ZERO2REPO: CAN CODING AGENTS BUILD REPOSITORIES FROM SCRATCH?

Anonymous authors
Paper under double-blind review

ABSTRACT

Coding agents are increasingly asked to build software rather than patch it, yet benchmarks for from-scratch repository construction are mostly limited to a single language and depend on manually curated tasks. We introduce Zero2Repo, a benchmark in which an agent receives a product requirements document, an interface contract, and an empty workspace, and must deliver a complete repository in the project's native ecosystem. Tasks are produced by a language-agnostic authoring pipeline that converts real, version-pinned open-source projects into behavioral specifications, reproducible environments, and hidden acceptance tests. Each task is validated by execution: a reference implementation derived from the upstream project must pass, and adversarial validation must show that the tests reject incorrect implementations. Evaluation runs production coding agents in isolated containers, withholds the acceptance tests until an explicit submission, and assigns a binary reward only when every test passes, with no LLM judge. The pipeline and harness make no language-specific assumptions and apply to mainstream programming ecosystems; the current release contains Python, TypeScript, Go, and C++ tasks. Even on 11 tasks drawn from repositories that frontier models have very likely seen during training, the strongest agent solves only 10, and every failing submission passes 89–99% of the hidden tests; for the two strongest agents, 87–100% of failed tests trace to a single omission or a low-frequency rule stated in the specification rather than to a missing subsystem, so each failure is a concrete target for improvement.

1 INTRODUCTION

Writing a correct function and delivering a working repository test different abilities. To deliver a repository, an agent must turn a product-level description into an architecture, public interfaces, dependencies, and a build configuration, then find and fix integration failures from terminal output over a long session. Widely used evaluations do not require this. They either isolate a short program (Chen et al., 2021; Austin et al., 2021) or start from a mature repository and ask for a local patch (Jimenez et al., 2024), so the architecture, dependency graph, packaging, and build system already exist.

Recent benchmarks move toward from-scratch development but draw the task boundary differently. Commit0 supplies library scaffolds and interactive test feedback (Zhao et al., 2024). NL2Repo-Bench starts from an empty workspace for Python projects and restores reference packaging and tests before evaluation (Ding et al., 2026). DevBench evaluates stages of software development as separate tasks (Li et al., 2024), RepoGenesis targets Python and Java web microservices (Peng et al., 2026), and RepoZero frames repository generation as cross-language behavioral reproduction from API specifications (Zhang et al., 2026). None of these settings asks whether an agent can deliver a complete repository in its native ecosystem, including its own packaging and build artifacts, without seeing the acceptance tests while it works. Answering that question requires the agent to own the entire deliverable, the acceptance tests to stay sealed, reuse of the upstream implementation to be blocked, and specifications, environments, and tests to remain consistent across ecosystems.

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.

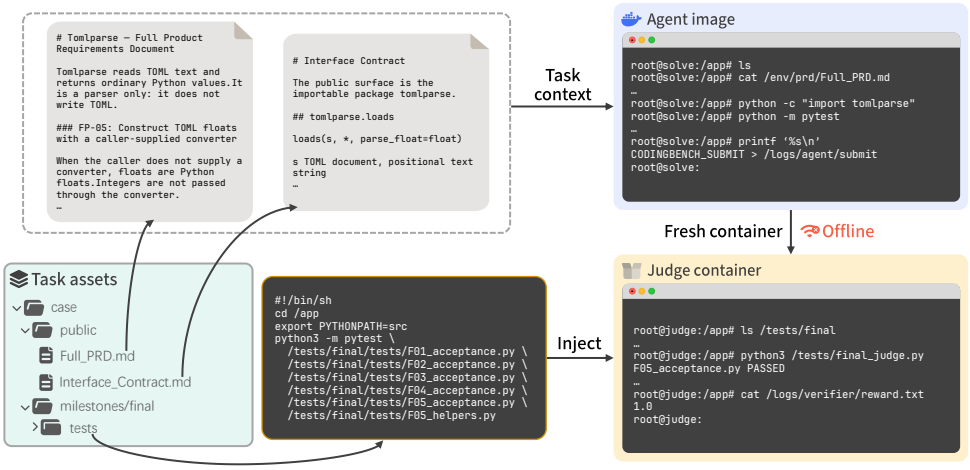

Figure 1: Data flow for one Zero2Repo task (Tomlparse). The PRD and Interface Contract from the task assets are given to the agent as task context in the agent image, where it builds the repository in `/app` and submits by writing the submit file. The workspace is then copied into a fresh, offline judge container, the hidden acceptance tests are injected, and the judge writes a binary reward.

Zero2Repo is built around this requirement.[1] Each task provides a product requirements document (PRD), an Interface Contract, a reproducible toolchain, and an empty workspace. The PRD describes the required behavior; the Interface Contract fixes the observable API and command-line surface and leaves architecture and implementation to the agent. The agent creates the repository, installs dependencies, builds the software when required, and submits the workspace. A task is solved only if every hidden acceptance test passes, and no LLM judge contributes to the score. Figure 1 traces one task through this process.

The current release contains 11 tasks, eight in Python and one each in TypeScript, Go, and C++, covering parsers, protocols, command-line tools, security utilities, and software that depends on external resources. We plan to extend the benchmark to 30–50 tasks with broader language coverage for the v1 release.

Tasks are derived from real, version-pinned open-source projects by an automated pipeline in which executable evidence decides what is released. Language models read code and draft artifacts; each requirement must correspond to runnable behavior of the pinned project, each environment must rebuild from its recipe, and each test suite must pass a reference implementation and survive adversarial validation. Public names and repository identifiers are neutralized before release. During evaluation, the agent's image contains no hidden tests, code-hosting sites are unreachable, and installing or importing the upstream implementation is blocked. After an explicit submission, the workspace is copied into a fresh judge container where the hidden tests are injected and run. The judge neither repairs the repository nor supplies missing components.

We evaluate three frontier model–harness configurations. The first 11 tasks were deliberately drawn from well-known projects that models have very likely encountered in training. Even so, no configuration solves every task, and failing submissions pass most hidden tests: the strongest agents fail on long-tail rules and single inconsistencies that break a group of tests, rather than on missing subsystems. Our contributions are:

- **A spec-to-repository benchmark in native ecosystems.** Agents deliver complete repositories from an empty workspace; acceptance is sealed, deterministic, and all-or-nothing. The design is language-agnostic, and the current release covers four languages.
- **An evidence-gated construction method with little human effort.** Real pinned repositories are turned into neutralized specifications, reproducible environments, and implementation-

[1] Anonymized code, released tasks, and experiment scripts: `https://anonymous.4open.science/r/Zero2Repo-3AEC/README.md`

Table 1: Benchmarks closest to Zero2Repo.

