From Dead Code and Static Requirements to Working Engines: Software Revival with Coding Agents

Anonymous authors
Paper under double-blind review

Abstract

Can coding agents restore software that no longer runs while preserving its underlying methods, and reconstruct industrial software engines from open specifications? Here we introduce ReviveBench, a benchmark with two task families evaluated by hidden verifiers calibrated against native execution environments, established engineering tools, or purpose-built reference implementations. The revival family comprises ten tasks involving dependency incompatibilities, deleted core modules, legacy builds, and a GPU-based foundation model. Every starting workspace fails verification, and the strongest evaluated model passes all ten tasks in at least one run each. In contamination-control experiments, identifier obfuscation reduces line similarity to the original implementations from 0.51–0.96 to 0.03–0.44 without reducing the observed pass rate of any evaluated model. On repositories created after the stated knowledge cutoffs, the strongest model passes eight of nine runs. The reconstruction family comprises thirteen tasks spanning numerical, geometric, hardware, and transactional systems (e.g. CAD and CRM). Two models meet the benchmark's pass criteria on all thirteen, although our audit shows that the CFD task cannot establish numerical-solver capability. Benchmark construction and auditing uncover 28 verifier defects, including 24 false negatives and two false positives. These findings show that executable verification can itself introduce substantial measurement error. We present three practical checks: test whether prescribed methods can reach the grading thresholds, investigate agreement among independently generated candidates, and recompute diagnostics from submitted artifacts. ReviveBench thus provides both an evaluation of software revival and engine reconstruction, and cases in validating the verifiers used to measure coding agents for software design.

1 Introduction

Useful software can be inaccessible for different reasons. An older scientific package may no longer install because its dependencies removed required APIs or its build system is incompatible with the current environment. Industrial capabilities may instead be available only through closed-source engines, even when public standards describe parts of their behavior. We study two complementary tasks: *software revival*, which restores an existing implementation, and *clean-room reconstruction*, which implements a specified engine without access to its source.

Repository-level benchmarks such as SWE-bench ask models to resolve issues in existing codebases (Jimenez et al., 2023); systems such as SWE-agent provide interfaces for this work (Yang et al., 2024). Revival and reconstruction add complementary requirements. A revived package must reproduce what the original computed, and a reconstructed engine has externally defined, checkable semantics: a netlist either is or is not equivalent to its RTL; a ledger either balances or does not. We exploit that property with verifiers the agent never sees, calibrated against the software's native environment or a reference system. Our reconstruction tasks target bounded core functionality specified in public documents. The

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.

agent receives a specification and three worked examples and is prohibited from installing the existing implementations listed for that task.

The suite combines three forms of checking: numerical tolerances, exact or formally proved properties, and transactional invariants. Its construction also exposes a measurement problem: we logged 28 verifier defects, including 24 false negatives that rejected correct work and two false positives that accepted incorrect work. These defect counts describe the audit history; they are not directly comparable to agent failure rates. We report the observed failure modes, the signals that exposed them, and practical validation checks. This complements prior audits of test adequacy and benchmark quality (Liu et al., 2023a; Aleithan et al., 2024; Liu et al., 2026a) with evidence from constructing and auditing hidden grading programs as well as softwares.

Contributions.

- **ReviveBench**: ten software-revival tasks with contamination controls and thirteen clean-room engine-reconstruction tasks, graded by hidden verifiers (Section 3, Figure 1).
- **Revival results**: the strongest model passes all ten revival tasks, including a 2019 C++ build and a GPU foundation model, and obfuscated and post-cutoff controls provide evidence of capability beyond verbatim recall (Section 5).
- An **eight-model evaluation** across three vendors: two models meet the pass criteria on all thirteen reconstruction tasks; a low-cost model passes only two, both of which are passed by all eight models and therefore do not distinguish this cohort (Section 6).
- A **verifier-defect taxonomy** with 28 documented cases, including two false positives, one of which led us to withdraw a claim from an earlier version of this work (Section 8).
- **Three reusable methods**: the *achievability probe*, *consensus among candidates*, and *never grade a self-reported quantity* (Section 8).

2 Related work

Code-generation and agentic software-engineering benchmarks. Function-level suites established execution-based evaluation for code models (Chen et al., 2021; Austin et al., 2021). Later benchmarks evaluate class-level generation and complex library use (Du et al., 2023; Zhuo et al., 2024), while LiveCodeBench collects recent competition problems (Jain et al., 2024). At the repository level, SWE-bench evaluates patches for real GitHub issues (Jimenez et al., 2023). SWE-agent provides an agent–computer interface for such work (Yang et al., 2024), and SWE-Gym and SWE-smith support training software-engineering agents (Pan et al., 2024; Yang et al., 2025). Adjacent benchmarks address different scientific workflows: MLE-bench evaluates machine-learning engineering through Kaggle competitions, ScienceAgentBench evaluates programs for data-driven scientific tasks, and CORE-Bench evaluates computational reproducibility (Chan et al., 2024; Chen et al., 2024; Siegel et al., 2024; Liu et al., 2026b). Commit0 is particularly close to our reconstruction setting: agents implement libraries from API specifications with access to interactive unit tests and execution feedback (Zhao et al., 2024b). Our reconstruction tasks instead withhold their graders and combine public engineering specifications with reference tools, certified values, and purpose-built reference implementations.

Specification-to-implementation in engineering domains. VerilogEval and RTLLM evaluate hardware-description generation using simulation and other correctness or design-quality checks (Liu et al., 2023b; Lu et al., 2023). RealBench extends evaluation to real-world IP designs with testbenches and a formal checker (Jin et al., 2025). In industrial control, LLM4PLC and Agents4PLC combine PLC program generation with verification (Fakih et al., 2024; Liu et al., 2024). Our RTL and PLC tasks ask agents to implement a synthesizer or a scan-cycle runtime. We evaluate the resulting tools using netlist equivalence checks and

reference execution traces, respectively. This changes the target from a design expressed in a language to the software that processes that language.

