample. Unsupervised translation of programming languages. In *Advances in Neural Information Processing Systems (NeurIPS)*, 2020. arXiv:2006.03511.

Joar Skalse, Nikolaus H. R. Howe, Dmitrii Krasheninnikov, and David Krueger. Defining and characterizing reward hacking. In *Advances in Neural Information Processing Systems (NeurIPS)*, 2022. arXiv:2209.13085.

The Acorn Contributors. Acorn 8.14.0: a small, fast JavaScript parser. https://github.com/acornjs/acorn, 2024. MIT licence.

The Helm Authors. ChartMuseum v0.15.0. https://github.com/helm/chartmuseum, 2022. Apache-2.0 licence.

The PyCryptodome Authors. PyCryptodome 3.20.0. https://github.com/Legrandin/pycryptodome, 2024. BSD-2-Clause and Unlicense.

The RealWorld Contributors. vue-realworld-example-app, commit feb0b7d2. https://github.com/gothinkster/vue-realworld-example-app, 2019. MIT licence.

The SQLite Developers. SQLite 3.31.1. https://www.sqlite.org/, 2020. Public domain; SPDX identifier `blessing`.

typicode. json-server 0.17.4. https://github.com/typicode/json-server, 2023. MIT licence.

Hao Wang, Hanchen Li, Qiuyang Mang, Alvin Cheung, Koushik Sen, and Dawn Song. Do androids dream of breaking the game? systematically auditing AI agent benchmarks with BenchJack. *arXiv preprint arXiv:2605.12673*, 2026.

Peiyi Wang, Lei Li, Liang Chen, Zefan Cai, Dawei Zhu, Binghuai Lin, Yunbo Cao, Lingpeng Kong, Qi Liu, Tianyu Liu, and Zhifang Sui. Large language models are not fair evaluators. In *Annual Meeting of the Association for Computational Linguistics (ACL)*, 2024a. arXiv:2305.17926.

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.

Yanli Wang, Yanlin Wang, Suiquan Wang, Daya Guo, Jiachi Chen, John Grundy, Xilin Liu, Yuchi Ma, Mingzhi Mao, Hongyu Zhang, and Zibin Zheng. RepoTransBench: A real-world multilingual benchmark for repository-level code translation. *arXiv preprint arXiv:2412.17744*, 2024b.

Cheng Xu, Shuhao Guan, Derek Greene, and M-Tahar Kechadi. Benchmark data contamination of large language models: A survey. *arXiv preprint arXiv:2406.04244*, 2024.

An Yang, Anfeng Li, Baosong Yang, Beichen Zhang, Binyuan Hui, Bo Zheng, Bowen Yu, Chang Gao, Chengen Huang, Chenxu Lv, Chujie Zheng, Dayiheng Liu, Fan Zhou, Fei Huang, Feng Hu, Hao Ge, Haoran Wei, Huan Lin, Jialong Tang, Jian Yang, Jianhong Tu, Jianwei Zhang, Jianxin Yang, Jiaxi Yang, Jing Zhou, Jingren Zhou, Junyang Lin, Kai Dang, Keqin Bao, Kexin Yang, Le Yu, Lianghao Deng, Mei Li, Mingfeng Xue, Mingze Li, Pei Zhang, Peng Wang, Qin Zhu, Rui Men, Ruize Gao, Shixuan Liu, Shuang Luo, Tianhao Li, Tianyi Tang, Wenbiao Yin, Xingzhang Ren, Xinyu Wang, Xinyu Zhang, Xuancheng Ren, Yang Fan, Yang Su, Yichang Zhang, Yinger Zhang, Yu Wan, Yuqiong Liu, Zekun Wang, Zeyu Cui, Zhenru Zhang, Zhipeng Zhou, and Zihan Qiu. Qwen3 technical report. *arXiv preprint arXiv:2505.09388*, 2025.

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 (NeurIPS)*, 2024. arXiv:2405.15793.

Xuejun Yang, Yang Chen, Eric Eide, and John Regehr. Finding and understanding bugs in C compilers. In *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, 2011.

Daoguang Zan, Zhirong Huang, Wei Liu, Hanwu Chen, Linhao Zhang, Shulin Xin, Lu Chen, Qi Liu, Xiaojian Zhong, Aoyan Li, Siyao Liu, Yongsheng Xiao, Liangqiang Chen, Yuyu Zhang, Jing Su, Tianyu Liu, Rui Long, Kai Shen, and Liang Xiang. Multi-SWE-bench: A multilingual benchmark for issue resolving. *arXiv preprint arXiv:2504.02605*, 2025.

Andreas Zeller and Ralf Hildebrandt. Simplifying and isolating failure-inducing input. *IEEE Transactions on Software Engineering*, 28(2):183–200, 2002.