Benchmark	Initial artifact	Required deliverable	Languages	Task construction
Long-horizon agent tasks				
SWE-Together	Repository and first user turn	Repository changes	Mixed	109 tasks from recorded sessions
TerminalWorld	Container and instruction	Completed terminal task	Mixed	Derived from command recordings
FrontierSWE	Prepared environment	Open-ended solution	Mixed	17 expert-curated tasks
From-scratch repository construction				
DevBench	Stage-dependent repository context	Per-stage artifacts	Python, Java, JS, C/C++	22 curated repositories
Commit0	API specification and skeleton	Library implementation	Python	54 curated libraries
NL2Repo-Bench	Specification; empty workspace	Repository implementation	Python	104 annotator-written specifications
RepoGenesis	Requirements and dependency file	Deployable web microservice	Python, Java	106 repositories, 30 evaluated
RepoZero	API specification; empty workspace	Cross-language reproduction	Cross-language	Automated from open-source repositories
Zero2Repo	PRD and Interface Contract; empty workspace	Complete native-ecosystem repository	Mainstream languages	Model-assisted derivation with evidence gates

independent tests. Every task is grounded in the executable behavior of a real project and checked by reference execution and adversarial validation; human review is limited to inspecting the resulting artifacts.

- **A controlled evaluation with interpretable failures.** Under a common submission and judging protocol, the benchmark identifies tasks that remain unsolved even when the source project is likely in the training data, and test-level results show which specified behaviors were missed.
- **Reusable training assets.** The same task assets yield supervised trajectories and reinforcement-learning environments with the hidden suite as a verifiable reward. Test-level results attribute each failure to a specific category, such as a long-tail edge-case rule, a non-functional subsystem, a single omission that breaks a group of tests, or environment-dependent behavior (Table 3), so the assets also indicate which behaviors training should target.

2 Related Work

Local code generation and repository modification. HumanEval, MBPP, LiveCodeBench, and BigCodeBench evaluate short programs (Chen et al., 2021; Austin et al., 2021; Jain et al., 2025; Zhuo et al., 2025); RepoBench and CrossCodeEval add repository context but keep completion as the target (Liu et al., 2024; Ding et al., 2023). The SWE-bench family evaluates issue resolution in existing repositories with the project's own tests (Jimenez et al., 2024; OpenAI, 2024; Zan et al., 2025). In all of these settings the architecture, dependency graph, and build system already exist.

From-scratch software construction. Benchmarks for new repositories differ in their starting artifacts and success criteria (Table 1). DevBench scores individual development stages with executable checks and an LLM judge (Li et al., 2024); Commit0 reconstructs Python libraries from skeletons with interactive test feedback (Zhao et al., 2024); NL2Repo-Bench starts from an annotator-written specification and an empty workspace, but restores reference packaging and tests before running the upstream pytest suite, which limits it to Python (Ding et al., 2026); RepoGenesis targets Python and Java microservices (Peng et al., 2026); RepoZero reformulates generation as cross-language reproduction checked by output equivalence (Zhang et al., 2026). Zero2Repo starts from a behavioral product specification, requires a repository in the project's native ecosystem with its own packaging and build artifacts, seals all acceptance tests during the solve, and awards credit only for a complete pass.

Long-horizon agent evaluation and task construction. Terminal-Bench evaluates agents on difficult command-line tasks in containerized environments, pairing each task with a human-written solution and executable tests (Merrill et al., 2026). FrontierSWE poses open-ended engineering and research problems with 20-hour budgets and hidden graders; its 17 tasks are curated by domain experts, which makes the suite costly to extend (**?**). SWE-Together reconstructs multi-turn tasks

from recorded user sessions and replays user feedback through an LLM simulator; whether a task can be solved therefore depends on the simulator conveying the original intent correctly (**?**). SetUpAgent and TerminalWorld automate task construction from historical execution environments and real command recordings, respectively (Vergopoulos et al., 2025; Chu et al., 2026). Zero2Repo targets repository delivery without prescribing a solution path, and it automatically authors the product specification, public interface, native build environment, and acceptance tests of each task, releasing a task only after reference execution and adversarial validation confirm that its tests are both passable and discriminating.

3 THE ZERO2REPO BENCHMARK

3.1 TASK FORMULATION

A Zero2Repo task is a tuple $(\mathcal{S}, \mathcal{E}, \mathcal{T})$ of a public specification $\mathcal{S}$, an execution environment $\mathcal{E}$, and a hidden acceptance suite $\mathcal{T}$. The agent receives $\mathcal{S}$ and an empty workspace inside $\mathcal{E}$, in which the required toolchain and dependencies are already installed. Its objective is to deliver a working repository that implements the specified behavior and interfaces (Figure 2).

Public specification and implementation freedom. The public specification states what the repository must do and how its functionality is accessed. The PRD groups behavioral requirements into feature points, each covering normal operation, boundary conditions, and failure cases. The Interface Contract fixes the package names, API and CLI entry points, argument conventions, return types, and required error types through which those requirements are tested. The task also declares its language, build command, and hardware requirements. Within these constraints the agent chooses its own architecture, algorithms, and code organization.

Development and deliverable. The agent develops and tests the repository within a wall-clock budget B, without access to the hidden tests, a reference implementation, or acceptance feedback. It submits a workspace W containing the source code, project configuration, and any build artifacts required by the public build command. The submission protocol is described in §3.3.

Success criterion. Each valid submission W is scored by running the hidden acceptance suite $\mathcal{T}$ in environment $\mathcal{E}$:

$$R(W) = \mathbf{1}[W \text{ passes all tests in } \mathcal{T} \text{ under } \mathcal{E}]\,.$$

The benchmark score is the mean of R over tasks (§4.1).

Example requirement. In the Tomlparse task, one PRD requirement is to support a caller-supplied floating-point converter without changing integer parsing. The Interface Contract exposes this capability as `tomlparse.loads(s, *, parse_float=float)`. The acceptance tests pass `Decimal` as the converter and check that parsing `0.1` yields `Decimal("0.1")` without intermediate binary-float rounding, while integer tokens bypass the converter.

3.2 DATASET CONSTRUCTION AND VERIFICATION

Each task is derived from an open-source repository. Language models derive the specification and tests from the repository's source, documentation, and test suite; executable checks and review then decide whether the task is released (Figure 3).

