COMMANDBENCH: TOWARD DIVERSE, CHALLENGING, AND CONTAMINATION-RESISTANT REPOSITORY GENERATION

Anonymous authors
Paper under double-blind review

ABSTRACT

Coding agents are advancing quickly, and automating the entire software engineering process is becoming an increasingly important setting. Existing benchmarks, however, evaluate this setting only weakly: open-source repositories, the single most important source from which benchmarks are built, have been widely absorbed into the training corpora of frontier LLMs, and it is no longer possible to tell whether a high score reflects genuine ability or a replay of the training corpus. Having experts write tasks by hand does resist contamination effectively, but it runs into a clear bottleneck in scale and diversity. We therefore present `CommandBench`, which asks an agent to implement a complete repository from scratch and encapsulate it as a set of CLI commands. `CommandBench` is resistant to contamination in three ways: (i) lexical obfuscation, which makes a located repository unusable as it stands; (ii) cross-language implementation, which blocks verbatim copying; and (iii) unmet-specification tests, which target behaviour that the specification requires but that the original repository does not yet achieve. Most distinctively, `CommandBench` guarantees that an implementation that reproduces the source repository's behaviour scores exactly zero, so that faithfully replaying the source implementation earns no credit by construction. `CommandBench` is also markedly more diverse and more challenging: its tasks are synthesised from 138 distinct repositories, covering 7 programming languages, with tasks from outside computer science accounting for 48.6% of the benchmark. On `CommandBench`, models pass 89% of the met-specification tests on average, yet none passes more than 63% of the unmet-specification ones, and the best score is only 58.93 out of 100.

1 INTRODUCTION

Practitioners are delegating ever larger units of programming work to coding agents: from a single function (Chen et al., 2021; Austin et al., 2021), to file-level and module-level completion inside an existing codebase (Liu et al., 2024; Jimenez et al., 2024; Yang et al., 2024), and finally to agents that carry out the whole development process end to end (Zhao et al., 2025; Wang et al., 2025). Users therefore urgently need high-quality benchmarks for end-to-end codebase generation in order to quantify what these systems can actually do. A number of benchmarks have responded to this need (Hu et al., 2025; Zhao et al., 2025; Luo et al., 2026; Zeng et al., 2025; Ding et al., 2025; Liu et al., 2026; Peng et al., 2026; Zhang et al., 2026b). We find, however, that they still fall short of a truly fair and comprehensive evaluation, and are inadequate in particular along the three dimensions users care about most.

① **Diversity.** Real development requests span many programming languages, technical ecosystems, and even academic disciplines, yet the coverage of existing benchmarks remains narrow. As shown in Table 1, most benchmarks examine a single language (Zhao et al., 2025; Luo et al., 2026; Zeng et al., 2025; Ding et al., 2025); even the broadest of them covers no more than three. Their technical ecosystems are just as homogeneous, being confined to general-purpose software infrastructure: RepoCraft (Luo et al., 2026), for instance, has more than a thousand tasks, yet they all derive from six data-science or web-framework repositories. Beyond computer science, moreover, the open-

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.

Table 1: Comparison of end-to-end repository generation benchmarks. #Src lan counts the languages of the source repositories and #Tgt lan those in which an implementation is requested. Fresh requires recent repositories; Novel logic adds requirements beyond the source implementation; Obfusc. renames repositories or identifiers; Cross-language requires reimplementation in another language; New tests indicates newly synthesised private tests. A checkmark denotes the presence of the corresponding property.

Benchmark	#Src lan[1]	#Tgt lan[1]	#Repo	Fresh	Novel logic	Obfusc.	Cross-language	New tests
rSDE-Bench (Hu et al., 2025)	1	1	53	×	✓	×	×	✓
Commit0 (Zhao et al., 2025)	1	1	54	×	×	×	×	×
RepoCraft (Luo et al., 2026)	1	1	6	×	×	✓	×	✓
E2EDevBench (Zeng et al., 2025)	1	1	50	✓	×	×	×	×
NL2Repo-Bench (Ding et al., 2025)	1	1	104	✓	×	×	×	×
E2EDev (Liu et al., 2026)	3	3	46	×	×	×	×	✓
RepoGenesis (Peng et al., 2026)	2	2	30	×	✓	×	×	✓
RepoZero (Zhang et al., 2026b)	3	3	35	×	×	×	✓	✓
CommandBench	7	7	138	✓	✓	✓	✓	✓

source ecosystem hosts a large body of code from the natural sciences, engineering, and even the humanities and economics, which these benchmarks ignore almost entirely.

② **Difficulty.** Most existing studies adopt the average pass rate over all tests as their final metric, which treats every test as equally important; the many basic test cases, however, dilute the genuinely difficult ones, drastically reducing the effective range of the metric. An implementation that covers the common paths and fails at every boundary still scores well.

③ **Contamination resistance.** Most existing benchmarks are built from public GitHub repositories. Given how scarce expert human labour is, that choice is reasonable in itself. Unfortunately, these benchmarks all turn out to be weak in contamination resistance. Commit0 (Zhao et al., 2025) restricts retrieval of the source repository and runs a 10-gram memorisation audit, neither of which defends against paraphrased recall. E2EDevBench (Zeng et al., 2025) and NL2Repo-Bench (Ding et al., 2025) rely on freshness-based repository sampling, but freshness decays as models are retrained. RepoCraft (Luo et al., 2026) renames surface symbols, leaving the logic to be implemented unchanged. RepoZero (Zhang et al., 2026b) designs a cross-language reimplementation mechanism, but its task specifications still do not go beyond the source repository, and E2EDev (Liu et al., 2026) reports no explicit defence at all. In contrast, a few benchmarks such as rSDE-Bench (Hu et al., 2025) and RepoGenesis (Peng et al., 2026) have experts and LLMs collaborate to write requirements and tests by hand; this buys higher contamination resistance, but it also condemns them to a lack of scale and diversity, with only 53 and 30 repositories respectively.

This leads to a *trilemma*: genuine contamination resistance requires task specifications that depart from any existing public implementation, and so far such a departure has largely relied on manual authoring by experts, which under a bounded budget makes diversity and difficulty hard to preserve. **How to satisfy all three at once remains an important and open question.**

In this paper we present `CommandBench` in an attempt to break this state of affairs. It asks an agent to write an entire repository end to end and to expose it as a set of self-contained command-line commands. Our starting point is simple: *a command-line contract is language-agnostic*. Once the arguments and the input–output behaviour of a command are fixed, the contract is independent of the implementation language and of the implementation itself, so a single test suite serves every implementation. This frees human effort from hand-authoring specifications for cross-language reimplementation.

On top of this, obfuscation rewrites command and argument names into names unrelated to the source, so that a model which does locate the source repository cannot lift code from it as it stands. Our central mechanism, however, is *unmet-specification tests*: cases that the original repository

[1]C is counted with C++ and JavaScript with TypeScript, as each pair shares one toolchain.



Figure 1: Constructing a `CommandBench` task from `networkx`. Stages 1–4 derive a CLI contract, a wrapper $\mathcal{C}$, and the two suites $\mathcal{T}^+$ (which $\mathcal{C}$ passes) and $\mathcal{T}^-$ (which it fails); stages 5–8 obfuscate every identifier and supply an environment in another language.

ought to pass but that its actual implementation does not. In other words, we score an attempt against the ideal behaviour its specification requires rather than the behaviour the source repository exhibits. `CommandBench` thereby acquires a rather unusual property: an implementation that reproduces the behaviour of the source repository earns no reward whatsoever, whether reproduced verbatim or recalled in paraphrase. Credit therefore accrues only to behaviour that the specification demands and the source implementation does not exhibit.