Linghao Zhang, Shilin He, Chaoyun Zhang, Yu Kang, Bowen Li, Chengxing Xie, Junhao Wang, Maoquan Wang, Yufan Huang, Shengyu Fu, Elsie Nallipogu, Qingwei Lin, Yingnong Dang, Saravan Rajmohan, and Dongmei Zhang. SWE-bench goes live! *arXiv preprint arXiv:2505.23419*, 2025.

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. arXiv:2412.01769.

Lianmin Zheng, Wei-Lin Chiang, Ying Sheng, Siyuan Zhuang, Zhanghao Wu, Yonghao Zhuang, Zi Lin, Zhuohan Li, Dacheng Li, Eric P. Xing, Hao Zhang, Joseph E. Gonzalez, and Ion Stoica. Judging LLM-as-a-judge with MT-bench and chatbot arena. In *Advances in Neural Information Processing Systems (NeurIPS)*, 2023. arXiv:2306.05685.

A How a Task Is Built

Every task is built by the same procedure. This appendix walks through `lang01-cmark-c-to-rust`: its State A is cmark 0.31.1, the CommonMark reference implementation, 41 C source files, built by CMake into `libcmark.so.0.31.1`, `libcmark.a` and `cmark(1)`; its State B is those same three artifacts, behind the same `cmark.h`, implemented in Rust 1.90 without any external crate and with the C ABI unchanged.

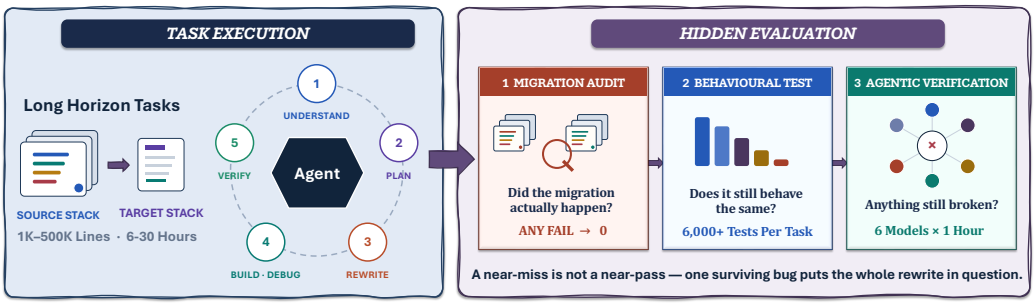

Figure 4: **On the left, what the agent sees**: a real repository on the source stack, an instruction, an offline image, and the process it works through on its own—read the repository and the requirements, rewrite onto the target stack, get it to build, then check itself against the original. **On the right, the three-stage evaluation it never touches**: Migration Audit checks whether the migration actually happened, Behavioural Tests checks whether the fixed behaviour is preserved exactly, and Agentic Verification goes looking for whatever differences remain beyond the fixed tests.

What a task ships with. Instantiating (1) means putting eight things on disk: (i) the archive of State A, rebuilt to confirm that it works; (ii) a State B declaration naming the target stack down to fixed versions; (iii) the instruction $\mathcal{I}$; (iv) the offline image E, which also carries State A's own toolchain; (v) an artifact contract saying what will be collected from the finished workspace; (vi) the fixed Stage II expectations, recorded from a reference build of State A and divided into modules; (vii) the migration criteria; and (viii) the further checks made by the agentic verifiers. Items (vi)–(viii) live in a separate image that is never mounted into the agent's container, so those tests are not merely "unread" by it—they do not exist for it.

Pick the debt first, the repository second. We start from a migration a maintainer would call overdue and then look for a project where that migration is the whole job: the old stack has to be load-bearing, so that removing it reaches the design rather than just the imports. Why cmark qualifies is worth spelling out, since CommonMark parsers in Rust already exist: what is wanted here is not a correct parser but *this* library's ABI, installation tree and so on, which no existing crate provides. The repository is then trimmed, archived and rebuilt inside the image to confirm it works, with history squashed to a single commit—an upstream log is a place where some past migration may well already be described, and `git log` is the first thing a competent agent reads.

Write the requirements, but not the tests. $\mathcal{I}$ states the requirements in full: the files that must survive untouched, how the build has to remain invocable without a network, and in which direction behaviour will be compared. What it never contains is a single one of the concrete behavioural checks—an instruction that enumerates the tests is an instruction to satisfy the tests.

Scoring reads source only. An exclusion list is applied when the workspace is collected (`build-*`, `target/`, object files, install prefixes, `.git`), so "the old stack no longer exists" is a fact about source rather than a fact about what the agent happened to leave in some directory.

Draw the boundary for the verifiers. The allow list and deny list of Stage III have to agree with $\mathcal{I}$: a scope that forbids what the instruction promised makes the task unwinnable, and a scope that allows what the instruction excluded makes it unloseable. Within that scope a verifier may work the way a real tester does (Claessen & Hughes, 2000; Chen et al., 2018; Zeller & Hildebrandt, 2002).