Benchmark quality, contamination and memorization. EvalPlus shows that additional tests expose incorrect generated programs and can change model rankings (Liu et al., 2023a). Analyses of code benchmarks also examine limitations in task quality, difficulty, and the interpretation of recurring model failures (Dai et al., 2024; Sharifloo et al., 2025). SWE-Bench+ identifies solution leakage and weak tests (Aleithan et al., 2024), while the SWE-Bench Illusion study probes memorization through file-path identification and reference-function reproduction (Liang et al., 2025). Work on contamination motivates benchmark-specific measurement (Sainz et al., 2023; Zhou et al., 2023), recent-task collection (Jain et al., 2024), and controlled dataset construction (Zhao et al., 2024a). Our revival experiments use identifier obfuscation and post-cutoff repositories to probe related concerns. Our verifier audit complements this work by examining grading programs during benchmark construction: it identifies 28 defects, including 24 false negatives and two false positives. Contamination controls and verifier validation address distinct risks; a task can be unseen by a model and still be graded incorrectly.

Imperfect verifiers, reward hacking and specification gaming. Stroebl et al. (2024) show that false positives in imperfect verifiers can limit the accuracy attainable through resampling. Related studies examine reward hacking under RLVR (Helff et al., 2026), gaps between visible-test and held-out performance in long-horizon coding (Zhao et al., 2026), and specification gaming in reasoning models (Nishimura-Gasparian et al., 2026). Ray (2026) studies adversarial completions that expose disagreements between buggy and stricter reference verifiers. We focus on defects encountered while constructing and auditing a hidden-verifier suite, where 24 of the 28 documented defects reject correct work. Such defects can arise without deliberate exploitation by an agent. Our achievability probe checks whether the method prescribed by the specification can attain the required threshold. It complements fuzzing by testing consistency between the specification, reference calculation, and grading criteria.

Model-based grading. Model judges support evaluation of open-ended outputs (Zheng et al., 2023; Gu et al., 2024), but their agreement with human judgments and evaluation consistency depend on design choices (Yamauchi et al., 2025). Our suite uses executable checks throughout during the verification steps. The observed defects show that executable grading also requires validation.

Software revival and dependency rot. Vangala et al. (2025) report gaps between declared and required dependencies in agent-generated projects, while CORE-Bench studies the reproduction of published computations (Siegel et al., 2024). Our revival tasks extend these concerns to older packages, missing modules, and builds that must run in newer environments while preserving reference behavior.

3 The ReviveBench suite

Software revival. Ten tasks restore nonfunctional scientific software (Table 1). Four are broken by the ecosystem around them or by missing code: two single-cell packages must run under TensorFlow 2.16, Keras 3 and NumPy 2, which removed the APIs they call; two delete a package's core module and ask the agent to rebuild it from the paper, the README and the remaining code, one of them with every identifier renamed. Three must be built and reproduced end to end: PyMOL 2.3, a C++ extension whose 2019 build targets Python 3.7 and distutils, under Python 3.12; QuPath 0.2.3, a Java 14 platform, with JDK 21 and a current Gradle; and Caduceus, a DNA foundation model whose Mamba GPU kernels must work with current PyTorch. Three are drawn from repositories created after the models' knowledge cutoff, with their most-imported module deleted and their own test suite hidden and used as the verifier. Verifiers compare against the software's native environment: PyMOL must reproduce reference atom counts, distances, secondary structure

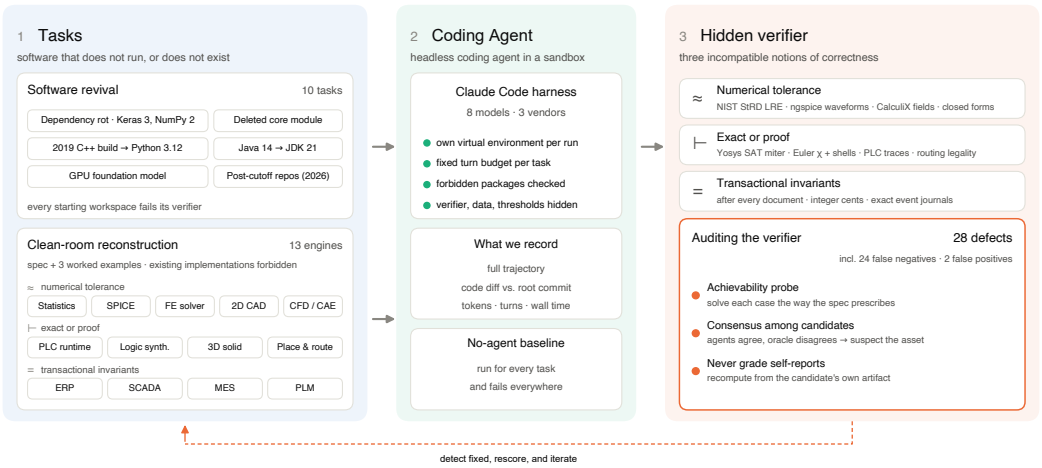


Figure 1: ReviveBench. Agents receive software that does not run or does not exist (left), work headlessly in an isolated sandbox (middle), and are graded by hidden verifiers that use three forms of correctness checking (right). Auditing the verifiers identified 28 defects; a fixed defect triggers re-scoring if the specification is unchanged and a re-run if the specification gained information.

and alignment RMSD and render headlessly; QuPath must reproduce the official release's cell detection on a test slide.

Clean-room reconstruction. Thirteen tasks (Figure 2a) specify bounded engine functionality using public standards and task-specific requirements. Reference checks draw on NIST StRD certified values, ngspice waveforms, CalculiX displacement fields, `ezdxf`+`shapely` geometry, Yosys SAT equivalence, OpenCASCADE analytic mass properties, and reference implementations we wrote for the transactional engines. The agent sees a specification, three worked examples, and an environment in which prohibited implementation libraries are absent and checked for.

Three notions of correctness. Five engines are graded by *numerical tolerance* (relative RMS, displacement error, log relative error). Four are graded *exactly or by proof*: a candidate netlist is combined with its source RTL in a miter circuit and checked using SAT: a complete proof over all input assignments for combinational designs, temporal induction for sequential ones; a candidate solid must match the reference Euler characteristic and shell count exactly, and classify 40 margin-sampled query points per model without a single error; a PLC runtime must reproduce the reference output traces cell for cell; and a placement must be legal with every net connected. These describe the individual checks; task-level pass thresholds are distinct from requiring every check to pass (Table 2). Four are graded by *transactional invariants* with no tolerance whatsoever: amounts are integers in minor currency units, invariants are checked after every document, and a discrepancy of one cent is a failure.