Source selection and specification generation. We select repositories with a working installation procedure, an existing test suite, and functionality reachable through public APIs or CLIs, and pin each to a source revision. The first 11 tasks were chosen deliberately from widely used, long-established projects that are very likely present in model training data. This choice tests whether agents can fully rebuild a repository they have plausibly seen; later releases add recently published repositories and balance the language distribution. From the pinned code, documentation, and tests, language models write the PRD feature points and the corresponding Interface Contract entries. Implementation details are kept only when they define an interface or a build constraint. An environment recipe records the toolchain, dependencies, and installation and build commands; external resources needed at acceptance time are listed with download URLs and checksums.

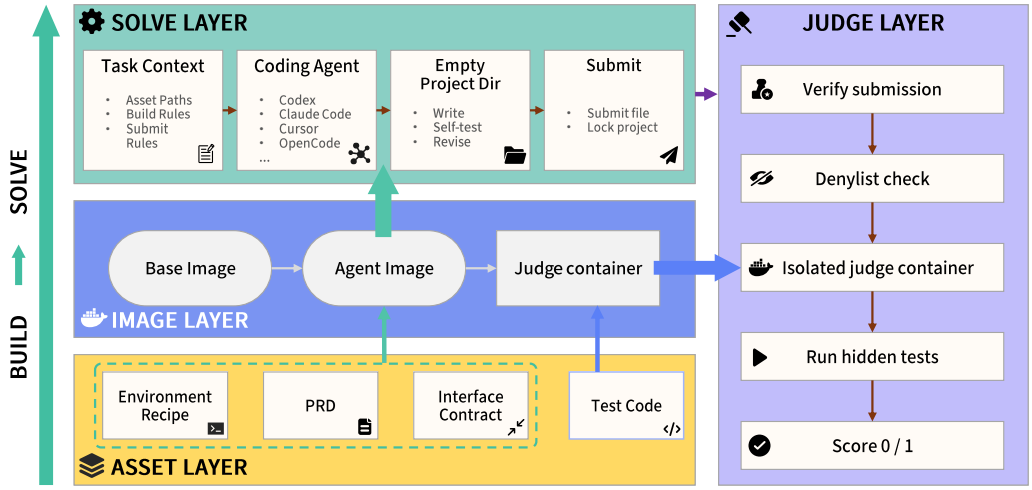

Figure 2: Zero2Repo evaluation architecture. *Asset layer*: the environment recipe, PRD, and Interface Contract are built into the agent image; the test code is used only by the judge. *Image layer*: a base image shared by all tasks, a per-task agent image built from the environment recipe without hidden tests, and the judge environment, which is a fresh container started from the agent image with the hidden tests injected rather than a separately built image. *Solve layer*: the agent receives the task context (specification paths, build rules, and submission rules), works in an empty project directory, and submits by writing a submit file. *Judge layer*: the submission is verified and checked against the package denylist, the workspace is copied into the isolated judge container, the hidden tests are run, and a binary score is recorded.

Acceptance tests and neutralization. Acceptance tests turn PRD feature points into executable checks against the interfaces fixed by the Interface Contract. Upstream tests are rewritten against these interfaces so that they no longer depend on private helpers, source layout, or undocumented implementation choices. Expected outcomes are computed without calling the candidate, using standard-library computations, fixed vectors, or external processes; a signing test, for example, compares the output with an independently computed HMAC. Before release, identifying cues in public artifacts, such as project and author names and repository URLs, are replaced with neutral names, and interface renames are propagated to the tests. A per-task list of sensitive terms drives an identifier-boundary scan for residual cues. Neutralization removes explicit identity cues; it does not prevent a model from recognizing the project or recalling its code.

Verification and review. A task is released only after it passes execution checks and review. The environment must rebuild from its recipe, and a reference implementation adapted from the pinned source to the neutral interfaces must pass the full acceptance suite. Adversarial validation then checks that the suite rejects incorrect implementations: implementations with injected defects or prohibited shortcuts must receive zero reward. Structural checks verify that requirements, interface references, and test references agree. Model-assisted review and author inspection look for specification gaps, unintended structural constraints, and weak tests. Any defect found sends the affected artifacts back for revision and revalidation. Each released task went through such rounds until none found a defect, at most four per task, followed by author inspection (Appendix E).

Dataset composition. The current release contains 11 tasks (Appendix Table 4): parsers, security libraries, a protocol state machine, a CLI framework, a version-control large-file client, and an NLP toolkit. Eight tasks use Python and one each uses TypeScript, Go, and C++, so the release does not support balanced comparisons across languages. The tasks contain 87 feature points and 87 acceptance modules, between 4 and 18 per task. A module groups related test cases, so module counts are smaller than test-case counts. The combined length of the PRD and Interface Contract ranges from 10,829 to 33,408 whitespace-delimited words per task (median 19,358).

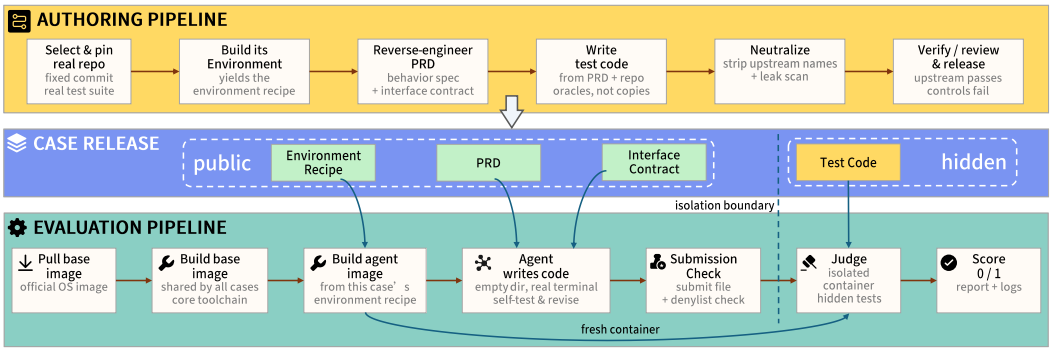

Figure 3: Zero2Repo pipeline. *Authoring*: a repository is pinned, its environment recipe is built, a PRD and Interface Contract are reverse-engineered from its behavior, test code is written from the PRD and the repository, public names are neutralized, and the task is verified, reviewed, and released. *Release*: the environment recipe, PRD, and Interface Contract are public; the test code is hidden. *Evaluation*: the base image is pulled and built once, the agent image is built from the task's environment recipe, the agent writes code in an empty directory, and the submission is checked. Across the isolation boundary, the judge runs the hidden tests in a fresh container started from the agent image and records a binary score with logs.