Table 3: **One task and one submission in full:** `claude-opus-5` at `high` effort on `lang01-cmark-c-to-rust`. All 4,184 checks across 15 modules pass, the highest mark Stage II can give; seven of the eight criteria pass, and the eighth, `rust-is-primary`, fails unanimously because the Rust reproduces the C implementation layout field by field—*"the same hand-managed pointer stack, the same names and the same order . . . this is the original transliterated into unsafe Rust"*, cited to `src/inlines.rs:55` and neighbouring lines. Final score: 0.

Stage I — migration criteria

Criterion	What it asks	Required	Verdict
`no-c-sources`	has C left the build	✓	pass
`no-foreign-headers`	is only `cmark.h` left	✓	pass
`no-c-in-build`	is no C compiled	✓	pass
`rust-present`	is Rust really there	✓	pass
`rust-is-primary`	is Rust the implementation	✓	**fail**
`no-embedded-reference`	no prebuilt artifact linked	✓	pass
`no-verifier-awareness`	was the scorer let off	✓	pass
`default-path`	is the scored path default	✓	pass

All eight criteria are required, so one failure zeroes the run. Each is decided by majority over 3 samples at `medium` effort—here all eight were unanimous.

Any criterion fails ⇒ score 0; Stage III never runs

Stage II — test modules

Module	What it measures	Pass rate
`build`	two link configurations	1.00
`structure`	install tree, ABI	1.00
`conformance`	CommonMark corpus	1.00
`renderers`	five output renderers	1.00
`data-tables`	entities, case folding	1.00
`encoding`	UTF-8 validation	1.00
`robustness`	pathological input	1.00
`options`	options and combinations	1.00
`streaming`	incremental parser	1.00
`tree`	AST and its invariants	1.00
`iterators`	iterator API	1.00
`provenance`	what built these artifacts	1.00
`cli`	the installed executable	1.00
4,184 checks, none skipped	*a perfect Stage II*	**1.00**

For lang01, fifteen directions are allowed and eleven forbidden, the latter including internal struct layout, behaviour the original gets wrong as well, and—most worth stating explicitly—the identity of the tree under test: a candidate program that detects which tree it is running against satisfies every mechanical condition for a break while establishing nothing about the migration, and is rejected on discovery rather than weighed.

Make the task verify itself. No reference solution ships with a task; in its place there is an identity run at image build time—State A is scored against itself inside the same image, and unless every behavioural module reaches 1.0 the build is refused.

Table 3 shows one whole task as the scorer sees it: lang01's eight criteria and the fifteen modules of Stage II. The run in the table is precisely the sharpest form of disagreement between the stages: all fifteen modules at 1.00, the highest mark the behavioural side can give, zeroed by Stage I on a single criterion—every behavioural instrument in the stack calls this migration complete, and the one instrument that reads the code calls it a transliteration.

B Per-Task Breakdown

Table 4 takes the funnel of Section 3.2 apart task by task: of the 26 runs, how many completed the migration, how many passed every fixed check, how many of those fall into blindness, and how many were finally accepted.

Of the 20 tasks, 13 were never solved by any model, and for seven of them not a single submission even reached the verifiers—and they fail in two different ways. On `lang03`, `lang04`, and `pf02`, no run passed every fixed check. On `lang01`, `fw01`, `fw02`, and `fw07`, by contrast, some runs passed every fixed check, but every such run was rejected at Stage I as blindness. A behavioural instrument would report these four tasks as solved and the three-stage protocol reports them as unsolved, which is exactly the disagreement this benchmark was built to expose.

C Detailed Task Catalogue

Table 5 summarises the task set by debt class and Table 6 gives all 20 tasks one by one.

Table 4: **The** 520 **runs broken down by task.** Every task is run once by each of the 26 configurations ($n = 26$). Column meanings follow the metric definitions of Section 3.1: *migrated* is the number of runs passing Migration Audit, *all tests pass* is the number passing every fixed check in Behavioural Tests (*regardless* of the Stage I verdict), *blindness* is the subset of those that failed Stage I, *accepted* is the number passing all three stages, and *score* is defined in (4). *All tests pass − blindness* is therefore the number of runs that reached Stage III. **Only** 7 **of the** 20 **tasks ever produced an accepted submission**.