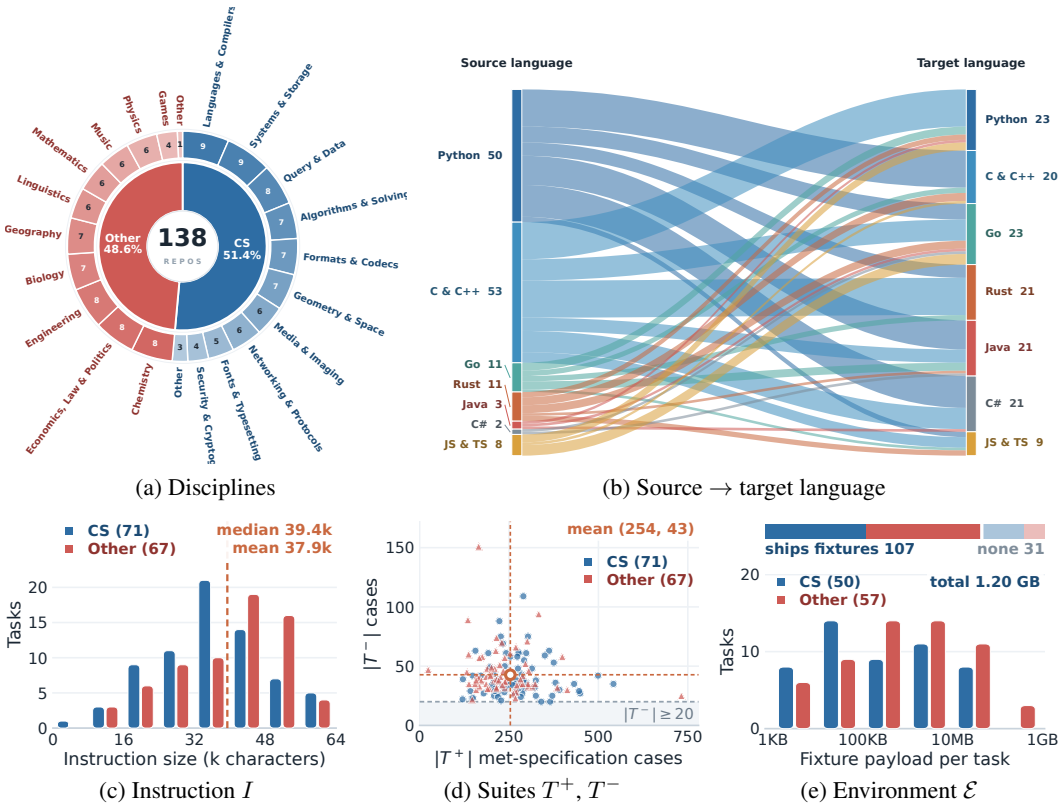
Figure 1 shows the construction of one task end to end. Thanks to this largely automated construction, `CommandBench` enjoys a clear advantage in both diversity and difficulty. Its tasks are synthesised from 138 distinct repositories covering 7 programming languages, with 48.6% of them coming from outside computer science (Figure 2a). In our evaluation, no model scores above 58.93 out of 100: the functional floor is largely within reach, boundary behaviour is not.

2 Curation

Stage 1: Repository sourcing. We assemble a set $\mathbf{R}$ of open-source repositories that serves as the starting point. To be admitted into $\mathbf{R}$, a repository must meet three criteria: (i) its core functionality can be expressed in full as a set of CLI contracts; (ii) it comprises more than 4,000 lines of code; and (iii) it has at least 100 stars.

We grow $\mathbf{R}$ iteratively from the empty set with the help of a deep research agent. The agent first proposes five repositories meeting these criteria as seeds; in each subsequent round we then ask it to steer clear of the functionality, scope and languages already covered by the current set and to propose new repositories instead, until further rounds no longer yield repositories outside the coverage already attained. This procedure enriches the diversity of $\mathbf{R}$ considerably, and proves especially effective at covering the languages, functionality and disciplines that sit in the long tail of the open-source ecosystem.

Stage 2: Plan synthesis. We first build, for each repository $\mathcal{R} \in \mathbf{R}$, an environment $\mathcal{E}_0$ given as a Dockerfile, which checks out the source of $\mathcal{R}$ and builds all of its dependencies. We then have a deep research agent explore that source inside $\mathcal{E}_0$ and produce an initial plan $\mathcal{S}_0$, which records the arguments and the input–output behaviour of every command obtained by turning $\mathcal{R}$ into a CLI.



(a) Disciplines (b) Source → target language

(c) Instruction I (d) Suites T^+, T^- (e) Environment $\mathcal{E}$

Figure 2: **(a)** Disciplines covered. **(b)** Every task targets a language other than its source. **(c)** Length of the instruction the agent reads. **(d)** Size of the two hidden suites; every task carries at least 20 unmet-specification cases, and the two sizes are independent (Pearson $r = -0.11$). **(e)** How many tasks ship a fixture, then the payload of those that do; the 31 that ship none build their inputs from command-line arguments alone.

We subject this stage to three quality checks. (i) *Determinism.* We build $\mathcal{E}_0$ three times over and run the repository's own test samples independently inside each of the three containers, requiring that they pass in full and agree exactly on their output. (ii) *Coverage.* We ask three independent LLM judges whether $\mathcal{S}_0$ covers the core functionality of the repository in full, and accept it only when all three agree that it does. (iii) *Runtime.* We keep only those repositories whose own test suite runs in no more than 30 minutes, so as to avoid excessive cost downstream. Candidates failing (i) or (ii) are granted a single round of revision accompanied by the diagnosis; those that still cannot be repaired are discarded.

Stage 3: Wrapper construction. For each repository, we ask a coding agent to implement a CLI wrapper according to $\mathcal{S}_0$, and denote the wrapped repository by $\mathcal{C}$. During implementation, the agent may make only minor revisions to $\mathcal{S}_0$ based on the concrete implementation, yielding $\mathcal{S}_1$, so as to eliminate any functionality in $\mathcal{S}_0$ that proves infeasible.

We subject each wrapper to two quality checks. (i) *Installation stability.* We independently install the wrapper multiple times and verify that it consistently exposes every command and argument specified in the plan. (ii) *Command-level audit.* With an LLM judge, we inspect each command to ensure (a) behavioural consistency, meaning that its runtime behaviour agrees with the plan, and (b) thinness, meaning that the wrapper introduces no additional logic beyond the input–output conversions that are strictly necessary. Any wrapper that fails these checks undergoes targeted revision.

Stage 4: Test synthesis. With the wrapper in place, the repository is exposed as a set of CLI commands. On this basis, we produce two disjoint test suites.

met-specification tests ($\mathcal{T}^+$): defines the usability floor of $\mathcal{C}$ and consists of basic behaviours that $\mathcal{C}$ already passes reliably.

unmet-specification tests ($\mathcal{T}^-$): probes boundary behaviours. It consists of cases that the specification requires but that the current $\mathcal{C}$ does not yet pass. $\mathcal{T}^-$ measures the robustness of an implementation on complex boundary conditions. Together, $\mathcal{T}^+$ and $\mathcal{T}^-$ define an idealised "utopian implementation" $\mathcal{C}^\star$. No existing implementation satisfies $\mathcal{C}^\star$, so it cannot be recovered by replaying one.

Candidate tests come from three sources: (i) adaptations of tests from the source repository; (ii) bug-fix issues mined from the repository; and (iii) cases found by an agent through autonomous adversarial exploration. We run multiple rounds of test synthesis for each repository until $\mathcal{T}^+$ covers every command and argument and $\mathcal{T}^-$ contains more than 20 semantically distinct cases.

For quality assurance, we require every test to produce consistent output across repeated runs. For each test in $\mathcal{T}^-$, the test-generation agent must also provide a correctness claim. Its supporting evidence may come from promises or statements made by the repository authors in documentation, comments or issues, or from existing interface semantics and behavioural invariants. A case is admitted to $\mathcal{T}^-$ only if its correctness claim is consistent with $\mathcal{S}_1$.

Stage 5: Instruction drafting. Based on the revised plan $\mathcal{S}_1$ and the two test suites, we produce a draft instruction file $\mathcal{I}_0$ for each repository. It specifies the CLI commands to be implemented, their arguments, input–output formats and expected behaviour. These specifications must remain language- and implementation-agnostic, and must not reveal any concrete test cases.