3.3 Evaluation Framework

The `cbrun` runner executes each trial in two phases, solve and judge, and records the same information for every configuration.

Agent execution and submission. The agent runs in the task's solve container. The instruction names the paths of the PRD and Interface Contract rather than inlining them, states the build contract, and explains how to submit (Appendix C). Code-hosting sites are blocked, package-manager shims refuse to install the upstream package, and a post-submission scan rejects imports of the upstream implementation. A submission is an explicit marker file checked after the agent exits; a successful CLI exit is not a submission. A denylist violation found by the scan allows one correction attempt, with no hidden-test feedback.

Independent acceptance. The agent image contains no hidden tests. After a valid submission, the runner copies the workspace into a fresh container started from the agent image with networking disabled, injects the hidden suite, and runs it. The copy keeps local dependencies, build artifacts, and agent-written tests; it drops changes outside the workspace, globally installed packages, and solve-time environment variables. The judge does not run the project's installation or build commands, so the workspace must already contain the artifacts named by the build contract; acceptance tests may compile their own test drivers. Judge output is stored for analysis and never returned to the agent.

Budgets and outcomes. The solve and judge phases each have a wall-clock limit of 7200 seconds. A task receives reward 1 only if the judge report shows that at least one test ran and every test passed; failures, errors, and missing submissions receive 0. Trial records distinguish solve timeouts, incomplete judge runs, and test failures, and separate infrastructure failures from implementation failures. Supported agent harnesses and recorded trial metadata are described in Appendix F.

4 Experiments

4.1 Setup

Configurations. We evaluate three model–harness configurations: GPT-6 Astra with Codex (codex-cli 0.155.0), Claude Opus 5.5 with Claude Code (2.1.280), and Grok 4.7 High with the Cursor CLI (2026.09.18-9a7762b). Each model is run through a single harness; this release does not compare harnesses for a fixed model. Solve and judge phases each have a 7200-second limit (§3.3).

Table 2: Results on the 11 released tasks, one adopted run per task. Time is the mean time from start to submission; cost is the total token cost over the 11 tasks.

Model	Harness	Solved	Pass rate	Submitted	Time (min)	Cost ($)
GPT-6 Astra	Codex	10/11	90.9%	11/11	14.4	47.04
Claude Opus 5.5	Claude Code	8/11	72.7%	11/11	19.7	173.35
Grok 4.7 High	Cursor CLI	7/11	63.6%	11/11	51.5	45.75

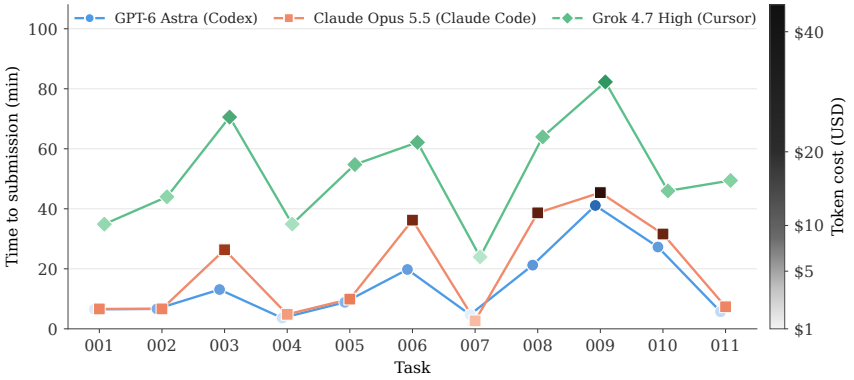

Figure 4: Time to submission per task for each configuration. Marker shade encodes the token cost of the run: each configuration keeps its own hue, and darker means more expensive. The gray color bar gives the mapping from shade to cost.

Protocol and metrics. Each configuration is run once on each of the 11 tasks. The primary metric is the pass rate, the fraction of tasks with reward 1. We also report the time from start to submission and the token cost. Two runs did not submit on the first attempt. Claude Opus 5.5 ended its first session on task 010 without submitting and was resumed; its reported time includes both sessions. The first Grok 4.7 High run on task 011 was interrupted and is replaced by a rerun. Versions and budgets are listed in Appendix F.

4.2 Main Results

Table 2 summarizes the results. GPT-6 Astra solves 10 of 11 tasks, Claude Opus 5.5 solves 8, and Grok 4.7 High solves 7. Every adopted run submitted a workspace, and no run reached the solve limit; every failure is a submission that did not pass all hidden tests. GPT-6 Astra fails only task 009 (Go). Claude Opus 5.5 also fails tasks 006 and 008, and Grok 4.7 High additionally fails task 003. The failing submissions pass between 88.7% and 99.4% of their hidden tests (§5).

Time and cost. Figure 4 shows time to submission and token cost per task. Grok 4.7 High takes the longest on every task (mean 51.5 minutes), while its total cost ($45.75) is similar to that of GPT-6 Astra ($47.04). Claude Opus 5.5 is the most expensive configuration ($173.35 in total), with per-task costs of up to $43.25 on task 009. For all three configurations, the tasks that take longest are also the tasks they fail, with task 009 the longest for each; longer runs did not turn these tasks into passes.

Results by language. On the eight Python tasks the configurations solve 8, 7, and 7 tasks; the only Python failure is task 006, missed by Claude Opus 5.5 and Grok 4.7 High. On the three non-Python tasks they solve 2, 1, and 0 (Appendix Table 7). Because TypeScript, Go, and C++ are each represented by a single task, these numbers describe individual tasks and do not support conclusions about language-level difficulty.

Table 3: Failed hidden tests by failure mode, summed over each configuration's failing tasks. Each failed test is assigned to one mode.

Failure mode	Grok 4.7 High	Claude Opus 5.5	GPT-6 Astra
Long-tail edge-case rule	27 (31%)	12 (26%)	8 (38%)
Less common subsystem non-functional	45 (52%)	2 (4%)	0
Single omission breaking a test group	5 (6%)	28 (61%)	13 (62%)
Environment-dependent behavior	10 (11%)	2 (4%)	0
Crash	0	2 (4%)	0
Total	87	46	21

5 Failure Analysis

The current release was built from well-known repositories, so the models have very likely seen the source projects during training. We therefore ask why familiar repositories are still not reproduced in full, what the remaining failures have in common, and what a benchmark of this form can tell agent and model developers. The analysis covers the eight failing runs in Table 2, using the judge logs and the final workspaces. Every failed hidden test was read individually and assigned to one failure mode based on its first error message.