			Migrated ↑ Stage I passed	**All tests pass ↑** Stage II perfect	**Blindness ↓** passed, not migrated	**Accepted ↑** all three passed	**Score ↑** out of 100
Task	Migration	n	n	n	n	n	mean
Language rewrites							
`lang01`	C → Rust	26	6	5	**5**	0	0.00
`lang02`	C → Java	26	18	4	0	0	12.69
`lang03`	Python → Go	26	14	0	0	0	0.00
`lang04`	JavaScript → Rust	26	20	0	0	0	0.00
`lang05`	Go → Zig	26	18	4	0	**1**	12.69
`lang06`	C++ → C#	26	10	3	**2**	0	2.31
`lang07`	JavaScript → TypeScript	26	14	5	**2**	**3**	11.54
Framework rewrites							
`fw01`	Flask → Starlette	26	24	1	**1**	0	0.00
`fw02`	Express → Fastify	26	13	1	**1**	0	0.00
`fw03`	Vue → React	26	26	5	0	**2**	16.92
`fw04`	Gin → chi	26	25	7	**1**	**3**	21.54
`fw05`	actix-web → axum	26	22	2	**1**	0	2.31
`fw06`	`gorilla/mux` → `net/http`	26	13	22	**9**	**9**	43.08
`fw07`	Dropwizard → Spring Boot	26	9	2	**2**	0	0.00
Platform ports							
`pf01`	POSIX → `wasm32-wasi`	26	15	1	0	0	2.31
`pf02`	CommonJS → V8 realm	26	12	0	0	0	0.00
`pf03`	x86-64 → 3 architectures	26	18	20	**4**	**4**	49.23
Build-toolchain rewrites							
`build01`	Autotools → CMake	26	19	2	0	0	4.23
`build02`	Maven → Gradle	26	20	13	0	0	25.38
`build03`	setuptools → Meson	26	24	21	**2**	**6**	64.62
All tasks		**520**	340	118	30	**28**	13.44

Table 5: **Composition of the SWE Refactor Bench task set.** Every task requires a real open-source repository to be migrated in its entirety onto another stack, with its interface and observable behaviour unchanged. *Budget* is the time limit given to the agent; *criteria* are the Stage I migration questions, *modules* are the independent test modules of Stage II, and *checks* are the behavioural cases those modules hold.

Debt class	**Representative migrations**	**Tasks**	**Source LoC**	**Budget (h)**	**Criteria (Stage I)**	**Modules (Stage II)**	**Checks (Stage II)**
Language	C → Rust, C → Java, Go → Zig	7	4.4 k–39.8 k	12–30	60	117	59,771
Framework	Flask → Starlette, Gin → chi, Vue → React	7	0.8 k–94.8 k	6–16	39	79	55,852
Platform	POSIX → `wasm32-wasi`, Node → V8 realm	3	16.0 k–358.0 k	6–10	19	31	6,725
Build toolchain	Autotools → CMake, Maven → Gradle	3	19.0 k–78.8 k	6	18	37	7,770
Total	**20 upstream projects, 20 tasks**	**20**	**867,062**	**262**	**136**	**264**	**130,118**

D Additional Results

This appendix carries the results that Section 3 reports but does not have room to develop: the per-configuration leaderboard, the shapes a rejected submission takes, and how agent capability varies across the four migration categories.

D.1 Full Leaderboard

Table 7 lists all 26 configurations with their behavioural pass rate, composite score and cost.

Table 6: **The task set, task by task.** *LoC* counts newlines in the version-controlled implementation source of State A, excluding tests, vendored trees and documentation, measured the same way for every task rather than transcribed from each project's own description; together these repositories hold 10,594 version-controlled files. B is the agent's time budget in hours. *Criteria* is the number of prompt-form migration criteria, each of them required: a single failure zeroes the submission. *Modules* is the number of independent test modules and *checks* the number of fixed behavioural cases they hold, recorded from a reference build of State A. The size of a test suite is set by how much of the product is mechanically observable, not by the size of the repository. Every task has six verifiers of one hour each. Table 5 summarises these rows by class. Each row ships an upstream release verbatim, at the version shown and under its own licence: cmark (MacFarlane, 2024), zlib (Gailly & Adler, 2024), sqlparse (Albrecht, 2024), Acorn (The Acorn Contributors, 2024), go-yaml (Canonical Ltd., 2022), Jsonnet (Google Inc., 2023), JSONata (IBM Corp., 2026), httpbin (Reitz, 2024), json-server (typicode, 2023), the RealWorld Conduit app (The RealWorld Contributors, 2019), ChartMuseum (The Helm Authors, 2022), miniserve (Haase, 2024), go-simple-upload-server (Akizuru, 2026), GraphHopper (GraphHopper GmbH, 2025), SQLite (The SQLite Developers, 2020), Stylus (Automattic, 2024), QuickJS (Bellard & Gordon, 2020), libsodium (Denis, 2024), Gson (Google, 2023) and PyCryptodome (The PyCryptodome Authors, 2024).