Hidden verifiers and anti-cheat. The agent cannot read the verifier, its hidden data, or its thresholds. The verifier additionally checks that no forbidden implementation is installed, that dependency versions were not downgraded, and records whether the agent modified upstream tests or reached outside its workspace. Every run executes in its own virtual environment.

4 Experimental setup

All runs execute on a Slurm cluster, one job per run, in isolated environments. Agents are driven headlessly by Claude Code with a fixed turn budget per task, identical across

Table 1: Software revival. Each cell reports successful runs divided by all runs recorded for that model and task. Every starting workspace fails its verifier. The models are the same Fable 5.1, Sonnet 5 and Haiku 4.5 used in the reconstruction suite, reached through Anthropic's own endpoint or, for eight later runs, Amazon Bedrock.

Category	Software	What is broken	Fable 5.1	Sonnet 5	Haiku 4.5
Dependency rot	deepimpute	TF 2.16 / Keras 3 / NumPy 2	2/3	1/1	1/1
	DCA	removed Keras and TF-1 APIs	3/4	1/1	0/1
Deleted core module	deepimpute	core module deleted	3/3	1/1	0/1
	deepimpute (obf.)	same, identifiers renamed	3/3	2/2	1/1
Build and reproduce	PyMOL 2.3	2019 C++ build, Python 3.12	2/6	0/1	0/1
	QuPath 0.2.3	Java 14 build, JDK 21	3/4	1/1	1/1
	Caduceus	GPU Mamba kernels, torch $\geq$2.5	2/2	1/1	0/1
Post-cutoff (2026)	binderranker	module deleted, tests hidden	2/3	0/1	0/1
	interelate	module deleted, tests hidden	3/3	1/1	1/1
	sirna_data	module deleted, tests hidden	3/3	1/1	0/1
Tasks restored			**10/10**	**8/10**	**4/10**

models, and a wall-clock limit. The revival family was evaluated on Fable 5.1, Sonnet 5 and Haiku 4.5 through Anthropic's own endpoint, with eight later runs through Amazon Bedrock; the reconstruction suite on eight models from three vendors: the same three models and Opus 5 through Amazon Bedrock; three GPT-5.6 variants (sol, luna, terra); and GLM-5.3 flash. The GPT-5.6 variants and GLM-5.3 flash expose only an OpenAI-style API and are driven through a local protocol adapter (Section 9). We report the *protocol* condition throughout. A no-agent baseline is run for every task and fails everywhere, confirming that the starting workspace does not already pass.

Excluding GLM-5.3 flash, the suite comprises 215 runs, of which 155 are valid; 37 terminated in gateway or quota errors and are excluded and counted separately, and 6 exhausted their turn budget. Total real spend is $610.70. GLM-5.3 flash adds 33 runs, of which 13 enter the results (Section 9).

5 Software revival results

The strongest model passes all ten revival tasks. Fable 5.1 passes all ten revival tasks in at least one run each (Table 1); Sonnet 5 restores eight and Haiku 4.5 four; both fail the 2019 PyMOL build, Sonnet 5 after exhausting its 120-turn budget. The hardest builds succeed with verifiable fidelity. PyMOL 2.3 builds under Python 3.12 with NumPy 2.5, reproduces the reference atom counts, Cα distances, secondary-structure assignment and alignment RMSD, and renders headlessly. QuPath 0.2.3, rebuilt with a current JDK, detects 3,842 cells on the test slide in agreement with the official release. Caduceus loads its pretrained weights under PyTorch 2.11 with a reverse-complement equivariance error of exactly zero and a masked-language modeling accuracy of 0.75, against 0.00 for a random initialization. Therefore, our conclusion provides meaningful suggestions for software design.

Contamination controls. These packages predate model knowledge cutoffs, so success could reflect memorized fixes. Motivated by concerns about benchmark contamination (Sainz et al., 2023) and the use of recent tasks in LiveCodeBench (Jain et al., 2024), we examine similarity, identifier obfuscation, and repository recency (Figure 2b). *Similarity*: normalized line-level similarity between a rebuilt module and the deleted original ranges from 0.51 to 0.96 on the original task; Sonnet 5's 0.96 reproduces internal variable names and print strings, a strong signal consistent with recall. *Obfuscation*: with package, class, function, argument and file names renamed and paper references removed, similarity falls to 0.03–0.44 while no model's observed pass rate falls. This reduces evidence of verbatim reconstruction but does not rule out prior exposure to the underlying methods. *Recency*: on repositories created after the cutoff, Fable 5.1 passes 8 of 9 runs. For the weaker models the comparison is inconclusive: Sonnet 5 passes 2 of 3 post-cutoff runs against 5 of 6 pre-cutoff, and Haiku 4.5 1 of 3 against 2 of 6, cohorts too small and too different in task mix to separate

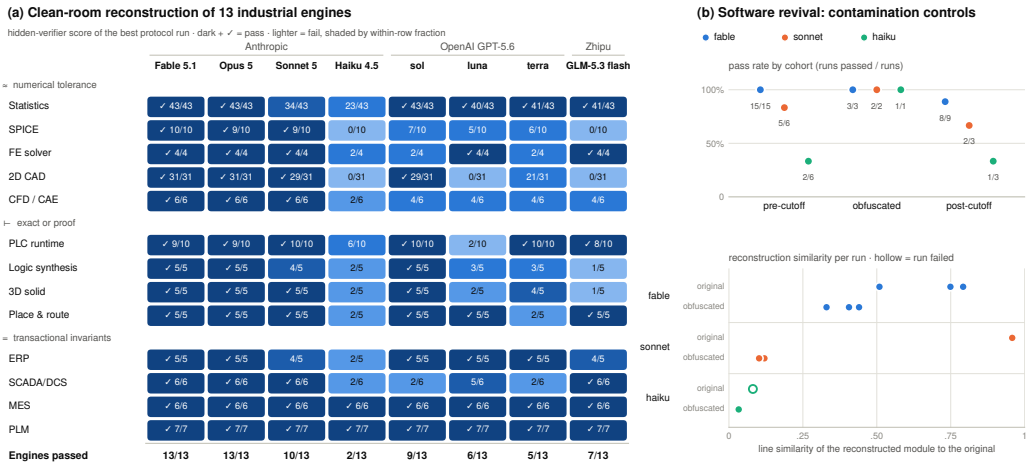


Figure 2: Results. (a) Clean-room reconstruction of thirteen industrial engines by eight models; each cell is the hidden-verifier score of the best protocol run and is comparable within a row only (full table in Appendix A). (b) Software-revival contamination controls: pass rate per cohort (top) and line similarity of each rebuilt module to the deleted original, for the original and the identifier-obfuscated task (bottom; hollow markers are failed runs).

capability from recall; Sonnet 5's verbatim reconstruction is the clearer recall signal. New revival benchmarks should prefer post-cutoff repositories and ship an obfuscated control.

