TESTSUITEBENCH: CODING AGENTS STRUGGLE TO GENERATE STRONG TEST SUITES FROM SPECIFICATIONS

Anonymous authors
Paper under double-blind review

ABSTRACT

Coding agents are increasingly used to create large software projects. While code-generation benchmarks have scaled accordingly to full applications, test-generation benchmarks remain limited to individual functions or bug fixes. In particular, generating complete test suites for complex software from scratch remains unexplored. To address this gap, we introduce TestSuiteBench, a benchmark of 165 open-source applications, each with a structured and detailed specification of intended behavior, a reference implementation, a reference test suite, and on average 74 mutants. Each mutant is generated from a single requirement of the specification and refined to isolate that behavior: 73% are killed by exactly one reference test and, on average, by only 0.2% of the remaining tests. We find that even frontier coding agents struggle to generate strong test suites given our specifications, achieving mutation scores below 60%, despite their suites being largely valid and 88% of their tests passing on the reference implementation. We observe that generated test suites often lack relevant input combinations and are smaller than reference test suites. Meanwhile, implementations generated for the same specifications pass 82% of the reference tests. These results show agents can build software they cannot verify, a gap that directly limits fully autonomous software development.

1 INTRODUCTION

Coding agents have rapidly progressed from resolving isolated issues (Yang et al., 2024) to being used across the software engineering lifecycle (Mürtz & Müller, 2025; Huang et al., 2026). Increasingly, they re-implemented complex applications from scratch with little human involvement, including a C compiler and a web browser (Carlini, 2026; Lin, 2026). Systems like these took human teams years to build, suggesting that fully autonomous development of genuinely new software is within reach.

However, such autonomous development depends on strong verification that checks the agent's work against the intended behavior (Carlini, 2026). While re-implementations can rely on existing test suites and reference implementations for verification, those are not available for genuinely new software, which initially only has a natural language specification. The agent must thus be able to verify their implementation against this specification by translating it into a strong test suite.

Existing benchmarks do not measure this capability. Code generation benchmarks have scaled from resolving individual issues (Jimenez et al., 2024; Deng et al., 2025) to building complete applications (Ding et al., 2025; Yang et al., 2026; Lu et al., 2025; Vero et al., 2025). Test generation benchmarks, in contrast, remain limited to individual issues (Mündler et al., 2024; Sun et al., 2026), few files (Jain et al., 2025; Wang et al., 2026), or single scenarios (Kogler et al., 2026), while always providing the implementation to test. Building a benchmark closing this gap is challenging: ambiguity, inconsistency, and incompleteness of natural language are a main source of software testing errors (Mustafa et al., 2021), and measuring a test suite's strength requires faults that violate specific requirements, which generic rule-based mutations cannot guarantee (Jia & Harman, 2011).

Our work We introduce TESTSUITEBENCH, a benchmark of 165 open-source applications in which agents must write a complete test suite given only a natural-language specification (Figure 1). We derive each specification from the application's reference test suite, so that it describes every tested

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.

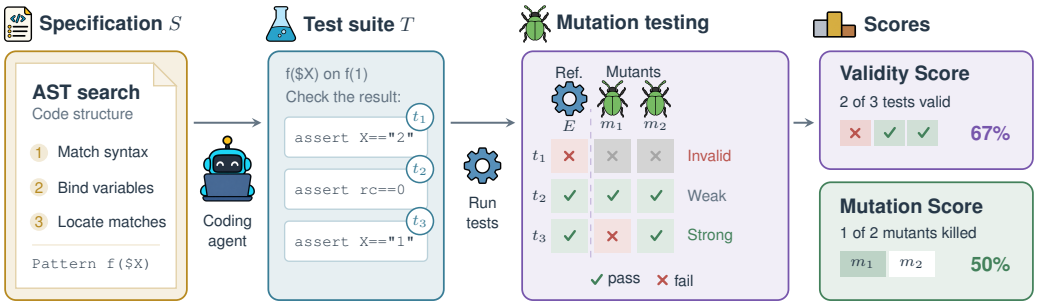


Figure 1: TESTSUITEBENCH: One instance consists of a specification S, a test suite T asserting its requirements, and a set of mutants M that break these requirements in targeted ways. An agent is tasked with generating a full test suite based solely on the natural-language specification S. We then evaluate the *Validity Score* $\mathcal{V}$, the fraction of tests that pass on the reference implementation, and the *Mutation Score* $\mathcal{M}$, the fraction of mutants killed by the generated test suite.