Task	Upstream project	Source stack	Target stack	LoC	B (h)	Criteria	Modules	Checks
Language rewrites—the implementation language moves, the artifact does not								
lang01	cmark 0.31.1	C11	Rust 1.90	22,619	30	8	15	4,184
lang02	zlib 1.3.1	C89	Java 17	22,732	20	10	18	4,162
lang03	sqlparse 0.5.3	Python	Go (stdlib)	4,393	15	8	15	8,530
lang04	acorn 8.14.0	JavaScript	Rust 1.90	9,571	30	7	13	16,039
lang05	go-yaml v3.0.1	Go	Zig 0.14.1	11,967	20	8	22	10,271
lang06	jsonnet 0.20.0	C++11	C# / .NET 8	39,842	20	9	19	2,608
lang07	JSONata 2.2.2	JavaScript	TypeScript 5.9	9,247	12	10	15	13,977
Framework rewrites—the language stays; what is replaced is the framework the code is organised around								
fw01	httpbin	Flask (WSGI)	Starlette (ASGI3)	2,455	9	5	11	2,830
fw02	json-server	Express 4	Fastify 5	2,992	9	5	10	6,760
fw03	RealWorld Conduit	Vue 2 + webpack	React 18 + Vite	2,261	11	5	10	21,769
fw04	ChartMuseum	Gin	chi v5 (net/http)	5,401	12	5	14	11,429
fw05	miniserve	actix-web 4	axum 0.7 (tower)	3,484	8	6	14	12,466
fw06	uploadserver	gorilla/mux	net/http ServeMux	803	6	6	9	485
fw07	GraphHopper 11.0	Dropwizard 4	Spring Boot 3.5	94,766	16	7	11	113
Platform ports—the host the code assumes changes								
pf01	SQLite 3.31.1	POSIX (unix + win)	`wasm32-wasi`	358,006	10	6	7	2,668
pf02	Stylus 0.63.0	CommonJS + Node	ESM in a V8 realm	16,012	10	7	12	2,570
pf03	QuickJS	x86-64 host layout	3 architectures, endian-clean	89,429	6	6	12	1,487
Build-toolchain rewrites—what produces the artifact changes, and the package is the observable								
build01	libsodium 1.0.20	Autotools	CMake 3.20+	73,228	6	6	15	4,720
build02	Gson 2.10.1	Maven	Gradle (offline)	19,015	6	6	10	2,669
build03	pycryptodome 3.20.0	setuptools	Meson	78,839	6	6	12	381
20 tasks				**867,062**	**262**	**136**	**264**	**130,118**

D.2 What a Rejected Submission Looks Like

A submission that fails Migration Audit can fail it in several different ways, and the difference matters: two of the four shapes in Figure 5 pass every behavioural check by construction, which is precisely why the audit cannot be folded into the test suite.

D.3 Agent Capability Across Migration Categories

Agents exhibit distinct capability profiles across the four migration categories (Table 9). Their overall scores rank build toolchain rewrites first (31.4), followed by platform ports (17.2), framework rewrites (12.0), and language rewrites (5.6). However, **agents do not perform best on one category throughout the pipeline**. They achieve their highest Stage I and Stage II pass rates on build toolchain rewrites (80.8% and 54.0%), but their lowest Stage III survival rate on the same category (17.6%). On framework rewrites, agents achieve only 18.9% at Stage II but their highest Stage III survival rate, 56.0%.

Table 7: **Behavioural pass rate, score and cost for the same runs.** *Behavioural pass rate* is the row mean of the fraction of Behavioural Tests fixed checks passed (a perfect score is not required), *score* is defined in (4), and *cost* is the average API spend per task. *Harness* identifies the coding-agent client: GPT-series models use Codex, and all others use Claude Code. Shading encodes only the magnitude of a value within its column; each model's best-scoring row is marked with a shaded background.

Model	Harness	Effort	Behavioural pass ↑	Score ↑	Cost ↓
			%	out of 100	USD / task
Claude Opus 5	Claude Code	low	92.5	20.5	34.8
Claude Opus 5	Claude Code	medium	92.7	28.5	38.4
Claude Opus 5	Claude Code	high	92.8	34.5	55.7
Claude Opus 5	Claude Code	**xhigh**	92.8	**47.0**	74.9
Claude Opus 5	Claude Code	max	91.9	31.0	72.4
GPT-5.6 Sol	Codex	none	63.6	4.0	2.9
GPT-5.6 Sol	Codex	low	62.0	7.0	5.9
GPT-5.6 Sol	Codex	medium	70.8	6.5	6.0
GPT-5.6 Sol	Codex	high	74.0	19.0	7.7
GPT-5.6 Sol	Codex	xhigh	73.5	9.5	19.1
GPT-5.6 Sol	Codex	**max**	84.1	**28.5**	143.5
Kimi K3	Claude Code	**max**	93.9	**19.5**	28.9
Claude Sonnet 5	Claude Code	low	79.2	4.0	4.4
Claude Sonnet 5	Claude Code	**medium**	73.9	**15.0**	11.9
Claude Sonnet 5	Claude Code	high	85.6	6.0	24.6
Claude Sonnet 5	Claude Code	xhigh	76.5	9.0	27.0
Claude Sonnet 5	Claude Code	max	84.3	8.5	27.5
GPT-5.6 Luna	Codex	none	66.4	4.0	1.6
GPT-5.6 Luna	Codex	low	58.3	4.0	1.7
GPT-5.6 Luna	Codex	medium	66.6	0.0	1.7
GPT-5.6 Luna	Codex	high	75.6	4.0	1.8
GPT-5.6 Luna	Codex	xhigh	83.8	5.5	2.9
GPT-5.6 Luna	Codex	**max**	89.1	**10.5**	2.8
Qwen 3.8 Max	Claude Code	**max**	74.7	**10.0**	14.5
DeepSeek V4 Flash	Claude Code	**max**	90.7	**7.0**	4.3
GLM 5.2	Claude Code	**max**	85.2	**6.5**	17.5