6 Reconstruction results

Two models meet all thirteen task-level pass criteria. Fable 5.1 and Opus 5 meet the pass criteria on all thirteen reconstruction tasks; Sonnet 5 passes ten. These totals include the CFD task, whose limitations are discussed in Section 8. Fable 5.1 averages 20.8 turns per engine against 74.1 for Opus 5, writing complete modules per turn rather than incremental edits, and cost does not track capability: Sonnet 5 clears ten engines for $35 where Opus 5 spends $76 for thirteen (Table 3).

A low-cost model as a difficulty probe. Haiku 4.5 passes only two of thirteen, at roughly $1 per run, and its failures are associated with identifiable implementation errors: a reversed mesh normal that yields a volume of −598.60 against a reference of 598.997, a ledger that reports `Ledger out of balance`. We use it as a low-cost probe of task difficulty. Passing this model alone does not establish that a task is uninformative. In this suite, however, the two tasks it passes, MES and PLM, are the two on which all eight models score full marks, and we report them as category coverage rather than as capability findings.

Cross-vendor comparison. The three GPT-5.6 variants pass 9, 6 and 5 of thirteen tasks, respectively. Their circuit-simulator failures affect different subsets of netlists, with only two lightly damped resonant circuits shared by all three failure sets. Such differences motivate inspecting candidate implementations as well as shared benchmark assets (Section 8); they do not, by themselves, establish the cause of failure. GLM-5.3 flash passes seven engines, including SCADA/DCS, which all three GPT-5.6 variants fail, but no numerical-tolerance engine other than statistics and finite elements. Its runs are dominated by long reasoning: single turns reach the 64K-token output cap and take up to half an hour, and four of its thirteen runs end at the three-hour wall-clock limit.

7 Beyond single engines

A multi-module toolchain. To test whether the approach survives decomposition, Fable 5.1 built *plcforge*, a six-module IEC 61131-3 toolchain (compiler front end, scan-cycle runtime, ladder front end, PLCopen XML, Modbus TCP server, historian), in three phases: architecture with machine-checkable interface contracts (12 turns, $6.02), implementation (81 turns, $20.56, 7,520 lines of Python), and repair from CI-style symptoms only (80 turns, $12.50). Acceptance includes a differential fuzzer that compares execution traces of random well-typed programs with a reference interpreter; the final system passes all nine checks, including 298 of 300 fuzzed programs.

Engines as tools. The reconstructed engines are usable downstream. Given a natural-language mechanical requirement, Fable 5.1 chose every dimension and drove the 2D CAD kernel that an agent had built earlier to produce each part; a hidden checker measures the resulting geometry. All three parts pass every check (8/8, 7/7 and 9/9 on bounding box, hole count and position, minimum wall thickness, symmetry, fillet radii and connectivity) in 4 to 9 turns and under $1.02 each.

Turn budget versus capability. Budgets can decide outcomes. In two headroom probes run before the output-cap correction of Section 9, doubling GLM-5.3 flash's turn budget turned a circuit-simulator failure (0/10 netlists at 100 turns) into a pass at 200 turns, whereas its logic synthesizer still produced no legal netlist after three hours at twice the budget. The first result shows that additional turns helped under that harness condition; the second remained unresolved within the tested budget and time limit. We keep budgets identical across models in the matrix and report such probes separately.

8 The verifier is the bottleneck

Constructing and auditing this suite identified **28 verifier defects, including 24 false negatives and 2 false positives**. In all but one case, we initially attributed the discrepancy to the model; review instead identified a problem in the benchmark asset. The asymmetry matters: false negatives cost credit, whereas false positives can limit the accuracy attainable through resampling (Stroebl et al., 2024). We group them and give the signal that exposed each group.

Unreachable thresholds, caught by an achievability probe. Calibration, which runs the reference implementation through the verifier, checks agreement with that reference on the evaluated cases. It cannot show that a threshold is reachable by the route the specification prescribes, particularly when reference outputs and grading formulas share assumptions. This concern is related to the verifier bugs that fuzzing uncovers before training (Ray, 2026). We therefore added an *achievability probe*: solve every case the way the specification tells the agent to, and record the error actually attained. On first application it found four tolerances that the prescribed methods did not meet, and caused one case to be deleted outright rather than ship a threshold we could not demonstrate was reachable.

Undefined quantities, caught by candidate consensus. When N separately generated implementations agree with each other and disagree with the oracle, investigate the benchmark asset. Three models produced bit-identical foreign-currency figures contradicting our reference; the reference carried a scale factor the specification never defined. The signal can be obtained from existing runs by checking whether failing candidates fail *in the same way*; it prompted three investigations. We record one important refinement: consensus can flag a problem in the specification, data, or grading path, as well as in the reference value. Shared model errors remain another possible explanation. In one case four candidates agreed on a number that differed from the reference, and the reference was right: the generator repaired a self-intersecting polygon before measuring its area and the grading path did not, so two different area conventions were in use. The rule is to check that the generating and grading paths compute the same quantity before declaring a reference wrong.