behavior, and iteratively refine it to remove ambiguities and contradictions. To measure a suite's strength, we construct on average 74 mutants per application, each violating a single requirement. We refine mutants to isolate the violated requirement, with 73% being killed by exactly one reference test. We report a generated suite's validity score $\mathcal{V}$ as the fraction of tests that pass on the reference implementation and its strength as the mutation score $\mathcal{M}$, the fraction of mutants killed by valid tests; since the reference suite kills every mutant, a perfect score is attainable by construction.

We evaluate coding agents with six recent frontier models and find that they struggle to write strong test suites: even the best agent's suites kill only 55% of mutants, and four of the six agents stay below 40%. This is not due to a lack of understanding of the specification: on average, 88% of generated tests are valid, and implementations built by the same agents from the same specifications pass up to 93% of the reference tests. Even when given the reference executable in addition to the specification, mutation scores remain unchanged, indicating that the bottleneck lies in test design rather than in the specification's precision. Indeed, generated suites are often too small and miss the input combinations needed to expose violated requirements.

These results reveal a critical gap: agents can build software they cannot verify. Until they can turn specifications into strong test suites, fully autonomous development of new software remains out of reach; TESTSUITEBENCH measures progress toward this goal.

Key Contributions of our work are:

- TESTSUITEBENCH, a benchmark for repository-scale test suite generation from natural-language specifications, comprising 165 applications with detailed specifications, reference implementations, reference test suites, and requirement-targeted mutants (Section 2).
- A scalable pipeline that derives detailed specifications from existing test suites, refines them by attributing agent failures to specification defects, and generates mutants that each violate a single requirement, enabling the benchmark to be extended to new applications (Section 2).
- An extensive evaluation of 6 coding agents, revealing that they can implement specifications they cannot test (Section 3).

2 CONSTRUCTING TESTSUITEBENCH

Notation We denote a repository as R, containing a reference implementation I^* from which a build script produces a reference executable E^*, and a reference test suite T^* that validates the correctness of I^* and E^*. Throughout, we mark reference artifacts with a star (e.g., T^*) and artifacts under evaluation with a hat (e.g., $\hat{T}$). A test suite T is a set of individual tests t. Typically, T can be considered an executable verifier that checks the alignment between an executable E and a specification S, which describes in natural language how E should behave. Mutants M are executables that have specific defects with respect to S and can be used to assess the strength of T.

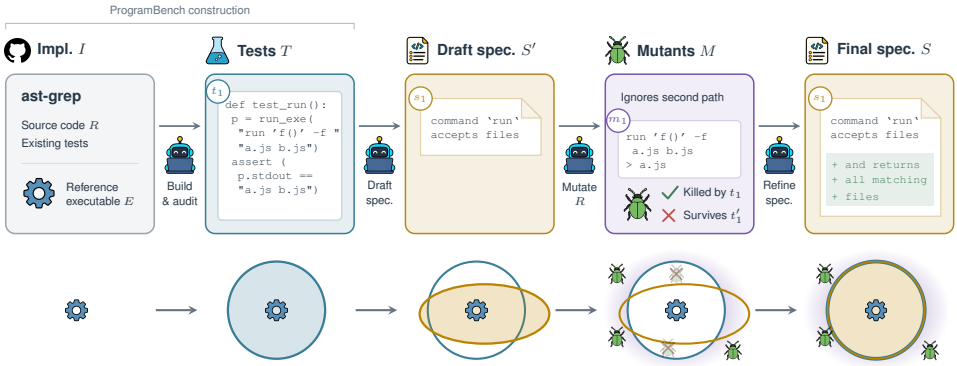


Figure 2: (Top) To construct TESTSUITEBENCH, we first obtain a reference test suites T^*, implementation I^* and its executable E^* from a GitHub repository R. We then create a draft specification S', and mutant candidates M that violate individual requirements, before refining the mutants to reduce unrelated failures against T^*. We then refine the specification S' by iteratively generating implementations and test suites from it, analysing resulting errors, and addressing the ones attributed to the specification. (Bottom) In the space of implementations, we first define the set of admissible implementations using T, and align it with the set according to the specification S.