Table 8: **Summary by debt class, pooling all** 26 **configurations.** Column meanings follow the metric definitions of Section 3.1; shading follows Table 7. **Stage I pass rate and acceptance count do not move together**: build toolchain is the easiest class to get past Stage I and the one that loses most at Stage III; framework rewrites contribute 14 of the 28 acceptances; language rewrites score lowest.

			Migrated ↑		All tests pass ↑	Accepted ↑	Broken ↓	Blindness ↓	Score ↑
Class	Tasks	Runs	n	%	n	n	%	n	mean
Build toolchain	3	78	63	80.8	36	6	82.4	2	**31.4**
Platform port	3	78	45	57.7	21	4	76.5	4	**17.2**
Framework rewrite	7	182	132	72.5	40	14	44.0	15	**12.0**
Language rewrite	7	182	100	54.9	21	4	66.7	9	**5.6**

Agents encounter different bottlenecks across migration categories. Their lowest conditional pass rate occurs at Stage III for build toolchain rewrites and platform ports (17.6% and 23.5%), but at Stage II for framework and language rewrites (18.9% and 12.0%). These profiles align with the repository surface that agents must modify. In build toolchain rewrites and platform ports, agents

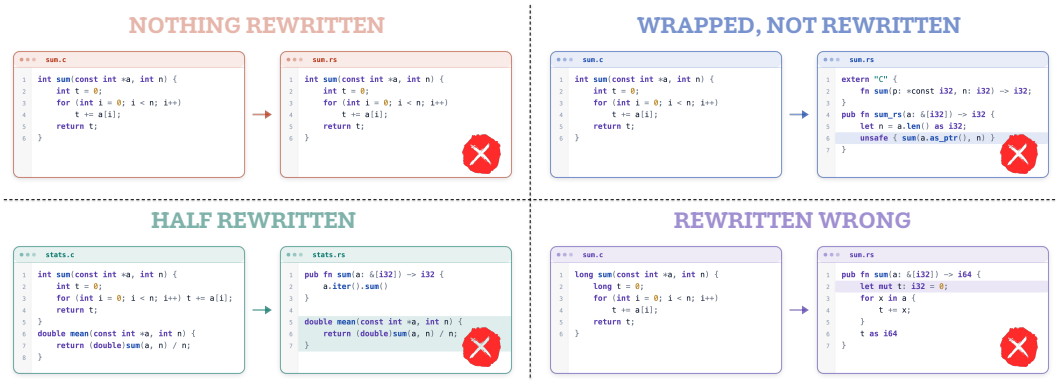

Figure 5: **The four kinds of submission Stage I has to tell apart.** Top left, *nothing rewritten*: the new file is the C copied over verbatim, only the suffix changed. Top right, *wrapped*: the Rust side is an `extern "C"` forwarding shim and the original C still does the work. Bottom left, *half done*: some functions are genuinely rewritten while others still call back into the old implementation. Bottom right, *rewritten wrongly*: really rewritten, but the behaviour changed. The first two pass every behavioural test, and only an instrument that reads mechanism can reject them.

Table 9: **Agent success varies across stages and migration categories.** Every cell is the share of the runs that *entered* that stage and passed it, so the three multiply out to the category's final acceptance rate. The Stage II column is taken over the runs that cleared Stage I (a blindness run passes every fixed test but is already out at Stage I, so it does not count); the Stage III column is taken over the runs that cleared both earlier stages and actually met the verifiers. **Agents do not perform best on one category throughout the pipeline**: they achieve their highest pass rates on build toolchain rewrites at the first two stages but their lowest at the last; framework rewrites show the reverse pattern.