Failures are near misses, not missing knowledge. The eight failing submissions pass between 88.7% and 99.4% of their hidden tests (Appendix Table 8). All eight submitted a workspace before the budget, and all eight were judged to completion; no failure is a timeout, a build error, or a missing package. Familiarity with the source is visible in the code: on one parser task, 71% of the normalized lines in the Claude Opus 5.5 implementation are identical to the upstream implementation, and 26 of 34 long comments match verbatim. The agents therefore know these repositories well enough to reproduce their main functionality. Under all-or-nothing acceptance, however, a single unimplemented rule is sufficient for a task to fail, and the remaining sections examine where those rules are.

Failure modes. Table 3 assigns each failed test to one of five modes. A *long-tail edge-case rule* is a rule stated in the specification whose surrounding feature works but whose rare branch does not: empty input, an unknown server state, an option combination, a retention or exclusion rule. A *less common subsystem non-functional* failure is one where the core semantics of a whole component are wrong, so that most tests of that component fail. A *single omission breaking a test group* is one root cause, usually an interface or convention shared with the ecosystem, that fails many tests at once. *Environment-dependent behavior* covers rules whose correctness depends on the process environment: file descriptors, terminals, encodings, TLS, and the Git working tree. A *crash* is an uncaught runtime error in the submitted program. The boundary between the first two modes involves judgment: a feature whose body is right but whose rare rule is wrong is counted as long-tail; a component whose core semantics are wrong is counted as non-functional.

The failure structure changes with capability. The three configurations fail in different ways, and the ordering matches their pass rates. Grok 4.7 High fails mainly on breadth: 52% of its failed tests belong to less commonly used subsystems that do not work at all, such as URL pattern matching, custom transfer agents, and pure-SSH transfer. Claude Opus 5.5 implements these subsystems largely correctly; its failures are the last mile, mostly long-tail rules plus one omission that breaks 28 tests. GPT-6 Astra fails only on task 009, and its 21 failed tests reduce to eight root causes, three of which account for 13 tests and are each a one-line fix. As the configurations get stronger, non-functional subsystems disappear first, then environment-dependent behavior, while long-tail rules and single-point inconsistencies remain. Spending more time did not close the gap: for every configuration, the tasks it fails are also the tasks on which it runs longest (Figure 4). The two largest single omissions rest on upstream or ecosystem conventions that the specification does not state; both will be written into the Interface Contract in the next release. Per-task evidence, including these conventions and the rules missed by every configuration, is given in Appendix H.

Implications for agents. Every failure in this study was detectable before submission from the public specification alone, yet none of the agents checked for it. The gap is methodological: the agents verified what they had built rather than what was specified, so their own tests inherited their own omissions. A specification-driven verification loop closes this gap: derive one executable check from each clause of the specification, including error paths, rarely used options, every declared call form, and non-default environments; run the checks against the workspace; and gate submission on their passing. Verification anchored in the specification rather than in the implementation targets exactly the long-tail rules and single omissions that dominate Table 3.

Implications for training. The failing submissions already pass nearly all hidden tests, so the remaining gains lie in coverage of the specification rather than in the amount of code generated. Three directions follow. Supervised data should weight rarely exercised behavior by its importance under acceptance rather than by its frequency in the corpus, pairing specification clauses with the tests that check them, and should represent ecosystem conventions and runtime behavior through execution traces rather than source code alone. Reinforcement learning should use the full-repository all-pass result as the reward, so that a missed rule costs as much as a missing subsystem. Finally, verification behavior should itself be rewarded, so that deriving and running checks before submission becomes part of the policy rather than an instruction the agent may skip. The task assets provide these environments (§3.1).

6 Conclusion, Limitations, and Future Work

Zero2Repo measures whether a coding agent can take a product specification and an empty directory and deliver a repository that runs in its native ecosystem. Two results stand out. First, even with 11 tasks drawn from well-known repositories, the benchmark separates current frontier configurations: GPT-6 Astra, Claude Opus 5.5, and Grok 4.7 High solve 10, 8, and 7 tasks, an ordering consistent with the expected standing of these models, and because every failed hidden test is recorded, each failure is traced to a specific missed rule rather than reported as a score alone (§5). Second, the authoring pipeline is language-agnostic, and its validation stages leave one to two hours of author inspection per task, so the benchmark can grow across languages and be refreshed with newly published repositories without manual authoring. The same task assets also yield supervised trajectories and reinforcement-learning environments.

Limitations. The main limitation is scale. The current release contains 11 tasks, eight in Python and one each in TypeScript, Go, and C++, so non-Python results describe single tasks, and the seed projects are well known, so pass rates are probably higher than they would be on unseen repositories. Extension to 30–50 tasks with a balanced language distribution and a mix of older and recently published repositories is in progress. The evaluation is also limited in coverage. Each configuration was run once per task, so run-to-run variance is not measured; GPT-6 Astra was run at medium reasoning effort rather than its highest setting; and only three frontier configurations were tested. Claude Fable 5.1 was excluded because it sometimes declined to proceed on safety grounds during task construction, and lower-cost models such as DeepSeek and Kimi have not yet been evaluated. Repeated runs, higher reasoning settings, and a broader set of models are planned for the next release.

Future work. The benchmark will be versioned: as model training data moves forward, new versions will draw tasks from the most recently published repositories, and where possible from private repositories, to limit exposure in training data.

AI use statement

Generative AI tools were used in three ways. First, they are part of the task-authoring pipeline described in §3.2: models read upstream source code, draft the product requirements documents and interface contracts, propose acceptance tests, and run commands during task construction. Every artifact produced this way passed the execution checks and adversarial validation described in §3.2 before release, and the released tasks were inspected by the authors. Second, we used generative AI tools to draft and revise sections of the manuscript, to translate drafts between Chinese and English, and to produce initial versions of the schematic figures, which the authors then redrew. Third, the

coding agents evaluated in §4 are generative AI systems and are the object of study. All claims, numbers, and analyses in the paper were checked by the authors against the experiment records, and the authors take full responsibility for the content of this work.

ETHICS STATEMENT

Zero2Repo is built from publicly available open-source repositories released under permissive licenses (for example MIT, BSD, and Apache-2.0), and derived artifacts follow the terms of the corresponding licenses. Public artifacts are neutralized to remove author and organization names and other identifying information; hidden tests and environment recipes are derived from the projects' own tests and build configurations. The benchmark evaluates software agents, not human subjects. Releasing a benchmark on which agents can be trained may accelerate saturation; the benchmark is therefore versioned, and new versions draw tasks from recently published repositories. Denylists containing upstream package names are distributed privately so that the release does not publish a direct mapping from neutralized tasks to upstream projects.