Stage 6: Obfuscation mapping. For each wrapped repository $\mathcal{C}$, we construct an obfuscation mapping $\mathcal{M}$ that assigns every CLI command and argument a new name. We require $\mathcal{M}$ to be bijective. Its purpose is to minimise the likelihood that an evaluated agent can identify and recall the source repository from the instruction file.

Stage 7: Instruction finalisation. We consistently apply the obfuscation mapping $\mathcal{M}$ to every command and argument name in the draft $\mathcal{I}_0$, obtaining the final instruction $\mathcal{I} = \mathcal{M}(\mathcal{I}_0)$.

Stage 8: Environment generation. For each task, we select a target language different from the primary language of the source repository, and construct the corresponding evaluation environment $\mathcal{E}$. The language must suit the task's core functionality: the task must be implementable within its standard ecosystem without relying on another language. The environment is specified by a Dockerfile and includes the public data and fixtures required to implement the task, while the test logic and expected outputs remain hidden.

Taken together, a task $\mathcal{X}$ therefore consists of five components,

$$\mathcal{X} = \big(\mathcal{I},\, \mathcal{M},\, \mathcal{T}^+,\, \mathcal{T}^-,\, \mathcal{E}\big), \tag{1}$$

and `CommandBench` is the set $\mathbf{X}$ of the 138 tasks so obtained (Figure 2).

Task Correction and Freezing The pipeline produces tasks but cannot judge whether they are well formed. We therefore verify and correct every task so as to secure its *satisfiability* and its *inferability*: the former requires that an implementation exist which passes every test, the latter that every behaviour a test asserts be determined by $\mathcal{I}$. The verification applies two probes in turn. The first is a reference implementation of $\mathcal{C}^\star$, obtained by starting from the wrapped repository $\mathcal{C}$ and repairing its boundary behaviour according to the plan $\mathcal{S}_1$. The second is an actual attempt by a set of pilot models, which implement the task in full from an empty repository on the basis of $\mathcal{I}$. Read together, the two yield a diagnosis: a case the reference implementation fails to pass suggests that satisfiability may not hold; a case the reference implementation passes while every pilot model fails suggests that inferability may not hold. An additional independent agent then judges whether either kind of failure originates in the task itself; where it does, we revise $\mathcal{T}$ or $\mathcal{I}$ accordingly and enter the next round. The loop runs for three rounds, and any case still judged after the third round to be in error in the task itself is deleted.

We finally add two human checks. The first concerns *satisfiability*: we inspect a sample of the repairs the reference implementation makes to $\mathcal{C}$ and confirm that each handles its boundary in general terms

Table 2: Main results on `CommandBench`. S is the score of Equation 2. Every column is defined in Table 6. All models are queried at fixed, dated snapshots, each at the highest reasoning effort its provider supports; two of them are additionally run at a lower effort, everything else being equal.

Model	Effort	S	$\mathcal{T}^+$ (%)	$\mathcal{T}^-$ (%)	Time (min)	In	Out	Tool calls
GPT-5.6 Sol	medium	51.34	85.64	59.04	8	1.5M	20.6k	59
GPT-5.6 Sol	xhigh	58.93	93.11	62.56	118	28.4M	57.1k	153
GLM-5.3 Flash	low	49.23	86.22	56.77	23	3.8M	29.6k	70
GLM-5.3 Flash	max	53.41	89.93	58.95	207	99.3M	123.6k	263
Kimi K3	max	56.67	89.54	62.02	221	76.9M	108.7k	233
Grok 4.6	xhigh	55.68	92.11	60.07	157	15.4M	81.4k	200
Muse Spark 1.3	max	52.66	90.79	57.87	142	65.0M	153.2k	287
Qwen-3.8 Flash	max	52.83	84.59	61.20	235	119.4M	186.7k	293
DeepSeek-V4.1 Flash	max	52.47	91.25	57.29	43	31.8M	68.6k	171

rather than branching on a particular input. The second concerns *inferability*: we sample assertions, hide their values, and ask annotators separately for the answer while they can see nothing but $\mathcal{I}$; an answer that disagrees with the assertion marks that assertion as erroneous. A task with few erroneous assertions has them repaired by hand; a task with many is removed as a whole. The benchmark is frozen at this point. Details are given in Appendix A.

3 Evaluation

We instantiate the environment $\mathcal{E}$ and ask the agent to complete the task, guided by $\mathcal{I}$ and using the designated language. Obfuscation reduces the likelihood that the agent recognises and recalls the source repository, while the cross-language setting prevents direct replay of an implementation from the training corpus.

Once the agent produces a repository $\mathcal{A}$, a task-agnostic translator uses $\mathcal{M}$ to map its obfuscated commands and arguments back to their original names, after which $\mathcal{T}^+$ and $\mathcal{T}^-$ are run against it separately. The translator remains hidden from the evaluated agent.

We use the product of the pass rates on the two suites $\mathcal{T}^+$ and $\mathcal{T}^-$ as our metric,

$$S = \frac{|\{t \in \mathcal{T}^+ : \mathcal{A} \text{ passes } t\}|}{|\mathcal{T}^+|} \cdot \frac{|\{t \in \mathcal{T}^- : \mathcal{A} \text{ passes } t\}|}{|\mathcal{T}^-|} \times 100\%. \tag{2}$$

The point of this metric is that the two suites do not compensate for each other: a high pass rate on $\mathcal{T}^-$ cannot make up for a gap in the core functionality, and passing the core in full does not by itself earn any credit for boundary behaviour that the implementation never handles. In particular, a source-faithful implementation cannot pass any test in $\mathcal{T}^-$ and therefore receives zero on this metric, so that faithful replay is worthless under it by construction.

The complete execution and reporting protocol is specified in Appendix B.

4 Experiments

We use `CommandBench` to evaluate seven models: GPT-5.6 Sol, Kimi K3, Grok 4.6, Muse Spark 1.3, Qwen-3.8 Flash, GLM-5.3 Flash, and DeepSeek-V4.1 Flash. Each task is evaluated once, independently, in its target-language environment: the execution environment is allotted 16 GB of memory and a six-hour time budget, the agent may use the network as it would in ordinary development, and its sole reference for the task is the instruction $\mathcal{I}$. The complete execution protocol is given in Appendix B.

4.1 Main Results

Table 2 reports the main results of the seven models on `CommandBench`. Overall, `CommandBench` still poses a substantial challenge to the strongest models available today: the best-performing model, GPT-5.6 Sol at xhigh effort, obtains only 58.93 out of 100, so the benchmark retains ample headroom for discrimination for some time to come.

We further report the average pass rates of each model on the two suites $\mathcal{T}^+$ and $\mathcal{T}^-$ separately, and the gap between them is pronounced: at their highest effort the average pass rate on $\mathcal{T}^+$ is 88.66%, whereas that on $\mathcal{T}^-$ is only 59.99%. In other words, most models manage to build a usable functional floor, yet lose most of their credit on boundary behaviour—precisely the watershed between reproducing an implementation and understanding one: an implementation that faithfully copies the source repository cannot pass a single case of $\mathcal{T}^-$ and therefore necessarily scores zero on S; a high score on $\mathcal{T}^-$ therefore requires behaviour that the source implementation does not exhibit. The cost of a task tells the same story: at their highest effort the models consume between 15.4M and 119.4M input tokens and issue between 153 and 293 tool calls per task, so a submission is the product of sustained engineering effort rather than a replay of training memory.

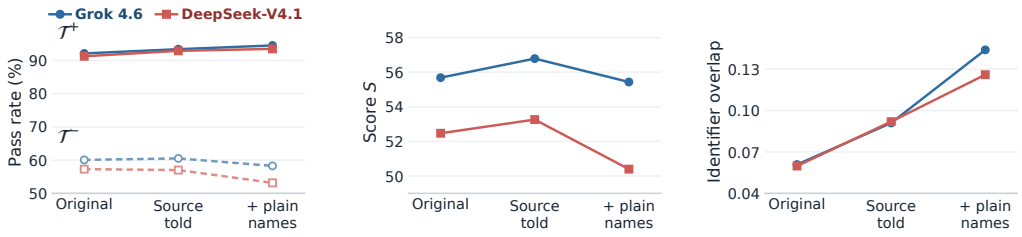
4.2 Mechanism Ablations

Table 3: Cross-language controls on Grok 4.6 and DeepSeek-V4.1 Flash. Each cell gives the value under that condition with its signed difference from the main experiment in parentheses. *Same-language*: every task is implemented in the primary language of its source repository. *All-Python*: every task is implemented in Python regardless of source language. Identifier overlap is the Jaccard overlap between the lexical tokens of the submission and upstream repository (Appendix B).