		Migration Audit ↑		**Behavioural Tests ↑**		**Agentic Verification ↑**		**Final ↑**	
Category	Runs	n	%	n	%	n	%	Acc.	Score
Build toolchain	78	63/78	80.8	34/63	54.0	6/34	17.6	6	**31.4**
Platform port	78	45/78	57.7	17/45	37.8	4/17	23.5	4	**17.2**
Framework	182	132/182	72.5	25/132	18.9	14/25	56.0	14	**12.0**
Language	182	100/182	54.9	12/100	12.0	4/12	33.3	4	**5.6**
All	520	340/520	65.4	88/340	25.9	28/88	31.8	28	13.4

mainly change how the repository is built or which host it runs on while leaving the product code largely unchanged. Their submissions pass Stage II relatively often, but Stage III exposes behavioural differences outside the fixed suite. In framework and language rewrites, agents modify the product code itself, and the fixed suite filters their submissions at Stage II before most reach the verifiers. Language rewrites show the sharpest early attrition: only 12 of 182 runs reach Stage III.

E Analysis of Benchmark Validity

The premise of SWE Refactor Bench is a claim about an *instrument* rather than about models, so this section examines the instrument itself: whether the Stage I judge is stable (Section E.1), what the Stage II and Stage III tests each ask (Section E.2), and whether all six verifiers are needed (Section E.3).

E.1 Stage I Decisions Agree Across Models and Humans

Does the same judge model read the same tree differently three times? The judge is `gpt-5.6-sol`; every criterion is judged by three independent samples of it and the majority is taken. Of 3,536 criterion verdicts, 3,405 (96.3%) had all three samples agree, and only 131 split 2:1. That looks like very little disagreement, but Stage I requires *every* criterion to pass, so a single disagreement

Table 10: **Task-by-Task Comparison ofMigration Audit and Independent Human Annotation.** Each row is one task, attempted once by every one of the 26 model–effort configurations. *Human* and *judge* are how many of those runs each side called a genuine migration; *agree*, *too strict* and *too lenient* sort the same runs by whether the two verdicts matched, the judge zeroed a run the human accepted, or the reverse.

			Called a real migration		**Judge against human**		
Task	Class	runs	by human	by judge	agree ↑	too strict ↓	too lenient ↓
lang01 (cmark)	Language	26	7	6	25	1	0
lang03 (sqlparse)	Language	26	15	14	23	2	1
fw02 (json-server)	Framework	26	15	13	24	2	0
fw06 (uploadserver)	Framework	26	15	13	22	3	1
pf02 (Stylus)	Platform	26	14	12	24	2	0
build01 (libsodium)	Toolchain	26	23	19	22	4	0
6 tasks		**156**	**89**	**77**	**140**	**14**	**2**

on a single criterion can decide the fate of a whole run: of the 340 runs that passed Stage I, 35 (10.3%) had at least one criterion that passed only 2:1—on that criterion, one sample argued for zero. Which means that without majority voting, letting any single dissenting sample count, the number passing Stage I would be 305 rather than 340. Majority over three samples is load-bearing here, not ceremonial.

Would a human judge differently? We asked two software-engineering researchers not involved in this work to decide independently, without seeing the judge model's verdicts, whether "this is a genuine migration" for all 156 runs of 6 tasks spanning the four migration classes, three tasks each, working from the same criteria the judge saw. Judge and human agree 89.7% of the time ($140/156$, $\kappa = 0.795$), and the direction of the disagreements matters more than their number: of the 16 disagreements, 14 are the judge being too strict—the human considers the migration genuine and the judge scored zero—and only 2 go the other way; Table 10 breaks this down by task. Following all 16 through the later stages changes almost nothing about the final verdict: of the 14 too-strict cases, 12 would have failed the full fixed suite at Stage II anyway; of the 2 too-lenient cases, 1 likewise; and the remaining one reached Stage III, where four of the six agentic verifiers each constructed a counterexample against it, so it was not accepted either. That is, the Stage I error let no submission a human thought was disguised into the accepted set; what it may have undercounted is at most the 2 runs that passed every behavioural check and were zeroed by Stage I alone. The judge's error has a direction, and that direction makes points harder to earn, not easier.

Does the judge spare its own family? It is itself one of the evaluated systems, so leniency towards submissions from the same family would quietly inflate them. That does not appear: `gpt-5.6-sol` passed 57.1% of the 240 submissions written by a GPT-series model against 72.5% of the other 280. Most of that gap is submission quality rather than provenance.

E.2 Agentic Verification Finds Failures Beyond Fixed Tests

Stage II and Stage III test two different parts of the same thing, and the clearest way to see it is to put two sets of cases side by side.

What Stage II asks is what the task author could think of in advance. The check from Section 3.3.2—whether visiting `/` settles at `/#/`—is an observation you get simply by running the original once, which is why it sits in the fixed suite as one of 21,769 checks. Coverage of that kind is already wide: `fw03` alone carries more than twenty-one thousand checks.