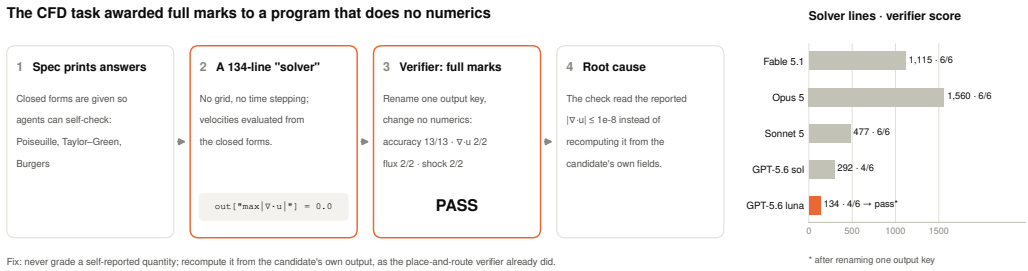


Figure 3: Case study: the CFD verifier graded a self-reported number. A 134-line program with no grid evaluates the closed forms printed in the specification and hard-codes the divergence diagnostic to zero; renaming one output key, with no numerical change, turns it into a full pass. Right: solver lines (excluding tests) of the run behind each model's matrix cell, with that cell's score.

False positives. Two defects admitted incorrect work. First, three criteria were written as "passes $\geq 0.9\times$ cases"; with 13 designs this let 12/13 count as full marks, so a netlist with a *proven* counterexample passed. Percentage tolerances suit numerical agreement and are wrong for criteria where a failure is a proof of error. Second, and more seriously, the CFD task awarded full marks to a 134-line program with no grid at all, which evaluates the closed-form solutions the specification itself prints and hard-codes the divergence diagnostic to zero (Figure 3). We verified this by copying that workspace, changing *one output key name* and no numerical logic, and re-running the hidden verifier: every check passed, including all three "exact" ones. The root cause generalizes: **the verifier graded a self-reported diagnostic instead of recomputing it from the candidate's own artifact**. We had done this correctly in place-and-route, where the reported wirelength must match a recomputation from the emitted geometry, but omitted that safeguard here.

This retracts a claim from an earlier version of this work, which held that the exact checks demonstrated the candidates had written genuine conservative schemes. Inspection of the solver packages shows that the three strongest models *did* write real discretizations (1,115, 1,560 and 477 lines of solver code excluding tests, each constructing its own grids and none hard-coding the divergence on the Taylor–Green case where it is graded), but the task never *required* it, so those scores are not evidence of solver capability. The supported interpretation is that this task tests whether an agent can deliver a program that works against a specified interface, not whether it can solve partial differential equations. This vulnerability is related to specification gaming (Zhao et al., 2026; Nishimura-Gasparian et al., 2026), but our audit does not establish agent intent. The three strongest candidates contained discretizations; the minimal passing counterexample demonstrates a limitation of the verifier.

Oracles that grade themselves. Where reference answers are generated by the reference implementation, calibration compares the oracle with itself and agrees perfectly, so a semantic error can survive indefinitely. One such error, a scan-delay convention that disagreed with the specification's own worked example, was found only when three LLMs produced byte-identical logs one scan earlier than the oracle. The remedy is a hand-derived self-test committed alongside the asset; expectations copied from the implementation test nothing. This error plays an important role.

9 Threats to validity

Sample size and thresholds. Most cells contain one or two seeds, and the reported matrix selects the best eligible run. Pass thresholds are chosen by us and anchored to reference measurements. These results describe performance under the reported conditions; they do not establish stable model rankings, pass probabilities, or robustness to alternative thresholds. Unequal numbers of attempts may also affect best-run comparisons.

Cost is not comparable across vendors here. The GPT-5.6 variants and GLM-5.3 flash are only exposed in a different API dialect, so they are driven through a local protocol adapter and launched under a model name the harness recognizes. The harness consequently prices them at the launch model's rate; that figure is an artifact and we withhold it. Token counts pass through unmodified and are genuine. For the GPT-5.6 variants the adapter also strips one protocol flag that the upstream endpoint rejects, which changes the tool set offered to those models and therefore limits comparability across experimental conditions.

GLM-5.3 flash required two harness corrections. Its first runs ended without producing any code. Its longest turns take up to half an hour; the adapter emitted nothing until the upstream call returned, and the agent harness abandons a request after about six minutes with no bytes, or about twelve minutes with keep-alive events but no content. We fixed this by streaming events while waiting and raising the harness's stream-idle timeout, which changes nothing the model sees. Second, the adapter had written the output cap into a request field that this provider accepts but silently ignores, so the early runs had no output cap at all; we verified that the corrected field is honored before re-running, using the same 64K-token cap the adapter passes to the GPT-5.6 variants. Nine of GLM's thirteen results come from these re-runs; the other four come from earlier runs whose total output over the whole run was at most 64K tokens and therefore cannot contain a turn over the cap. Of its 33 runs, 13 were infrastructure failures, 5 are excluded for the uncapped condition, 2 were headroom probes and 13 are reported. Two non-streaming auxiliary requests in the re-runs still carried no cap; they write no code but consume wall time. Wall-clock limits were not uniform across our batch scripts, but no run of any other model reached its limit (the longest took 2.4 hours), whereas four of GLM's thirteen reported runs reached their three-hour limit.

Turn counts. Turn counts are unreliable for runs ending in an API error or at the wall-clock limit; comparisons of effort use wall time and cost.

Two tasks do not discriminate. MES and PLM are passed by every model including the difficulty probe, and are reported as coverage. The CFD task does not discriminate transcription from solving and must not be cited as evidence of numerical capability.

10 Conclusion

ReviveBench evaluates software revival and bounded engine reconstruction across diverse domains. The strongest evaluated model passes all ten revival tasks in at least one run each, and the contamination controls provide evidence of capability beyond verbatim reconstruction. Two models meet the pass criteria on all thirteen reconstruction tasks, but those totals must be interpreted alongside the CFD verifier's inability to establish numerical-solver capability and the limited discriminative value of MES and PLM. The audit identifies 28 verifier defects, including two false positives, and a provider-specific harness error that silently removed the intended output cap. Reliable evaluation therefore requires validation of both grading logic and execution conditions. Useful checks include achievability probes, independently derived self-tests, diagnostics recomputed from submitted artifacts, and confirmation that requested settings are actually enforced.

Use of Large Language Models

We used an LLM-based coding assistant throughout this project and describe its use here in the terms ICLR's policy asks for. We revised our codes as well as the manuscript with the help of LLMs. We did not use LLMs to create fake citation or non-existing support.

Ethic statement

The users are solely responsible for the content they generate with models in this solution, and there are no mechanisms in place for addressing harmful, unfaithful, biased, and toxic content disclosure. Any modifications of the models should be released under different

version numbers to keep track of the original models related to this manuscript. The target of current solution only serves for academic research. The users cannot use it for other purposes. Finally, we are not responsible for any effects of the use of the model.