Model	S	$\mathcal{T}^+$ (%)	$\mathcal{T}^-$ (%)	Id. overlap	Time (min)	Tool calls
Same-language						
Grok 4.6	56.10 (+0.42)	92.53 (+0.42)	60.17 (+0.10)	0.0713 (+0.0104)	149 (−8)	188 (−12)
DeepSeek-V4.1 Flash	53.71 (+1.24)	93.57 (+2.32)	56.88 (−0.41)	0.0644 (+0.0046)	53 (+10)	192 (+21)
All-Python						
Grok 4.6	58.57 (+2.89)	92.43 (+0.32)	62.89 (+2.82)	0.0654 (+0.0045)	140 (−17)	165 (−35)
DeepSeek-V4.1 Flash	56.23 (+3.76)	93.89 (+2.64)	59.63 (+2.34)	0.0630 (+0.0032)	45 (+2)	177 (+6)

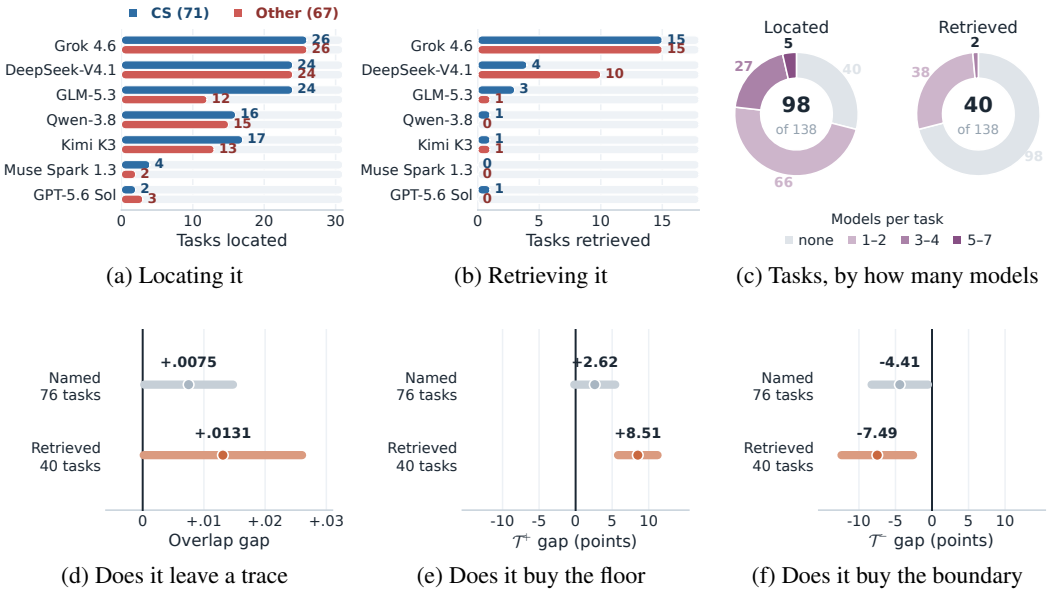
Cross-language. Table 3 varies the target language in two ways, leaving every other setting identical to the main experiment. Under *Same-language*, where each task is implemented in the primary language of its source repository, $\mathcal{T}^+$ rises by 0.42 and 2.32 points on the two models while $\mathcal{T}^-$ barely moves, by $+0.10$ and -0.41; identifier overlap rises on both, the more so of the two conditions. Writing in the language of the source helps a model reproduce the behaviour that source already has, and does little for the behaviour it lacks. Under *All-Python*, where every task is implemented in Python whatever its source language, both suites gain instead: $\mathcal{T}^-$ by 2.82 and 2.34 points, lifting S by 2.89 and 3.76. The models are simply better at writing Python, so the cross-language mechanism does two things at once: it keeps the source implementation from being reused directly, and it keeps the benchmark from resting on the one language the models write best.

Obfuscation. Figure 3 lowers the barrier between the model and the source repository in two steps. The first names the upstream repository in $\mathcal{I}$ and tells the model it may draw on it, leaving the choice of how much to the model itself. The offer is taken up: identifier overlap rises by half, from 0.0609 to 0.0910 on Grok 4.6 and from 0.0598 to 0.0920 on DeepSeek-V4.1 Flash. Yet $\mathcal{T}^-$ does not move, by $+0.45$ and -0.28 points, and S edges up on the strength of $\mathcal{T}^+$ alone. The second step leaves that offer in place and removes the obfuscation, so that the two conditions differ in the mapping and in nothing else. Now $\mathcal{T}^-$ drops by 2.25 and 3.84 points and S falls to 55.43 and 50.40, below where it began. The model could reach the source repository in both conditions; what the mapping takes away is the option of copying it. Under obfuscation the names do not match, so upstream code cannot be pasted in as it stands, and the model has to work out each behaviour from $\mathcal{I}$ instead. In plain names it can copy, and copying a repository that scores zero on $\mathcal{T}^-$ costs it those tests. Obfuscation does not so much hide the source repository as make a located one unusable.



(a) The two pass rates (b) The score (c) Identifier overlap

Figure 3: Obfuscation ablation on Grok 4.6 and DeepSeek-V4.1 Flash. Along the x axis each scenario removes a further barrier between the model and the source repository: *Original* is the task as it ships; *Source told* names the upstream repository in $\mathcal{I}$ and asks the model to follow that implementation; $+$ *plain names* additionally leaves the identifiers unobfuscated. The quantities are split by dimension: **(a)** the two pass rates, **(b)** the score S, and **(c)** identifier overlap with the upstream repository.



(a) Locating it (b) Retrieving it (c) Tasks, by how many models

(d) Does it leave a trace (e) Does it buy the floor (f) Does it buy the boundary

Figure 4: Locating the source repository, and what it buys. A run *names* the repository when it writes the owner and name out in its own text, *retrieves* it when it fetches the repository and receives the content, and *locates* it when it does either. **(a)** Tasks located. **(b)** Tasks retrieved; the rows are ordered as in (a). **(c)** The 138 tasks by how many of the seven models name them, and by how many retrieve them. **(d)–(f)** Identifier overlap, $\mathcal{T}^+$ and $\mathcal{T}^-$, each as a gap taken within one task against the runs that do neither; the point is the mean over tasks, the band its 95% interval, and the count beside each row the tasks the gap is taken over.

4.3 Replay or Understanding?

This section quantifies whether a model profits from the source repository while solving a task.

We read the trajectories for two events. A run *names* the source repository when it writes the owner and name out in its own text, and *retrieves* it when it fetches the repository and receives the content; it *locates* the repository when it does either. By panels (a) and (b) of Figure 4, a run locates the repository in 208 of all runs, at a rate comparable on the computer-science tasks and on the rest. How many tasks a model locates does not in general predict how many it retrieves. Grok 4.6 retrieves on more than half of the tasks it locates, twice the rate of the next model, and the remaining

models are uniformly low despite locating comparably many. Retrieval is therefore better read as a preference of the model than as the extent of what it recognises. Counting the other way round, by models per task, at least one model locates 98 of the 138 tasks, and most of those by only one or two; 40 tasks are retrieved by at least one model. Recognition is thus dispersed across the suite rather than confined to a small set of repositories.

Panels (d) to (f) report what naming or retrieving the repository does to the code the model writes. Identifier overlap with the upstream repository averages 0.0586 on the runs that do neither; naming alone raises it by 0.0075 and retrieval by 0.0131. Interestingly, however, retrieval raises $\mathcal{T}^+$ by 8.51 points while lowering $\mathcal{T}^-$ by 7.49, and neither of those two intervals includes zero.