REPRODUCIBILITY STATEMENT

The evaluation harness, the released tasks (public specifications, hidden suites, and environment recipes), and the scripts used to run the experiments are available at `https://anonymous.4open.science/r/Zero2Repo-3AEC/README.md` and will be released publicly. The task-authoring pipeline is not released; §3.2 and Figure 3 describe its stages and the checks a task must pass before release. §3.3 describes the isolation and scoring procedure; Appendix F lists the agent CLI versions, models, and budgets used in every run; Appendix E describes the validation each task passed before release. Environments rebuild from public base images and recipe locks without a private cache. Trial outputs include the reward, judge logs, agent logs, and reproducibility metadata (agent-specification hash, model, CLI version, and the names of forwarded credentials), as described in Appendix F.

REFERENCES

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.

Mark Chen, Jerry Tworek, Heewoo Jun, Qiming Yuan, Henrique Ponde de Oliveira Pinto, Jared Kaplan, Harri Edwards, Yuri Burda, Nicholas Joseph, Greg Brockman, et al. Evaluating large language models trained on code. *arXiv preprint arXiv:2107.03374*, 2021.

Zhaoyang Chu, Jiarui Hu, Xingyu Jiang, Pengyu Zou, Han Li, Chao Peng, Peter O'Hearn, Earl T. Barr, Mark Harman, Federica Sarro, and He Ye. TerminalWorld: Benchmarking agents on real-world terminal tasks. *arXiv preprint arXiv:2605.22535*, 2026.

Jingzhe Ding, Shengda Long, Changxin Pu, Huan Zhou, Hongwan Gao, et al. NL2Repo-bench: Towards long-horizon repository generation evaluation of coding agents. *arXiv preprint arXiv:2512.12730*, 2026.

Yangruibo Ding, Zijian Wang, Wasi Uddin Ahmad, Hantian Ding, Ming Tan, Nihal Jain, Murali Krishna Ramanathan, Ramesh Nallapati, Parminder Bhatia, Dan Roth, and Bing Xiang. CrossCodeEval: A diverse and multilingual benchmark for cross-file code completion. In *Advances in Neural Information Processing Systems (NeurIPS)*, 2023.

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.

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.

Bowen Li, Wenhan Wu, Ziwei Tang, Lin Shi, John Yang, Jinyang Li, Shunyu Yao, Chen Qian, Binyuan Hui, Qicheng Zhang, et al. DevBench: A comprehensive benchmark for software development. *arXiv preprint arXiv:2403.08604*, 2024.

Tianyang Liu, Canwen Xu, and Julian McAuley. RepoBench: Benchmarking repository-level code auto-completion systems. In *International Conference on Learning Representations (ICLR)*, 2024.

Mike A. Merrill, Alexander G. Shaw, Nicholas Carlini, Boxuan Li, Harsh Raj, et al. Terminal-bench: Benchmarking agents on hard, realistic tasks in command line interfaces. International Conference on Learning Representations (ICLR), 2026. arXiv:2601.11868.

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

Zhiyuan Peng, Xin Yin, Pu Zhao, Fangkai Yang, Lu Wang, Ran Jia, Xu Chen, Qingwei Lin, Saravan Rajmohan, and Dongmei Zhang. RepoGenesis: Benchmarking end-to-end microservice generation from readme to repository. *arXiv preprint arXiv:2601.13943*, 2026.

Konstantinos Vergopoulos, Mark Niklas Müller, and Martin Vechev. Automated benchmark generation for repository-level coding tasks. *arXiv preprint arXiv:2503.07701*, 2025.

Daoguang Zan, Zhirong Huang, Wei Liu, Hanwu Chen, Linhao Zhang, Shulin Xin, Lu Chen, Qi Liu, Xiaojian Zhong, Aoyan Li, et al. Multi-SWE-bench: A multilingual benchmark for issue resolving. *arXiv preprint arXiv:2504.02605*, 2025.

Zhaoxi Zhang, Yiming Xu, Weikang Li, Jiahui Liang, and Yunfang Wu. RepoZero: Can LLMs generate a code repository from scratch? *arXiv preprint arXiv:2605.07122*, 2026.

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*, 2024.

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

A Released Tasks

Table 4: Tasks in the first Zero2Repo release. The product name is the neutralized name given to the agent; the source is the upstream project the task was derived from. An acceptance module groups related test cases.

Task	Product	Source	Language	Domain	Acceptance modules
001	Tomlparse	tomli	Python	TOML v1.1 parser	5
002	python-envfile	python-dotenv	Python	`.env` loading and expansion	8
003	ymlcodec	js-yaml	TypeScript	YAML codec with schemas and tags	7
004	Signtoken	itsdangerous	Python	HMAC signing and serialization	4
005	httpwire	h11	Python	HTTP/1.1 protocol state machine	7
006	Optlyn	Click	Python	CLI framework with shell completion	14
007	Otpkit	pyotp	Python	One-time password generation	5
008	Hrefparse	ada-url	C++	WHATWG URL parsing	4
009	Git Orbulk	git-lfs	Go	Version-control large-file client	18
010	Lingora	NLTK	Python	NLP toolkit with model resources	7
011	PathSel	jmespath	Python	JSON path query language	8

B TASK PACKAGE LAYOUT

Each task under `cases/<task_id>/` contains the following files. Entries marked "no" are never present in the agent image.