Reproducibility statement

Task definitions, hidden verifiers, oracle references and complete per-run trajectories are archived, as are the per-run results table, the model matrix and the ablation data used for every table in this paper. Every number in the tables and figures is generated programmatically from those files, by scripts released with the benchmark.

Codes of this project can be found in this repo: `https://anonymous.4open.science/r/ReviveBench-A469/`.

References

Reem Aleithan, Haoran Xue, Mohammad Mahdi Mohajer, Elijah Nnorom, Gias Uddin, and Song Wang. SWE-Bench+: enhanced coding benchmark for LLMs. *arXiv preprint arXiv:2410.06992*, 2024. URL `https://arxiv.org/abs/2410.06992`.

Jacob Austin, Augustus Odena, Maxwell Nye, Maarten Bosma, Henryk Michalewski, David Dohan, Ellen Jiang, Carrie Cai, Michael Terry, Quoc Le, and Charles Sutton. Program synthesis with large language models. *arXiv preprint arXiv:2108.07732*, 2021. URL `https://arxiv.org/abs/2108.07732`.

Jun Shern Chan, Neil Chowdhury, Oliver Jaffe, James Aung, Dane Sherburn, Evan Mays, Giulio Starace, Kevin Liu, Leon Maksin, Tejal Patwardhan, Lilian Weng, and Aleksander Mądry. MLE-bench: evaluating machine learning agents on machine learning engineering. *arXiv preprint arXiv:2410.07095*, 2024. URL `https://arxiv.org/abs/2410.07095`.

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. URL `https://arxiv.org/abs/2107.03374`.

Ziru Chen, Shijie Chen, Yuting Ning, Qianheng Zhang, Boshi Wang, Botao Yu, Yifei Li, Zeyi Liao, Chen Wei, Zitong Lu, Vishal Dey, Mingyi Xue, Frazier N. Baker, Benjamin Burns, Daniel Adu-Ampratwum, Xuhui Huang, Xia Ning, Song Gao, Yu Su, and Huan Sun. ScienceAgentBench: toward rigorous assessment of language agents for Data-Driven scientific discovery. *arXiv preprint arXiv:2410.05080*, 2024. URL `https://arxiv.org/abs/2410.05080`.

Jianbo Dai, Jianqiao Lu, Yunlong Feng, Guangtao Zeng, Rongju Ruan, Ming Cheng, Dong Huang, Haochen Tan, and Zhijiang Guo. MHPP: exploring the capabilities and limitations of language models beyond basic code generation. *arXiv preprint arXiv:2405.11430*, 2024. URL `https://arxiv.org/abs/2405.11430`.

Xueying Du, Mingwei Liu, Kaixin Wang, Hanlin Wang, Junwei Liu, Yixuan Chen, Jiayi Feng, Chaofeng Sha, Xin Peng, and Yiling Lou. ClassEval: A Manually-Crafted benchmark for evaluating LLMs on class-level code generation. *arXiv preprint arXiv:2308.01861*, 2023. URL `https://arxiv.org/abs/2308.01861`.

Mohamad Fakih, Rahul Dharmaji, Yasamin Moghaddas, Gustavo Quiros Araya, Oluwatosin Ogundare, and Mohammad Abdullah Al Faruque. LLM4PLC: harnessing large language models for verifiable programming of PLCs in industrial control systems. *arXiv preprint arXiv:2401.05443*, 2024. URL https://arxiv.org/abs/2401.05443.

Jiawei Gu, Xuhui Jiang, Zhichao Shi, Hexiang Tan, Xuehao Zhai, Chengjin Xu, Wei Li, Yinghan Shen, Shengjie Ma, Honghao Liu, Saizhuo Wang, Kun Zhang, Yuanzhuo Wang, Wen Gao, Lionel Ni, and Jian Guo. A survey on LLM-as-a-Judge. *arXiv preprint arXiv:2411.15594*, 2024. URL https://arxiv.org/abs/2411.15594.

Lukas Helff, Quentin Delfosse, David Steinmann, Ruben Härle, Hikaru Shindo, Patrick Schramowski, Wolfgang Stammer, Kristian Kersting, and Felix Friedrich. LLMs gaming verifiers: RLVR can lead to reward hacking. *arXiv preprint arXiv:2604.15149*, 2026. URL https://arxiv.org/abs/2604.15149.

Naman Jain, King Han, Alex Gu, Wen-Ding Li, Fanjia Yan, Tianjun Zhang, Sida Wang, Armando Solar-Lezama, Koushik Sen, and Ion Stoica. LiveCodeBench: holistic and contamination free evaluation of large language models for code. *arXiv preprint arXiv:2403.07974*, 2024. 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? *arXiv preprint arXiv:2310.06770*, 2023. URL https://arxiv.org/abs/2310.06770.

Pengwei Jin, Di Huang, Chongxiao Li, Shuyao Cheng, Yang Zhao, Xinyao Zheng, Jiaguo Zhu, Shuyi Xing, Bohan Dou, Rui Zhang, Zidong Du, Qi Guo, and Xing Hu. RealBench: benchmarking verilog generation models with Real-World IP designs. *arXiv preprint arXiv:2507.16200*, 2025. URL https://arxiv.org/abs/2507.16200.

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. URL https://arxiv.org/abs/2506.12286.

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. *arXiv preprint arXiv:2305.01210*, 2023a. URL https://arxiv.org/abs/2305.01210.

Mingjie Liu, Nathaniel Pinckney, Brucek Khailany, and Haoxing Ren. VerilogEval: evaluating large language models for verilog code generation. *arXiv preprint arXiv:2309.07544*, 2023b. URL https://arxiv.org/abs/2309.07544.

Tianyu Liu, Dingyuan Dai, Weihao Xuan, Jialin Chen, Qingcheng Zeng, Rui Yang, Ada Fang, Zhen Yang, Peng Cui, Yinpeng Dong, Yuanqi Du, Marinka Zitnik, James Zou, and Jie Tang. Towards an agentic era of ai. *SSRN Electronic Journal*, 2026a. doi: 10.2139/ssrn.7245280.