In summary, naming the repository costs 4.41 points of $\mathcal{T}^-$ and has no effect on $\mathcal{T}^+$ separable from zero. Retrieving it gains 8.51 points of $\mathcal{T}^+$ and loses 7.49 of $\mathcal{T}^-$; multiplied through, however, S falls. This is by design: obfuscation withholds the interface from memory, and the product form of S grants no credit for recovering it, so that a model cannot profit from recalling the source of a task on `CommandBench` (Appendix A.4).

5 Related Work

From repository context to repository construction. Execution-based benchmarks evolved from single-function synthesis (Chen et al., 2021; Austin et al., 2021) through cross-file completion in real repositories (Liu et al., 2024; Ding et al., 2023; Chen et al., 2026) to issue-driven patching on an existing codebase (Jimenez et al., 2024; Yang et al., 2024). A parallel line translates existing programs across languages (Roziere et al., 2020; Ahmad et al., 2023; Yan et al., 2023), taking the source code itself as input. Repository generation instead requires producing an executable codebase from a specification and an empty workspace.

End-to-end repository generation and test adequacy. rSDE-Bench (Hu et al., 2025), Commit0 (Zhao et al., 2025), RepoCraft (Luo et al., 2026), E2EDevBench (Zeng et al., 2025), NL2Repo-Bench (Ding et al., 2025), E2EDev (Liu et al., 2026), RepoGenesis (Peng et al., 2026), and RepoZero (Zhang et al., 2026b) move toward this setting, evaluating against upstream, migrated, or newly authored tests. Measurement by execution is only as strong as the tests: EvalPlus shows that expanded suites expose incorrect programs the original benchmark accepted (Liu et al., 2023); upstream tests reward behavioural reproduction of the source, whereas newly authored tests require careful validation of their oracle, for which model-based judging is itself imperfect (Zheng et al., 2023). Source-derived designs scale by reusing implementations and tests but keep the source behaviour as the target, while independently authored requirements depart from it at substantially greater curation cost. `CommandBench` separates the roles: its CLI contract retains the breadth of real repositories, $\mathcal{T}^+$ certifies the source-backed usability floor, and every $\mathcal{T}^-$ case must be entailed by the released instruction yet failed by the source.

Contamination-resistant evaluation. Training-data studies find near-exact benchmark solutions in code corpora and show that performance rises sharply with training-set similarity (Riddell et al., 2024; Liang et al., 2026); semantically equivalent rephrasings further evade string-based decontamination (Yang et al., 2023), motivating both temporal freshness (Jain et al., 2025; Zhang et al., 2026a) and mechanism-based defences (Zhao et al., 2025; Zeng et al., 2025; Luo et al., 2026; Zhang et al., 2026b). These obstruct retrieval and copying yet leave the source semantics useful as the target, whereas `CommandBench` changes the behavioural target itself: a source-faithful replay scores zero by construction, a guarantee against direct replay only.

6 Conclusion

We presented `CommandBench`, a diverse, challenging, and contamination-resistant benchmark for end-to-end repository generation. Anchored on a language-agnostic CLI contract, it builds tasks from real repositories in many languages, and resists contamination through lexical obfuscation, cross-language implementation, and unmet-specification tests, under which faithful replay scores zero. No model we evaluate comes close to the ceiling. Scoring against the specification may generalise beyond this benchmark.

AI USE STATEMENT

AI assistants were used to polish the writing of this paper and to implement parts of the construction pipeline. Language models are also a component of the pipeline itself; that use is a method of this paper and is described in Section 2.

REFERENCES

Wasi Ahmad, Md Golam Rahman Tushar, Saikat Chakraborty, and Kai-Wei Chang. Avatar: A parallel corpus for java-python program translation. In *Findings of the Association for Computational Linguistics: ACL 2023*, pp. 2268–2281, 2023.

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

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

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

Jingzhe Ding, Shengda Long, Changxin Pu, Huan Zhou, Hongwan Gao, Xiang Gao, Chao He, Yue Hou, Fei Hu, Zhaojian Li, et al. Nl2repo-bench: Towards long-horizon repository generation evaluation of coding agents. *arXiv preprint arXiv:2512.12730*, 2025.

Yangruibo Ding, Zijian Wang, Wasi Ahmad, Hantian Ding, Ming Tan, Nihal Jain, Murali Krishna Ramanathan, Ramesh Nallapati, Parminder Bhatia, Dan Roth, et al. Crosscodeeval: A diverse and multilingual benchmark for cross-file code completion. *Advances in Neural Information Processing Systems*, 36:46701–46723, 2023.

Yue Hu, Yuzhu Cai, Yaxin Du, Xinyu Zhu, Xiangrui Liu, Zijie Yu, Yuchen Hou, Shuo Tang, and Siheng Chen. Self-evolving multi-agent collaboration networks for software development. In *International Conference on Learning Representations*, volume 2025, pp. 23007–23039, 2025.

Naman Jain, 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*, volume 2025, pp. 58791–58831, 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*, volume 2024, pp. 54107–54157, 2024.

Shanchao Liang, Spandan Garg, and Roshanak Zilouchian Moghaddam. The swe-bench illusion: When state-of-the-art llms remember instead of reason. In *Proceedings of the IEEE/ACM 48th International Conference on Software Engineering: Software Engineering in Practice*, pp. 395–405, 2026.

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. *Advances in neural information processing systems*, 36:21558–21572, 2023.

Jingyao Liu, Chen Huang, Zhizhao Guan, Wenqiang Lei, and Yang Deng. E2edev: Benchmarking large language models in end-to-end software development task. In *Proceedings of the 64th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pp. 35032–35068, 2026.

Tianyang Liu, Canwen Xu, and Julian McAuley. Repobench: Benchmarking repository-level code auto-completion systems. In *International Conference on Learning Representations*, volume 2024, pp. 47832–47850, 2024.

Jane Luo, Xin Zhang, Steven Liu, Jie Wu, Jianfeng Liu, Yiming Huang, Yangyu Huang, Chengyu Yin, Ying Xin, Yuefeng Zhan, et al. Rpg: A repository planning graph for unified and scalable codebase generation. In *International Conference on Learning Representations*, volume 2026, pp. 91932–91985, 2026.

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.

Martin Riddell, Ansong Ni, and Arman Cohan. Quantifying contamination in evaluating code generation capabilities of language models. In *Proceedings of the 62nd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pp. 14116–14137, 2024.

Baptiste Roziere, Marie-Anne Lachaux, Lowik Chanussot, and Guillaume Lample. Unsupervised translation of programming languages. *Advances in neural information processing systems*, 33: 20601–20611, 2020.

SST. opencode: The AI coding agent built for the terminal. https://github.com/sst/opencode, 2025.

Xingyao Wang, Boxuan Li, Yufan Song, Frank F Xu, Xiangru Tang, Mingchen Zhuge, Jiayi Pan, Yueqi Song, Bowen Li, Jaskirat Singh, et al. Openhands: An open platform for ai software developers as generalist agents. In *International Conference on Learning Representations*, volume 2025, pp. 65882–65919, 2025.

Weixiang Yan, Yuchen Tian, Yunzhe Li, Qian Chen, and Wen Wang. Codetransocean: A comprehensive multilingual benchmark for code translation. In *Findings of the Association for Computational Linguistics: EMNLP 2023*, pp. 5067–5089, 2023.

John Yang, Carlos Jimenez, Alexander Wettig, Kilian Lieret, Shunyu Yao, Karthik Narasimhan, and Ofir Press. Swe-agent: Agent-computer interfaces enable automated software engineering. *Advances in Neural Information Processing Systems*, 37:50528–50652, 2024.

Shuo Yang, Wei-Lin Chiang, Lianmin Zheng, Joseph E Gonzalez, and Ion Stoica. Rethinking benchmark and contamination for language models with rephrased samples. *arXiv preprint arXiv:2311.04850*, 2023.

Zhengran Zeng, Yixin Li, Rui Xie, Wei Ye, and Shikun Zhang. Benchmarking and studying the llm-based agent system in end-to-end software development. *arXiv preprint arXiv:2511.04064*, 2025.