Path	Role	Visible to agent
`public/Full_PRD.md`	Product requirements	yes
`public/Interface_Contract.md`	Public API / CLI contract	yes
`public/Hardware_Requirements.md`	Optional hardware constraints	yes
`source/manifest.json`	Runner metadata, sensitive terms, judge bans	no
`source/recipe.lock.json`	Environment recipe (base image, install, build)	no
`source/env/resources.json`	Declared external resources (URL, size, SHA-256)	no
`source/denylist.json`	Upstream package and import bans	no
`milestones/final/tests/`	Hidden acceptance tests	no
`milestones/final/test_manifest.json`	Test inventory and command template	no
`controls/<name>/`	Negative-control workspaces with expected reward	no

C AGENT INSTRUCTION

The instruction below is the text given to every agent for a task without a build step. The build-contract section at the end changes with the task's build command and working directory.

```
# Development Task

You are an autonomous software engineer. Build the complete project
    described
in the specification files listed below, from scratch, in your current
    working
directory. Implement every step of the development plan so that the
    finished
project fully satisfies the specification.

Read those files in full before you start writing code.

When you are done, the workspace must be in a state the hidden tests
    can use
directly. See the Build contract below for whether a build step is
    required
and where outputs must remain.

---

## Your environment

* Your workspace is `/app` (your current working directory).
  Build the project here. See the Build contract below for how the judge
  locates outputs.
* The full specification is in these files. Read them in full before
    you start:
  the PRD at `/environment/prd/Full_PRD.md` and the Interface Contract
      at
  `/environment/Interface_Contract.md`.
* Language runtimes and dependencies the project needs are already
    installed in
  this image. You have network access for your own model/tool calls.

## How you work

* You may develop freely: there is no limit on the number of steps,
    turns, edits
  or commands. The only limit is a wall-clock time budget for the whole
      session.
* **Implement from scratch.** Build the target system described in the
    PRD and
```

```
  Interface Contract yourself. Do not download, clone, vendor, copy, or
      install
  an existing upstream implementation of that target system (for example
       via
  `pip install`, `npm install`, or `cargo add` for the product you are
      building).
  General-purpose libraries and tools are allowed; the described product
       behavior
  must be your own code.
* Do not treat a runtime- or toolchain-bundled implementation of the
    same kind
  of product as a dependency or an oracle. The behavior you deliver must
       be
  your own code.
* GitHub and other code-hosting sites for upstream projects are **not
    reachable**
  from this environment during your session.
* You are encouraged to write and run your OWN tests and checks
    repeatedly to
  validate your implementation against the PRD and Interface Contract,
      then fix
  and iterate. A real develop -> test -> debug loop is expected, not a
      single
  pass.
* When you are confident the implementation is complete, write the
    submit file
  described below, then you may end your session. Ending the session is
      **not**
  a submission.
* Submit by writing exactly this one-line file (no extra words):
  path `/logs/agent/submit`
  contents `CODINGBENCH_SUBMIT`
* If that file is missing or its contents are wrong, this attempt fails
     and the
  hidden acceptance tests will not run.

## How you are scored

* After a valid submit file is present, a hidden acceptance test suite
    is run
  against your final workspace state and produces a binary pass/fail
      reward.
* The hidden tests are NOT present in this container and you cannot
    access them.
  Do not look for them, and do not special-case any test: implement the
      public
  Interface Contract behavior fully and correctly.

## Build contract

* The judge does **not** run any install or build step for you. The
    hidden
  acceptance tests run against your final `/app` exactly as you leave it
      .
* There is no build step for this project.
* The hidden tests run against `/app` as the working directory.
```

D EXAMPLE SPECIFICATION EXCERPT

The following excerpts from task 001 (Tomlparse) show the level of abstraction of the PRD and the Interface Contract. Text is quoted verbatim; omissions are marked with an ellipsis.

PRD, terminology (excerpt).

Term	Meaning in this PRD
TOML document	A Unicode text that the caller treats as one TOML v1.1.0 document. The product parses one document per call.
String-parse entry	The library entry that accepts TOML as a Python text string and returns a mapping, or fails.
Float converter	An optional callable the caller supplies so TOML floats (including `inf` and `nan` spellings) are built as something other than a Python float. Specified in FP-05.
Decode error	The product's documented parse-failure exception. It is a kind of value error. Specified in FP-04.
Discrimination	An assertion's ability to distinguish a faithful implementation from a hollow, skipped, or proxy one.

PRD, feature point FP-05 (excerpt).

> **FP-05: Construct TOML floats with a caller-supplied converter.** *Public entry:* The optional float converter on both the string-parse entry and the binary-file parse entry. When the caller does not supply a converter, FP-02's default (Python float) applies. This feature point does not change integers, strings, booleans, date-times, tables, or arrays.
>
> *Normal behavior:* For `precision-matters = 0.982492` and a converter that builds a standard-library decimal from the text it is given, the value of `precision-matters` is a decimal equal to the decimal of the characters `0.982492`, not a binary float of that magnitude. [...] Integers are not passed through the converter. In `a = 1` then `b = 1.0`, with a decimal converter, `a` remains a Python integer and `b` is a decimal.
>
> *Boundary / error behavior:* If the converter returns a dictionary or a list (including a subtype of either), the parse fails with a value error. [...] This is not a decode error: the observer can tell an illegal converter result from invalid TOML.
>
> *Verifiable oracle:* [...] *Failure / absence:* the converter is ignored and values stay Python floats; integers are also converted; a converter that returns a dictionary or a list is accepted and the result is treated as a table or array; the illegal-converter failure is reported as a decode error.

Interface Contract, `tomlparse.loads` (excerpt).

> `loads(s, *, parse_float=float)`
>
> `s` — the TOML document as a Python text string. Passed positionally. A bytes object, a boolean, a file object, or any other non-text value is refused with `TypeError`, not `TOMLDecodeError`.
>
> `parse_float` — optional callable that receives the spelling of a TOML float (including `inf` and `nan` tokens) and returns the constructed value. Keyword-only. The default is the builtin `float`. Integers, strings, booleans, and date-times are not passed through this converter. If the converter returns a dictionary or a list (including a subtype of either), the parse fails with `ValueError`, not `TOMLDecodeError`.

E VALIDATION BEFORE RELEASE

Every released task went through repeated rounds of validation followed by author inspection, until a round found no defect; no task needed more than four rounds (Table 5). In each round, a frontier coding agent solved the task under the release harness, and the resulting run was reviewed along three dimensions. *Run integrity*: the judge collected and ran the full hidden suite, the recorded counts are consistent, and the judged workspace matches the agent's final submission. *Specification, tests, and oracles*: every behavior the tests check is stated in the public specification, expected values are computed independently of the candidate, and the environment does not leak the upstream implementation. *Adversarial validation*: the passing implementation is mutated to break specified behavior, replaced by an empty or hollow workspace, or combined with a prohibited shortcut, and each variant must be rejected. The upstream implementation adapted to the neutral interfaces serves as the reference and must pass the full suite.

These reviews found defects that single-pass checks miss, for example a suite that accepted an implementation violating a stated no-I/O constraint, a status-code rule without a rejecting assertion, a toolchain that bundled the upstream implementation and could serve as an oracle, and a missing denylist entry. Each defect was fixed in the affected artifacts and the task was validated again in the next round. After its final round and author inspection, each of the 11 tasks was released with no open defects.

Table 5: Validation rounds per task before release.