Tianyu Liu, Allen Xin Wang, Antonia Panescu, Lisa Xinyi Chen, Wenxin Long, Xinyu Wei, Yueqian Jing, Ziyao Zeng, Jihang Chen, Sihan Jiang, et al. Benchmarking ai agents for addressing scientific challenges across scales. *arXiv preprint arXiv:2606.12736*, 2026b.

Zihan Liu, Ruinan Zeng, Dongxia Wang, Gengyun Peng, Jingyi Wang, Qiang Liu, Peiyu Liu, and Wenhai Wang. Agents4PLC: automating closed-loop PLC code generation and verification in industrial control systems using LLM-based agents. *arXiv preprint arXiv:2410.14209*, 2024. URL https://arxiv.org/abs/2410.14209.

Yao Lu, Shang Liu, Qijun Zhang, and Zhiyao Xie. RTLLM: an Open-Source benchmark for design RTL generation with large language model. *arXiv preprint arXiv:2308.05345*, 2023. URL https://arxiv.org/abs/2308.05345.

Kei Nishimura-Gasparian, Robert McCarthy, and David Lindner. Towards understanding specification gaming in reasoning models. *arXiv preprint arXiv:2605.02269*, 2026. URL https://arxiv.org/abs/2605.02269.

Jiayi Pan, Xingyao Wang, Graham Neubig, Navdeep Jaitly, Heng Ji, Alane Suhr, and Yizhe Zhang. Training software engineering agents and verifiers with SWE-Gym. *arXiv preprint arXiv:2412.21139*, 2024. URL https://arxiv.org/abs/2412.21139.

Jaideep Ray. Before the model learns the Bug:Fuzzing RLVR verifiers. *arXiv preprint arXiv:2606.01066*, 2026. URL https://arxiv.org/abs/2606.01066.

Oscar Sainz, Jon Ander Campos, Iker García-Ferrero, Julen Etxaniz, Oier Lopez de Lacalle, and Eneko Agirre. NLP evaluation in trouble: On the need to measure LLM data contamination for each benchmark. *arXiv preprint arXiv:2310.18018*, 2023. URL https://arxiv.org/abs/2310.18018.

Amir Molzam Sharifloo, Maedeh Heydari, Parsa Kazerooni, Daniel Maninger, and Mira Mezini. Where do LLMs still struggle? an In-Depth analysis of code generation benchmarks. *arXiv preprint arXiv:2511.04355*, 2025. URL https://arxiv.org/abs/2511.04355.

Zachary S. Siegel, Sayash Kapoor, Nitya Nadgir, Benedikt Stroebl, and Arvind Narayanan. CORE-Bench: fostering the credibility of published research through a computational reproducibility agent benchmark. *arXiv preprint arXiv:2409.11363*, 2024. URL https://arxiv.org/abs/2409.11363.

Benedikt Stroebl, Sayash Kapoor, and Arvind Narayanan. The limits of inference scaling through resampling. *arXiv preprint arXiv:2411.17501*, 2024. URL https://arxiv.org/abs/2411.17501.

Bhanu Prakash Vangala, Ali Adibifar, Ashish Gehani, and Tanu Malik. AI-Generated code is not reproducible (yet): An empirical study of dependency gaps in LLM-Based coding agents. *arXiv preprint arXiv:2512.22387*, 2025. URL https://arxiv.org/abs/2512.22387.

Yusuke Yamauchi, Taro Yano, and Masafumi Oyamada. An empirical study of LLM-as-a-Judge: how design choices impact evaluation reliability. *arXiv preprint arXiv:2506.13639*, 2025. URL https://arxiv.org/abs/2506.13639.

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. *arXiv preprint arXiv:2405.15793*, 2024. URL https://arxiv.org/abs/2405.15793.

John Yang, Kilian Lieret, Carlos E. Jimenez, Alexander Wettig, Kabir Khandpur, Yanzhe Zhang, Binyuan Hui, Ofir Press, Ludwig Schmidt, and Diyi Yang. SWE-smith: scaling data for software engineering agents. *arXiv preprint arXiv:2504.21798*, 2025. URL https://arxiv.org/abs/2504.21798.

Bingchen Zhao, Dhruv Srikanth, Yuxiang Wu, and Zhengyao Jiang. SpecBench: measuring reward hacking in Long-Horizon coding agents. *arXiv preprint arXiv:2605.21384*, 2026. URL https://arxiv.org/abs/2605.21384.

Qihao Zhao, Yangyu Huang, Tengchao Lv, Lei Cui, Qinzheng Sun, Shaoguang Mao, Xin Zhang, Ying Xin, Qiufeng Yin, Scarlett Li, and Furu Wei. MMLU-CF: A contamination-free multi-task language understanding benchmark. *arXiv preprint arXiv:2412.15194*, 2024a. URL https://arxiv.org/abs/2412.15194.

Wenting Zhao, Nan Jiang, Celine Lee, Justin T Chiu, Claire Cardie, Matthias Gallé, and Alexander M Rush. Commit0: library generation from scratch. *arXiv preprint arXiv:2412.01769*, 2024b. URL https://arxiv.org/abs/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. *arXiv preprint arXiv:2306.05685*, 2023. URL https://arxiv.org/abs/2306.05685.

Kun Zhou, Yutao Zhu, Zhipeng Chen, Wentong Chen, Wayne Xin Zhao, Xu Chen, Yankai Lin, Ji-Rong Wen, and Jiawei Han. Don't make your LLM an evaluation benchmark cheater. *arXiv preprint arXiv:2311.01964*, 2023. URL https://arxiv.org/abs/2311.01964.

Terry Yue Zhuo, Minh Chien Vu, Jenny Chim, Han Hu, Wenhao Yu, Ratnadira Widyasari, Imam Nur Bani Yusuf, Haolan Zhan, Junda He, Indraneil Paul, Simon Brunner, Chen Gong, Thong Hoang, Armel Randy Zebaze, Xiaoheng Hong, Wen-Ding Li, Jean Kaddour, Ming Xu, Zhihan Zhang, Prateek Yadav, Naman Jain, Alex Gu, Zhoujun Cheng, Jiawei Liu, Qian Liu, Zijian Wang, Binyuan Hui, Niklas Muennighoff, David Lo, Daniel Fried, Xiaoning Du, Harm de Vries, and Leandro Von Werra. BigCodeBench: benchmarking code generation with diverse function calls and complex instructions. *arXiv preprint arXiv:2406.15877*, 2024. URL https://arxiv.org/abs/2406.15877.