Linghao Zhang, Shilin He, Chaoyun Zhang, Yu Kang, Bowen Li, Chengxing Xie, Junhao Wang, Maoquan Wang, Yufan Huang, Shengyu Fu, et al. Swe-bench goes live! *Advances in Neural Information Processing Systems*, 38, 2026a.

Zhaoxi Zhang, Yiming Xu, Jiahui Liang, Weikang Li, Xiaoshuai Chen, Liwei Qian, Xin Pei, Jizhou Huang, Run Sun, and Yunfang Wu. Repozero: Can llms generate a code repository from scratch? *arXiv preprint arXiv:2605.07122*, 2026b.

Wenting Zhao, Nan Jiang, Celine Lee, Justin Chiu, Claire Cardie, Matthias Gallé, and Alexander Rush. Commit0: Library generation from scratch. In *International Conference on Learning Representations*, volume 2025, pp. 12061–12076, 2025.

Lianmin Zheng, Wei-Lin Chiang, Ying Sheng, Siyuan Zhuang, Zhanghao Wu, Yonghao Zhuang, Zi Lin, Zhuohan Li, Dacheng Li, Eric Xing, et al. Judging llm-as-a-judge with mt-bench and chatbot arena. *Advances in neural information processing systems*, 36:46595–46623, 2023.

Appendix Contents

A The Unmet-Specification Suite $\mathcal{T}^-$ **12**

A.1 Form and Strictness of Assertions 12

A.2 Automatic Verification 12

A.3 Human Verification 13

A.4 Baseline Calibration 13

A.5 Sources of Variation 13

B Execution and Scoring **14**

C Task List **14**

A The Unmet-Specification Suite $\mathcal{T}^-$

A.1 Form and Strictness of Assertions

Every case consists of a stretch of input preparation followed by a number of assertions, written in the style of pytest, which may constrain the exit code, stdout and stderr, the files a command writes, and changes to the state of the workspace. An assertion constrains those parts of these quantities that $\mathcal{I}$ prescribes, rather than the expected output in full. The wording of an error message, the order of the fields of a JSON object and the format of a log lie outside the assertions. Floating-point values are compared with a relative tolerance.

A.2 Automatic Verification

Automatic verification addresses satisfiability and inferability. Satisfiability requires that an implementation exist which passes every test case in $\mathcal{T}^+$ and $\mathcal{T}^-$ at once; inferability requires that the behaviour asserted by every test case be determined by $\mathcal{I}$. The two properties call for different evidence, and we therefore use two probes.

We use a **reference implementation** $\mathcal{C}^\star$ as the probe for satisfiability. $\mathcal{C}^\star$ is obtained by having an agent repair the boundary behaviour of $\mathcal{C}$ one case at a time, with the full content of both suites in view. A repair is bound by three constraints: it stays within the implementation of the command it concerns and may touch neither the wrapper nor the test harness; it may not branch on a particular input value, nor inline any input or expected output that appears in a test; and once it is done $\mathcal{T}^+$ is rerun and must still pass in full. Whenever $\mathcal{T}$ is revised, we rerun the affected assertions against $\mathcal{C}^\star$.

We use **pilot models** as the probe for inferability. Three pilot models from different families implement the task from an empty repository on the basis of $\mathcal{I}$ alone; their environment and execution protocol match the evaluation proper, except for a time budget of twelve hours rather than six, which keeps the signal "every pilot model fails" from being confounded with running out of time. Since revising $\mathcal{I}$ changes the premises from which an assertion is inferred, we have all three pilot models reimplement every task in each round.

A case is healthy exactly when the reference implementation passes it and at least one pilot model passes it. Every other case goes to an independent agent, which judges whether the defect originates in the task: it has no access to the source repository, receives the assertion, the evidence the assertion cites and the execution output of both probes, and must supply the reasoning behind its judgement. Where the judgement is that the task is at fault, we revise $\mathcal{T}$ or $\mathcal{I}$ accordingly and enter the next round; the loop runs for three rounds, and any case still judged to originate in the task after the third is deleted.

After the third round, $\mathcal{C}^\star$ passes every case of both suites on every task. Human verification then addresses two remaining questions: whether the repairs in $\mathcal{C}^\star$ generalise, and whether the assertions that no pilot model passes are inferable.

A.3 Human Verification

Human verification likewise addresses the two properties, with one check for each. Both checks were carried out by ten annotators holding an undergraduate degree or higher in computer science.

The first check asks whether the repairs in the reference implementation generalise. $\mathcal{C}^\star$ passes every case under automatic verification, but it may pass by branching on a particular input rather than by handling the boundary condition in general terms. We drew a random sample of 200 repairs, 16.7% of all repairs, and had annotators read each change together with the assertion it answers and decide whether it handles the entire class of inputs that the boundary condition covers, or only the particular instance appearing in the test. Three repairs, 1.5% of the sample, were judged to be the latter and were sent back to be repaired again.

The second check asks whether the assertions of $\mathcal{T}^-$ are inferable. For every task we sampled up to three assertions that no pilot model passes, 316 in total, or 21.1% of the assertions of that kind. For each assertion we hid the value it asserts and kept only the quantity it asks about, and two annotators answered independently: they could search the whole of $\mathcal{I}$ and could see the inputs of the case, but neither the asserted value, nor the source repository, nor any implementation, nor each other's answers. A single answer that disagrees with the assertion marks it as in need of repair.

Of the sampled assertions, 304 were judged inferable, 96.2% of the sample, while 12, or 3.8%, needed repair and were corrected by hand before freezing.

A.4 Baseline Calibration

Two conditions that do not depend on model capability check that the two suites behave as the construction requires, both run through the complete evaluation pipeline of Appendix B. *Upstream replay* submits the wrapped repository as it stands; by construction it passes every case in $\mathcal{T}^+$ and fails every case in $\mathcal{T}^-$, so it must score exactly zero. *Empty implementation* produces no artefacts at all and scores zero as well. Table 4 reports both. Upstream replay passes $\mathcal{T}^+$ in full and $\mathcal{T}^-$ not at all, so the wrapper, the translator and the language environment preserve the guarantee end to end; both anchors hold exactly at 0.00.

Table 4: Baseline calibration. *Upstream replay* submits the wrapped repository as it stands and *empty implementation* produces no artefacts; both anchor the bottom of the score range.

Condition	S	$\mathcal{T}^+$ **(%)**	$\mathcal{T}^-$ **(%)**
Upstream replay	0.00	100.00	0.00
Empty implementation	0.00	0.00	0.00

A.5 Sources of Variation

Human verification can only cover a sample: checking every assertion one by one is infeasible given how many of them there are and how expensive qualified annotators are, and defects may therefore remain in the suites. This section measures separately how the stochasticity of the model calls and how any such residual defect affect the evaluation.

We repeat the full evaluation of DeepSeek-V4.1 Flash three times on the frozen benchmark. The three runs use identical environments, budgets and protocols, so their differences arise only from the stochasticity of the model calls and of the resulting agent trajectories; the standard deviation of S across runs is 0.37 (Table 5, upper block).

A residual defect renders the verdict of an assertion untrustworthy, an effect we simulate by removing assertions at random: from the $\mathcal{T}^-$ of every task we remove a fixed fraction of the assertions and resample repeatedly. Removing a fifth of them—far more than the human verification of Appendix A.3 leaves room for—the standard deviation of S is 0.28 and the model ordering has a Spearman ρ of 0.994 against the ordering obtained on the full suites (lower block).

Here $\sigma(S)$ measures how far a score moves and ρ whether that movement can overturn a comparison. A residual defect is not specific to any one model: the same suites score them all, so its effect is

closer to a shared shift than a reordering. Even at a fifth, $\sigma(S)$ stays below that of three repetitions under an identical protocol, and the ordering is almost always recovered intact.

Table 5: Sources of variation in S: three repetitions of the full evaluation of DeepSeek-V4.1 Flash under an identical protocol (upper block), and random removal of a fifth of the assertions of every $\mathcal{T}^-$ (lower block).