Task	001	002	003	004	005	006	007	008	009	010	011
Rounds	3	1	3	3	2	4	2	2	3	3	2

This validation process is now part of the authoring pipeline and runs automatically for every new task. Author inspection is still required, but it takes about one to two hours per task. As agents become more capable, or with stronger and more expensive agents as reviewers, we expect task production to become fully automatic.

F Agent Integration, Versions, and Budgets

The runner supports Codex, Claude Code, OpenCode, and the Cursor CLI through declarative agent specifications, and each evaluated system is a model–harness configuration. Every trial records task and test provenance, image identifiers, the agent-specification hash, the resolved model, the CLI version, the budgets, execution logs, and wall time, as well as token usage when the harness reports it. The versions and budgets used in our experiments are listed below.

Model	**Harness**	**CLI version**	**Solve / judge limit**
GPT-6 Astra	Codex	codex-cli 0.155.0	7200 s / 7200 s
Claude Opus 5.5	Claude Code	2.1.280	7200 s / 7200 s
Grok 4.7 High	Cursor CLI	2026.09.18-9a7762b	7200 s / 7200 s

G Additional Results

Table 6: Per-task results. "Pass" means that every hidden test passed; otherwise the number of passed hidden tests is shown. Time is seconds from start to submission; cost is in US dollars.

	GPT-6 Astra			**Claude Opus 5.5**			**Grok 4.7 High**		
Task	Result	Time	Cost	Result	Time	Cost	Result	Time	Cost
001	Pass	391	1.32	Pass	398	5.49	Pass	2093	2.95
002	Pass	399	4.36	Pass	399	5.43	Pass	2640	3.20
003	Pass	786	5.42	Pass	1581	17.28	499/507	4234	6.10
004	Pass	217	1.65	Pass	290	3.94	Pass	2095	2.40
005	Pass	530	3.90	Pass	595	5.95	Pass	3285	4.55
006	Pass	1187	5.34	473/476	2174	24.78	467/476	3729	5.25
007	Pass	283	1.21	Pass	160	2.60	Pass	1438	2.05
008	Pass	1274	5.83	220/248	2319	31.22	223/248	3838	4.35
009	477/498	2467	10.45	483/498	2724	43.25	453/498	4937	7.60
010	Pass	1636	5.57	Pass	1896[a]	28.03	Pass	2758	3.85
011	Pass	342	1.99	Pass	441	5.38	Pass	2966	3.45

[a] Sum of a first session that ended without submission (1599 s) and a resumed session (297 s).

H Failure Details

This appendix gives the per-task evidence behind §5.

Table 7: Results by language. Python has eight tasks; TypeScript, C++, and Go have one task each.

Model	Python	TypeScript (003)	C++ (008)	Go (009)
GPT-6 Astra	8/8	Pass	Pass	Fail
Claude Opus 5.5	7/8	Pass	Fail	Fail
Grok 4.7 High	7/8	Fail	Fail	Fail

Table 8: Hidden tests passed by the failing submissions.

Task	Grok 4.7 High	Claude Opus 5.5	GPT-6 Astra
003 YAML codec	499/507 (98.4%)	Pass	Pass
006 CLI framework	467/476 (98.1%)	473/476 (99.4%)	Pass
008 URL parser	223/248 (89.9%)	220/248 (88.7%)	Pass
009 Large-file client	453/498 (91.0%)	483/498 (97.0%)	477/498 (95.8%)

Rules missed by every configuration. Task 009 is the only task that all three configurations fail, and four of its hidden tests fail for all three: staged objects matched by an exclusion rule must be kept; the verbose output of the prune command must be distinguishable from the default; the user must be prompted when server-side lock support is unknown; and shell completion at the flag position must offer flag candidates rather than Git reference names. Nine of the fifteen tests that Claude Opus 5.5 fails on this task also fail for Grok 4.7 High, and on the CLI framework (task 006) both miss the same rule, capturing writes made directly to file descriptors 1 and 2. Each of these rules is stated explicitly in the public specification and can be checked without the hidden tests. Their shared failure across models from three vendors points to a common cause rather than to per-model noise: these branches are rarely exercised in the upstream code and rarely discussed in the surrounding corpus, so they are underrepresented in whatever the models have memorized. We caution against over-reading the overlap on task 008: fourteen tests fail for both Grok 4.7 High and Claude Opus 5.5, but for different reasons, one a semantic error in matching and the other a compile failure, and we do not count them as a shared blind spot.

Grok 4.7 High. In the URL parser (task 008), the pattern matcher binds the second path segment of `/foo/bar` to the first named group of `/:a/:b` (14 tests), and the protocol setters refuse changes that the specification requires, including `wss` to `http` (11 tests). In the large-file client (task 009), the custom transfer agent protocol stalls until the 60-second test timeout (8 tests) and pure-SSH transfer is unusable (12 tests). Its long-tail failures on the YAML codec (task 003) are all rules of the format itself: whitespace-only documents are rejected, the integer key `1` and the string key `"1"` collapse into one key, the YAML 1.1 set syntax `? key` is unhandled, and text emitted with the no-extra-indent option cannot be read back.

Claude Opus 5.5. Eleven of its fifteen failed tests on task 009 are long-tail rules, and two are crashes of the submitted Go program (`slice bounds out of range`). All 28 failures on task 008 have one cause, described below.

GPT-6 Astra. Its 21 failed tests on task 009 reduce to eight root causes. Three of them account for 13 tests: the version banner appends the Git version (5 tests), the option table of the filter process is empty so `--skip` is rejected as unknown (2 tests), and the JSON status output stores whole porcelain lines where the specification requires the path (6 tests).

Implicit conventions. In the Claude Opus 5.5 URL parser, the compiled pattern type has no empty constructor, so the declaration `tl::expected<url_pattern<P>, errors> compiled;` used by the hidden probe does not compile and 28 tests fail before any matching logic is exercised. In the Grok 4.7 High large-file client, the SSH program is hard-coded as `ssh` instead of honoring the Git convention `GIT_SSH`, so every SSH test fails with `executable file not found`. The Interface Contract of task 008 states only that callers declare the compiled result as `tl::expected`, and neither the PRD nor the Interface Contract of task 009 mentions `GIT_SSH`.

Both conventions come from the upstream code or its ecosystem, which is where memory of the upstream code should help most, yet each model missed one of them while the other respected it. A reviewer may reasonably treat these two cases as specification-clarity issues. We report them as findings about both the agents and the benchmark, and both conventions will be written into the Interface Contract in the next release; test-level output is how a sealed benchmark can still improve its own specifications.