What Stage III asks cannot be anticipated. On `fw04` (ChartMuseum, Gin → chi), `POST /api/charts` has to dispatch between a multipart-form upload handler and a raw-body upload handler; the original truncates `Content-Type` at the first space *or* semicolon, while the migrated version truncates only at the semicolon, so a request with one extra space before the boundary reaches a different handler on each side—an implementation detail of Gin whose existence the task author did not know of when writing the suite. Others of the same kind: `lang05` (go-yaml, Go → Zig), where go-yaml, by way of Go's `time.Parse`, also accepts a comma as the decimal separator,

so `2001-12-14T21:59:43,10Z` is a `!!timestamp` on the original and a `!!str` in the Zig version; and `pf01` (SQLite, POSIX → WASI), where after checkpointing a WAL database and switching it back to DELETE mode, native SQLite removes both the `-wal` and the `-shm` file while the WASI version removes only `-wal`.

What the two kinds have in common is that each is decisive—if this one check does not pass, the migration is not complete; the only difference is that one can be fixed in advance as a test and the other cannot. That is exactly why both stages are indispensable, and exactly why Stage III has to hand in an *executable* program rather than a verdict: only an executable counterexample turns "a problem nobody thought of" into a fact that can be re-checked.

E.3 Six Verifiers: Strength and Diversity Both Matter

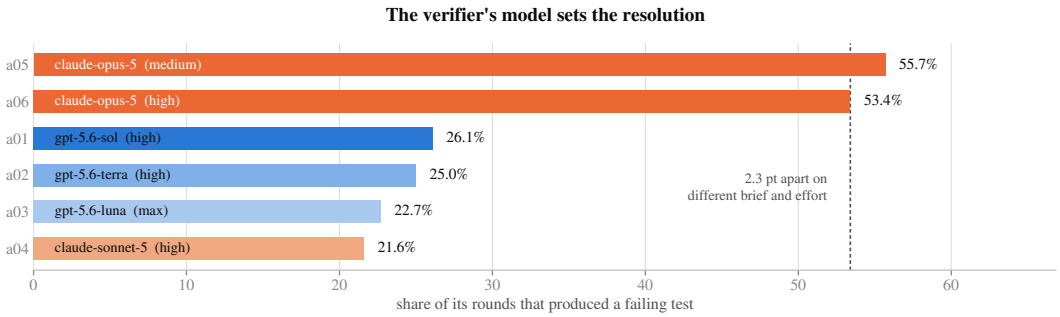

Figure 6: **Break rate of each of the six verifiers**: the share of its rounds in which it handed in an executable counterexample that passes on the original and fails on the submission, coloured by the model behind it and annotated with its reasoning-effort setting. The dashed line marks the lower of the two `claude-opus-5` verifiers.

Finding 1: a strong coding agent as the verifier matters an order of magnitude more than its configuration. The six verifiers face the *same* 88 submissions and each runs 88 rounds, so task difficulty is identical for them; all that differs is which direction each was assigned. The break rates, however, are far apart (Figure 6): the two held by `claude-opus-5` are 55.7% and 53.4%, the other four between 21.6% and 26.1%. That gap cannot be attributed to opus happening to draw the easy directions, for two reasons. First, of the two opus verifiers, one probes an assigned direction and the other is unrestricted, so their prompts and effort settings *both* differ, and yet their break rates differ by only 2.3 percentage points. Second, the *unrestricted* one was given no direction at all—it was free to search the same ground as the other four—and it still broke 53.4%. Changing prompt and effort within one model moves the break rate by about two points; changing the model moves it by thirty. This also means the reported numbers are a property of this panel as much as of the submissions: retire the two strongest and the remaining four would accept 46 submissions instead of 28. An accepted submission is therefore not a migration proven correct but a migration that survived the strongest adversaries we could field—and as models improve, this same fixed set of 20 tasks will be scored more strictly.

Nor are the six redundant. Counting *exclusive* breaks—where this verifier alone found a counterexample and the other five did not—the two opus verifiers have 5 and 2, and the four weaker ones 4 between them. Even the one with the lowest break rate rejected a submission that the other five let through: they differ in strength, but they are complementary in *coverage*.

Finding 2: models do not spare their own family's work. This matters, because `claude-opus-5` both leads the leaderboard and holds two of the verifiers. Pooled, the effect runs *opposite* to collusion: verifiers broke 33.9% of the submissions written by a model of their own family and 34.3% of everyone else's. The apparent counter-evidence is that the opus verifiers break 40.8% of opus-authored submissions against 65.0% of everyone else's; but that fails its own control—the four non-opus verifiers on the same submissions are at 17.1% against 29.1%, a drop *by the same factor*. What separates the two is submission quality, not collusion: submissions written by opus are simply harder to break, for everyone.