Condition	S	$\sigma(S)$	ρ
Repeated evaluation			
Run 1	52.47	—	—
Run 2	52.98	—	—
Run 3	52.27	—	—
Across runs	52.57	0.37	—
Random assertion removal			
20% removed	52.47	0.28	0.994

B Execution and Scoring

Each task runs in a Docker container built from the task's Dockerfile. The image carries the toolchain of the target language together with the basic libraries a command-line program needs, namely CLI argument handling, numerical routines and I/O, while the workspace holds the fixtures the instruction requires and nothing else. Each container is allotted four CPU cores, 16 GB of memory, and a six-hour wall-clock budget that includes model inference. As in other end-to-end repository generation benchmarks such as Commit0 (Zhao et al., 2025) and NL2Repo-Bench (Ding et al., 2025), the network is available while the agent works; it is unavailable while the suites run, and everything outside the workspace is read-only. The agent sees only the instruction $\mathcal{I}$ and those fixtures: the suites $\mathcal{T}^{\pm}$, the de-obfuscating translator and the source repository are never present in the container.

Every model is driven by the same scaffold, `opencode` (SST, 2025), and queried at a fixed, dated API snapshot with provider-default hyperparameters and no per-task tuning. Each task is attempted once. A run ends when the agent declares the task complete; if the budget expires first, whatever the workspace holds at that moment is submitted as it stands. The instruction states how each command is to be invoked, and the agent is responsible for delivering an artefact that can be driven in exactly that way.

The suites are injected into the submitted repository only after the agent has finished, so no run can observe them. Each case then executes from a fresh container state, in a fixed order, with no state carried between cases; a submission that fails to build scores zero rather than being dropped, and each case is executed three times with the verdict required to agree. All choices above were fixed before inspecting model outcomes.

Table 6 collects the definition and unit of every column that the tables of the main text report.

C Task List

Table 7 lists all 138 tasks. Every row pins the source revision the task was derived from, so the benchmark can be rebuilt exactly as evaluated; the obfuscation mapping $\mathcal{M}$ and the two suites $\mathcal{T}^{\pm}$ are withheld.

Table 7: Every task in `CommandBench`, grouped by discipline and category. Commit pins the exact source revision the task was built from. Src and Tgt are the language of that source and the language the implementation is requested in. $|\mathcal{T}^{+}|$ and $|\mathcal{T}^{-}|$ count the hidden met- and unmet-specification cases, Instr. is the size of the released instruction in thousands of characters, and Fixt. is the fixture payload shipped inside the environment; a dash marks a task whose inputs are built from command-line arguments alone.

Table 6: Definition and unit of every column reported in the main tables.

Column	Definition and unit
Effort	The reasoning effort the model is queried at, named as its provider names it; every model is run at the highest setting available to it.
S	Task score of Equation 2, averaged over tasks.
$\mathcal{T}^+$ (%)	Mean task-level pass rate on the suite that the wrapped repository passes.
$\mathcal{T}^-$ (%)	Mean task-level pass rate on the suite that it fails.
Id. overlap	Lexical overlap between the submission and the upstream repository at its recorded commit. From each side we take four kinds of token: identifiers of at least three characters whose lower-case form is absent from a 256-word stop list of language keywords and the commonest library names; the fragments those identifiers split into at camel-case and underscore boundaries, filtered the same way; long numeric literals, being hexadecimal of more than two digits, reals with more than two decimal places, or integers of at least four digits, which leaves out the ubiquitous 0, 1 and 256; and string literals of at least four characters that do not begin with `http`, `/` or `-`. Build output, third-party dependencies and any test, documentation or example directory are skipped on both sides. Each kind is compared as a set, by presence and not by frequency, through the Jaccard coefficient $\lvert A \cap B \rvert / \lvert A \cup B \rvert$; the four are averaged with equal weight to give the overlap of one task, and those are averaged over the tasks for which the model submits.
Time (min)	Per-task average of the wall-clock minutes the agent runs before submitting, inference included, capped by the budget of Appendix B.
In, Out	Per-task averages of the input and of the output tokens consumed.
Tool calls	Per-task average of all tool invocations.
( )	A signed difference from Table 2, in the unit of the column it qualifies.

Repository	Commit	Src	Tgt	$\lvert\mathcal{T}^+\rvert$	$\lvert\mathcal{T}^-\rvert$	Instr.	Fixt.
Computer Science (71 tasks)							
Algorithms and Solving (7)							
cadical	`c607304`	C++	Rust	499	42	36.7k	334 KB
datasketches-cpp	`de8553b`	C++	Java	291	31	39.9k	70 KB
libfsm	`74f34b3`	C	Go	261	34	45.6k	104 KB
libminizinc	`c9f9b64`	C++	Python	270	30	26.6k	1.0 MB
libsolv	`6aa632a`	C	Go	277	61	36.8k	25 KB
networkx	`9fca49c`	Python	Java	290	38	59.0k	–
regex	`7c28ee4`	Rust	Go	432	35	50.1k	–
Fonts and Typesetting (5)							
afdko	`cef990e`	Python	Java	208	25	33.0k	8.6 MB
fonttools	`95b28d3`	Python	C#	248	34	45.7k	1.4 MB
harfbuzz	`ab5ecbb`	C++	Java	269	35	29.8k	21.7 MB
taffy	`45a5629`	Rust	Go	148	42	37.5k	–
typst	`9dfd3a0`	Rust	Java	350	59	32.2k	17.5 MB
Formats and Codecs (7)							
csv-parser	`32e99be`	C++	Rust	325	38	39.5k	55.8 MB
go-yaml	`92bc79c`	Go	Python	314	48	35.7k	53 KB
icalendar	`138c845`	Python	C#	303	75	56.9k	29 KB
libxml2	`c94eb02`	C	C#	362	63	47.5k	1.6 MB
pyelftools	`52ce435`	Python	Go	232	34	60.9k	4.5 MB
rust-brotli	`9651aa3`	Rust	JS	285	43	13.7k	612 KB
XlsxWriter	`e943bee`	Python	C#	254	58	29.6k	792 KB
Geometry and Space (7)							
cgal	`cac3e9d`	C++	Python	298	36	38.1k	18.3 MB
Clipper2	`21ebba0`	C++	C#	156	63	26.8k	–