A Full results tables

Table 2: Clean-room reconstruction of thirteen industrial software engines (the data behind Figure 2a). Each cell is the hidden-verifier score of the best run under the *protocol* condition; scores are comparable *within* a row only; ✓ denotes a pass. GLM-5.3 flash cells use the runs described in Section 9. Pass marks denote task-level thresholds, not necessarily perfect scores. CFD scores measure interface compliance and do not establish numerical-solver capability (Section 8); MES and PLM do not distinguish the evaluated models.

		Anthropic				OpenAI GPT-5.6			Zhipu
Engine	Commercial analog	Fable 5.1	Opus 5	Sonnet 5	Haiku 4.5	sol	luna	terra	GLM-5.3 flash
Statistics engine	SAS, SPSS	✓ 43/43	✓ 43/43	✗ 34/43	✗ 23/43	✓ 43/43	✓ 40/43	✓ 41/43	✓ 41/43
Circuit simulator	HSPICE	✓ 10/10	✓ 9/10	✓ 9/10	✗ 0/10	✗ 7/10	✗ 5/10	✗ 6/10	✗ 0/10
PLC runtime	TIA Portal	✓ 9/10	✓ 9/10	✓ 10/10	✗ 6/10	✓ 10/10	✗ 2/10	✓ 10/10	✓ 8/10
FE solver	Nastran	✓ 4/4	✓ 4/4	✓ 4/4	✗ 2/4	✗ 2/4	✓ 4/4	✗ 2/4	✓ 4/4
2D CAD kernel	AutoCAD	✓ 31/31	✓ 31/31	✓ 29/31	✗ 0/31	✓ 29/31	✗ 0/31	✗ 21/31	✗ 0/31
Logic synthesizer	Design Compiler	✓ 5/5	✓ 5/5	✗ 4/5	✗ 2/5	✓ 5/5	✗ 3/5	✗ 3/5	✗ 1/5
3D solid modeler	Parasolid	✓ 5/5	✓ 5/5	✓ 5/5	✗ 2/5	✓ 5/5	✗ 2/5	✗ 4/5	✗ 1/5
ERP core	SAP S/4HANA	✓ 5/5	✓ 5/5	✗ 4/5	✗ 2/5	✓ 5/5	✓ 5/5	✓ 5/5	✗ 4/5
Place & route	Innovus	✓ 5/5	✓ 5/5	✓ 5/5	✗ 2/5	✓ 5/5	✓ 5/5	✗ 2/5	✓ 5/5
CFD / CAE	Fluent	✓ 6/6	✓ 6/6	✓ 6/6	✗ 2/6	✗ 4/6	✗ 4/6	✗ 4/6	✗ 4/6
SCADA/DCS runtime	WinCC	✓ 6/6	✓ 6/6	✓ 6/6	✗ 2/6	✗ 2/6	✗ 5/6	✗ 2/6	✓ 6/6
MES	SAP ME	✓ 6/6	✓ 6/6	✓ 6/6	✓ 6/6	✓ 6/6	✓ 6/6	✓ 6/6	✓ 6/6
PLM / BOM	Teamcenter	✓ 7/7	✓ 7/7	✓ 7/7	✓ 7/7	✓ 7/7	✓ 7/7	✓ 7/7	✓ 7/7
Engines passed		**13/13**	**13/13**	**10/13**	**2/13**	**9/13**	**6/13**	**5/13**	**7/13**

Table 3: Per-model totals on the reconstruction suite. "Engines passed" is taken from Table 2; runs, turns and cost are over all valid runs of that model.

Model	Engines passed	Valid runs	Mean turns	Cost (USD)
Fable 5.1	13/13	13	20.8	95
Opus 5	13/13	9	74.1	76
Sonnet 5	10/13	13	60.1	35
Haiku 4.5	2/13	15	80.3	15
sol	9/13	13	30.2	n/a[†]
luna	6/13	13	35.3	n/a[†]
terra	5/13	13	31.0	n/a[†]
GLM-5.3 flash	7/13	13[‡]	83.1[‡]	n/a[†]

[†] Run through a local protocol adapter, so the harness prices them at the launch model's rate; that figure is an artifact and is withheld. Token counts are genuine. [‡] The thirteen runs used in the matrix; four hit the wall-clock limit and record no turn count, so the mean is over the other nine.

B Reasoning-effort ablation

We varied the agent's reasoning-effort setting across its five levels on two tasks, after first confirming on the wire that the setting reaches the API and that all five levels produce distinct requests. Table 4 shows unchanged scores at all five levels. Cost, turns, and wall time increase monotonically for ERP, but not for CAD. On CAD, the highest setting costs 2.3× as much as the lowest and uses 2.2× as many output tokens, despite intermediate fluctuations.

We do not read this as "effort does not matter". Post-hoc analysis shows the model already achieves full sub-scores on both tasks at the default setting, which leaves no room for higher scores at that setting. The unchanged scores therefore provide limited evidence about the value of additional reasoning effort. The second arm was chosen *because* it appeared to have headroom (29/31), and the two missing operations turned out to be two of our own verifier defects; once fixed, all five levels score 31/31. This is the same rule as our difficulty probe, pointed at the other end of the ability range: **saturated tasks have limited power to reveal improvements in accuracy.** They can still reveal differences in resource use. Here, the ablation also had diagnostic value: identical failures across five settings prompted investigation of the same two verifier defects.

Table 4: Reasoning-effort ablation (Opus 5, five levels). Scores remain unchanged. Cost, turns, and wall time increase monotonically for ERP but fluctuate at intermediate settings for CAD. Both tasks are saturated at all five settings, limiting conclusions about improvements in accuracy (Appendix B).

Task	Effort	Score	Turns	Wall (s)	Cost (USD)
ERP core	low	5/5	9	204	1.00
	medium	5/5	18	411	1.85
	high	5/5	20	672	2.71
	xhigh	5/5	30	1070	4.37
	max	5/5	31	1464	4.79
2D CAD kernel	low	31/31	65	1426	6.95
	medium	31/31	64	2242	8.09
	high	31/31	46	1914	7.09
	xhigh	31/31	101	3117	13.36
	max	31/31	121	4588	15.93