We say a test t or a test suite T containing t *kills* an executable E if t fails when processing E. A high quality test suite never kills the reference executable E^*, while killing all mutants M.

Benchmark Construction We build a TESTSUITEBENCH instance in three conceptual steps, described below. We first create a detailed draft specification from the reference test suite (Section 2.1). Then we systematically construct a set of mutants from this draft specification and the reference test suite that each get killed by a single test (Section 2.2), before finally polishing the specification iteratively to remove any remaining ambiguities and contradictions (Section 2.3).

2.1 GENERATING A DRAFT SPECIFICATION

We manually inspected developer-written documentations for representative repositories and found them severely underspecified and thus unsuitable for test generation (see Table 5). Thus, we create richer specifications suitable for test and program generation. To this end, we first refine a test suite to remove low quality tests before generating a specification document from it. Throughout, we use Codex GPT-5.6 Sol (OpenAI, 2026a) with high reasoning whenever coding agent is mentioned.

Reference Implementation We start by collecting a reference implementation I^* building on the 200 instances constituting PROGRAMBENCH (Yang et al., 2026). We summarize their initial construction process below.

1. Collect 200 open-source GitHub repositories R with compiled standalone executables E^*, based primarily on Rust, Go or C++.
2. Instruct a coding agent to construct an extensive test suite T^* based on the test suite contained in R and the observed behavior of E^*. All tests in T^* are purely input-output behavior assertions.
3. Manually and automatically audit the constructed test suite for underspecification or overspecification with respect to E^*.

Test Suite Refinement We refine the PROGRAMBENCH test suites by filtering out tests that require internet access, excessive resources, or are flaky w.r.t. the reference implementation, i.e. produce inconsistent results across reruns. See App. B for more details. If, after this filtering, the size of T^* has been significantly reduced, we drop the instance. Overall, this leads to a set of 165 initial instances with reference implementation and test suite. Among those instances, the reference test suites are shrunken by 5.7% on average.

Draft Specification Generation We use a coding agent to generate a semi structured specification draft S' describing every tested behavior of T^* explicitly with at most 50 sections. This initial draft will later be iteratively refined to remove ambiguities, contradiction, and under specification w.r.t. input-output behaviour.

2.2 MUTANT CONSTRUCTION

The set of mutants are at the core of TESTSUITEBENCH and their specificity and diversity determine its ability to measure a test suites strength. We thus aim to create mutants with targeted changes that break only specific requirements, while being otherwise indistinguishable. For each mutant, we select a target test function t_{target} that is intended to kill it, and denote additional tests $T^* \setminus \{t_{target}\}$ that kill it as $\bar{T}$. We aim to construct mutants that are killed by exactly t_{target} and have an empty $\bar{T}$.

As T^* is often very large (median size of 679), we target only a subset of test functions, selected to uniformly cover S'. This is sufficient for strong mutation testing, since the later test generation can not know which requirements will be tested by the mutants and thus has to evenly cover the entire specification. We generate the actual mutants using the following two-step approach.

Candidate Construction We create a bi-graph G linking each section s_i of S' describing a set of requirements to the test functions $t_j \in T^*$ asserting them. Based on G, we then pick 6 test functions $t_{j,target}$ per specification section s_i, such that each test is selected by as few sections as possible. This can be described and solved optimally as a min-cost flow problem. See App. B.2 for more details.

We then use a separate coding agent for every $t_{j,target}$ to modify I^* into I_j such that it compiles into a mutant executable m_j killed by $t_{j,target}$ due to a violation of a specific behavior in s_i. The agent is instructed to retry the process in case I_j does not compile up to three times.

Candidate Refinement For every mutant candidate, we determine the test functions that kill it and rank the 6 mutants m_j associated with each s_i by (i) whether m_j is killed by $t_{j,target}$ and (ii) the size of $\bar{T}_j$. We select the top three mutants by these criteria. Then, we use a coding agent to inspect these mutant candidates and the test cases that kill it, and refine them to reduce $\bar{T}_j$ where needed.