Repository	Commit	Src	Tgt	$\lvert\mathcal{T}^+\rvert$	$\lvert\mathcal{T}^-\rvert$	Instr.	Fixt.
h3	1b536c3	C	Rust	237	28	38.2k	14.0 MB
kurbo	838b69e	Rust	C++	273	30	45.5k	–
PDAL	27008f6	C++	Python	179	27	50.7k	10.3 MB
proj4js	9abbc38	JS	Python	223	88	15.2k	179 KB
trimesh	cdb90eb	Python	C#	242	61	52.1k	12.7 MB
Languages and Compilers (9)							
acorn	365d729	JS	Go	275	62	37.2k	–
black	87928e6	Python	C++	542	35	33.4k	106 KB
capstone	0225758	C	Go	233	69	33.5k	–
markdig	fc70523	C#	Java	365	20	60.4k	–
pigeon	51359a4	Go	Rust	202	34	20.1k	115 KB
revm	fbb0028	Rust	Python	211	31	41.2k	–
sh	2f3f5e3	Go	Rust	446	29	53.2k	7 KB
Simple	2820970	Java	Go	322	32	17.4k	–
tinycc	2ba12e8	C	Rust	290	109	20.6k	92 KB
Media and Imaging (6)							
flac	1507800	C	Java	218	27	19.9k	17 KB
libjxl	a7a9c78	C++	JS	219	24	43.7k	69.4 MB
libpng	3061454	C	JS	208	37	27.9k	69 KB
Little-CMS	21c582a	C	C#	120	22	39.7k	949 KB
OpenTimelineIO	4423671	C++	Go	146	23	35.4k	40 KB
zxing-cpp	287c85d	C++	Python	228	21	13.6k	2 KB
Networking and Protocols (6)							
dnspython	9350a91	Python	C#	325	25	51.5k	14 KB
enmime	5314698	Go	C++	272	38	33.9k	70 KB
h2	73f39d4	Rust	Go	121	39	16.9k	–
nfdump	e198c8b	C	Python	276	31	42.3k	–
souin	fd6fa53	Go	Java	275	48	39.3k	2 KB
stun	7b40f80	Go	Python	196	49	22.3k	6 KB
Other (3)							
bat	979ba22	Rust	C++	253	35	22.3k	1.1 MB
nodatime	381de98	C#	Go	340	20	42.2k	42 KB
readability	04fd32f	JS	Python	371	25	17.9k	4.9 MB
Query and Data (8)							
bleve	d8f2ab9	Go	Python	217	38	39.7k	–
datafusion	d555234	Rust	Python	380	36	52.3k	701 KB
jq	34f7186	C	Python	375	53	29.2k	1 KB
jsonschema	c87a922	Python	Java	298	61	28.0k	–
sqlglot	9a8129b	Python	C++	273	58	32.0k	–
sqlite	b09c88c	C	Python	283	24	33.3k	–
xarray	0238035	Python	Rust	238	31	60.1k	–
yjs	1ce38f7	JS	Go	207	43	42.8k	–
Security and Cryptography (4)							
certificate-transparency-go	e8f9317	Go	C++	219	38	44.1k	18 KB
DOMPurify	4e6fe24	JS	Go	192	61	7.0k	–
in-toto	c82fe5d	Python	JS	238	50	37.9k	12 KB
zlint	9483582	Go	Java	210	29	24.5k	2.0 MB
Systems and Storage (9)							
coreutils	28b6856	Rust	JS	391	30	41.0k	4 KB
gitoxide	842bc44	Rust	C++	287	27	45.5k	2.3 MB
go-containerregistry	7b32099	Go	Java	292	55	50.3k	18 KB
leveldb	99b3c03	C++	Rust	171	42	19.5k	–
libriscv	4de56ec	C++	Rust	227	75	27.1k	1.5 MB
lmdbjava	f5af0c7	Java	Python	142	29	30.3k	–
ninja	3441b63	C++	Rust	266	26	40.0k	2 KB
restic	6aa3a51	Go	JS	149	23	33.1k	8.3 MB
rsync	471e17d	C	JS	449	27	44.6k	1 KB
Other Disciplines (67 tasks)							
Biology (7)							
covasim	1a48b88	Python	C++	155	35	50.8k	1.4 MB
cutadapt	ef85262	Python	Go	270	31	50.2k	45 KB
htslib	666463c	C	Go	288	42	45.3k	46.0 MB
libsbml	e65f60d	C++	Java	282	35	59.2k	82 KB
muscle	2cf9d33	C++	Go	210	48	37.7k	28.4 MB

Repository	Commit	Src	Tgt	$\lvert\mathcal{T}^+\rvert$	$\lvert\mathcal{T}^-\rvert$	Instr.	Fixt.
plink-ng	22df061	C	C#	186	41	35.4k	458 KB
pydicom	20aa4b7	Python	Java	226	41	48.7k	1.0 MB
Chemistry (8)							
cantera	4a8358e	C++	Java	237	37	50.0k	744 KB
cdk	3bc6efe	Java	C#	303	33	40.1k	252 KB
chempy	438f2e7	Python	Java	245	42	47.6k	–
gemmi	5cc1c23	C++	Python	239	37	44.8k	277 KB
openbabel	0e94434	C++	Rust	253	61	35.5k	4.3 MB
pycalphad	2d4c744	Python	C++	167	57	38.1k	4.0 MB
pyteomics	a118ee8	Python	Java	288	35	48.2k	–
spglib	12355c7	C	Go	221	38	33.5k	40 KB
Economics, Law and Politics (8)							
beancount	eeda2aa	Python	C#	296	52	49.8k	79 KB
bt	59a78d7	Python	Go	173	38	31.0k	2 KB
openfisca-core	09d008d	Python	C++	305	46	44.8k	8 KB
PyPortfolioOpt	0186e70	Python	C++	135	89	47.1k	535 KB
QuantEcon.py	20496bf	Python	C#	199	43	42.5k	–
statsforecast	264166e	Python	C#	231	34	42.6k	–
ta-lib-python	a9ff1b4	Python	Java	230	70	26.6k	5 KB
Tax-Calculator	c55400e	Python	Java	132	45	44.3k	24.2 MB
Engineering (8)							
EPANET	444725f	C	Rust	215	74	55.7k	8 KB
gdsfactory	9da61a2	Python	C++	24	47	47.1k	50 KB
gprMax	368e7c1	Python	Rust	158	38	29.8k	745 KB
idyntree	40b6568	C++	C#	333	94	49.0k	58 KB
iverilog	dfeee90	C++	Rust	175	58	16.0k	1.3 MB
ngspice	a80f6e3	C	C#	201	49	42.4k	46 KB
Pynite	438e536	Python	Rust	215	31	44.1k	–
scikit-rf	3465272	Python	C++	282	39	51.4k	3.1 MB
Games (4)							
pachi	f86fc61	C	Python	268	23	26.4k	16.3 MB
python-chess	3516d7c	Python	Rust	385	32	46.6k	5.5 MB
recastnavigation	6dc1667	C++	Python	215	32	32.4k	186 KB
Stockfish	cb3d4ee	C++	Python	165	151	20.5k	80.2 MB
Geography (7)							
eccodes	cbe16e9	C	Rust	315	33	36.8k	10.0 MB
esda	6ffd5b6	Python	Go	199	45	53.3k	7.5 MB
GSTools	1edd2b9	Python	C++	189	48	44.3k	–
libpysal	40cab61	Python	C++	208	38	48.4k	2.1 MB
MetPy	433bdd1	Python	C++	167	33	51.6k	151.0 MB
obspy	0fd195e	Python	Java	272	34	50.3k	2.2 MB
PROJ	f08fa86	C	JS	166	40	36.1k	1.9 MB
Linguistics (6)							
espeak-ng	4870adf	C	Python	270	28	24.2k	9.2 MB
hunspell	c5f9815	C++	Python	195	31	24.2k	275 KB
jieba	1e20c89	Python	Go	303	35	26.8k	11.1 MB
Pyphen	ed0bae0	Python	JS	138	35	15.4k	2.1 MB
python-ftfy	5340af6	Python	Java	732	25	20.6k	13 KB
ucto	72752b9	C++	C#	234	46	21.2k	87 KB
Mathematics (6)							
gap	1facc20	C	Go	413	30	37.2k	28.9 MB
HiGHS	04024d7	C++	Rust	158	30	43.8k	921 KB
lifelines	a21e432	Python	C#	301	71	61.8k	169 KB
POT	9932112	Python	Java	160	42	25.8k	–
pycma	9f4efc0	Python	C#	198	42	17.6k	–
SuiteSparse	4d40960	C	Rust	224	51	35.1k	223 KB
Music (6)							
aubio	90bd27a	C	Python	186	35	20.9k	11.8 MB
chromaprint	aed8eba	C++	C#	147	22	13.7k	554 KB
midi-file	f97a029	TS	C++	137	60	11.9k	–
music21	98a795e	Python	C#	262	27	45.4k	12.1 MB
pyroomacoustics	f02b01d	Python	C#	213	63	40.6k	130.2 MB
tonal	94b4d62	TS	Python	399	58	52.1k	3 KB
Other (1)							

Repository	Commit	Src	Tgt	$\lvert\mathcal{T}^{+}\rvert$	$\lvert\mathcal{T}^{-}\rvert$	Instr.	Fixt.
citeproc-js	7c8301a	JS	Go	244	40	28.5k	223 KB
Physics (6)							
bullet3	2c204c4	C++	Rust	238	32	61.7k	218 KB
healpy	fc67691	Python	C++	269	66	51.1k	1.3 MB
photutils	a89accd	Python	Go	215	39	46.4k	–
pyhf	b500270	Python	C++	238	57	60.2k	8 KB
python-skyfield	8e8eb20	Python	Rust	190	31	54.2k	83.4 MB
uproot5	a7a99a0	Python	C++	323	35	40.8k	249.2 MB
Total (138)				**34,994**	**5,899**	**5.23M**	**1.20 GB**