Finally, we run the reference test suite against the created mutants and drop mutants that are either (i) not killed by the intended test function $t_{j,target}$, or (ii) killed by $> 20\%$ of unrelated functions. This refinement process significantly increases the fraction of mutants killed exactly by the targeted test ($32.4\% \rightarrow 73.0\%$) and reduces the average size of $\bar{T}_j$ ($1.45\% \rightarrow 0.17\%$).

2.3 SPECIFICATION REFINEMENT

Finally, we refine the draft specifications to remove contradictions and ambiguities while ensuring that the specifications contain enough detail for test and program generation. To this end, we run an iterative process, creating tests and implementations based on the specification, analysing resulting failures, and remedying any errors attributed to the specification.

Trial Generations We use two coding agents to generate an implementation $\hat{I}$ and a test suite $\hat{T}$ based on the draft specification S'. We then evaluate $\hat{I}$ against the reference tests T^*, and $\hat{T}$ against the reference executable E^* and our mutant set M.

Error Attribution For each failure ($t \in T^*$ killing $\hat{I}$, $\hat{t} \in \hat{T}$ killing I^*, and $m \in M$ surviving $\hat{T}$), we classify whether the failure is due to a bad generation, or the underlying specification S', e.g., underspecification (e.g., ambiguous, insufficient) or contradiction (e.g., misleading, wrong) with respect to T.

Specification Refinement For every failure attributed to S', we refine the affected sections of the specification using an agent. Concretely, for contra-

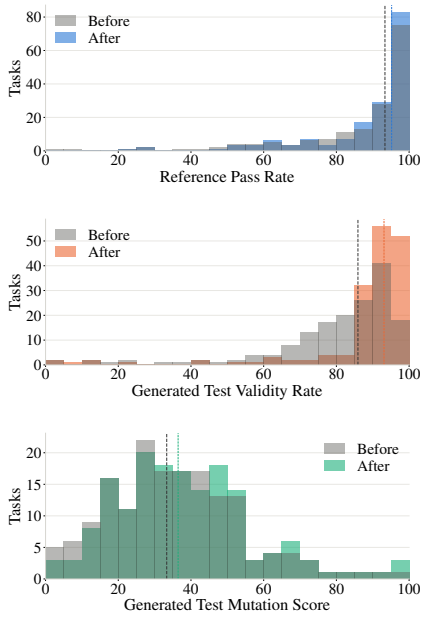


Figure 3: The iterative refinement increases the implementation's reference pass rate and test's validity rate, while mutation score remains low.

dictory information we expect a correction of the specification, and for an underspecification, we expect further details to be inserted based on the reference tests.

We iterate the above three steps until fewer than 5% of generated test failures were attributed to the specification. We visualize the improvement process in Figure 3. We see that in particular the test validity rate increased significantly as ambiguities in expected behaviour were clarified.

2.4 Manual Quality Control

Finally, we check the resulting specifications, mutants, and reference test suites manually, fixing minor issues such as tests from the original ProgramBench suite expecting crashes in specific lines. We discuss 34 instances where such manual intervention was required in App. B.

2.5 TestSuiteBench

Our final benchmark consists of 165 instances, spanning Rust, Go, and C/C++ and a diverse set of domains including text processing, file and disk utilities, and build tools. We show key statistics in Table 1, and a programming language breakdown in Figure 4. Our natural language specifications contain a median of 5k words across 23 sections, 679 reference test functions T^*, and 67 mutants M. The reference implementation length varies significantly from 538 to 1.6M LoC with a median of 7k.

Table 1: TestSuiteBench Statistics across benchmark instances.

		Min	Max	Median
S	# Words	2,065	27,406	5,635
	# Lines	158	3,258	576
	# Sections	10	50	23
	# Files	1	46	1
I	# LOC	538	1,613,380	7,008
T	# Func	85	13,298	679
M	# Amount	27	148	67

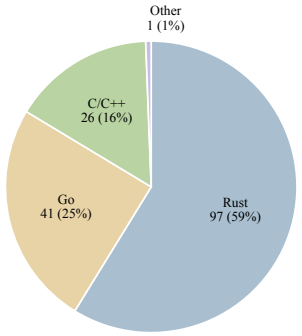


Figure 4: Implementation languages of TestSuiteBench instances.

2.6 Evaluation Metrics

We measure the quality of a test suite $\hat{T}$ with two core metrics: First, its *Mutation Score ($\mathcal{M}$)* is the fraction of mutants M that are killed by $\hat{T}$. Second, its *Validity Rate ($\mathcal{V}$)* is the fraction of generated tests $\hat{t} \in \hat{T}$ that do not kill the reference executable E^*. Importantly, since test functions that kill all executables, would achieve artificially high $\mathcal{M}$, we filter out all tests t that fail on the reference executable E^* when calculating $\mathcal{M}$. To handle flaky tests that kill non-deterministically, we run test suites 3 times on the reference executable and remove tests for which behavior changes across runs.

For implementations $\hat{I}$, we measure correctness using the *Reference Pass Rate ($\mathcal{R}$)*, the fraction of reference tests $t \in T^*$ that do *not* kill $\hat{I}$.

To prevent cheating, agents are run inside Docker containers without networking and CLI search tools disabled but all standard libraries including the dependencies of the reference implementation pre-installed.

3 Experimental Evaluation

Models and Setup We consider GPT-5.6 Sol and Luna (OpenAI, 2026a), GPT-6 Astra (OpenAI, 2026b), Claude Opus 5 (Anthropic, 2026), Gemini-3.8 Flash (Google DeepMind, 2026), DeepSeek V4 Flash (DeepSeek-AI et al., 2026). Unless otherwise indicated, we use the default reasoning effort and vendor-recommended agent harnesses, enforcing a 6h timeout on agents. We follow the inference and evaluation setup outlined in Section 2.5 and provide additional details in App. C.

3.1 Main Results

Agents Struggle to Generate Strong Tests. We compare different coding agents across test and program generation in Table 2 and observe that they all struggle to generate strong test suites, as measured by their mutation score $\mathcal{M}$. Claude Opus 5 achieves the highest score at 55.5%, at a cost

Table 2: Frontier agents struggle to achieve $\mathcal{M}$ beyond 56%, even when $\mathcal{V}$ and $\mathcal{R}$ are almost 100%.

	Test Generation				
Model	$\mathcal{M}$	$\mathcal{V}$	$	t (min)	$\mathcal{R}$
Claude Opus 5	**55.5**	**96.1**	10.56	37.0	91.8
GPT 6 Astra	50.1	94.3	2.18	12.4	**92.7**
Gemini 3.8 Flash	37.9	88.2	1.63	9.3	86.2
GPT 5.6 Sol	37.7	94.4	0.89	8.2	86.9
DeepSeek V4 Flash	28.6	78.1	0.14	19.0	66.8
GPT 5.6 Luna	19.3	74.1	**0.03**	**4.0**	65.4

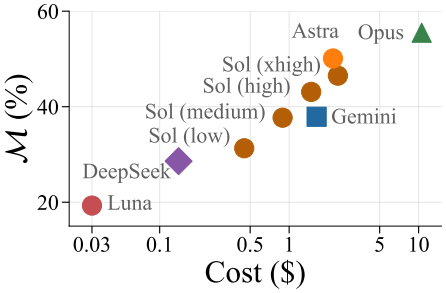


Figure 5: Increased cost provides marginal gains for $\mathcal{M}$.

of $10.56 per instance and taking on average 37 min. It is followed by GPT 6 Astra which achieves 50.1%, at only $2.18 and 12.4 min. All other agents achieve less than 40%.

Agents Generate Valid Tests and Implementations Interestingly, all models but DeepSeek V4 Flash and GPT 5.6 Luna generally produce valid tests and implementations with validity scores $\mathcal{V}$ between 88%-96% and reference test pass rates $\mathcal{R}$ between 86%-93%. Both imply that the specification is clear and precise enough to provide a detailed understanding of the expected behaviour, validating our construction process. In particular the high $\mathcal{R}$ implies that the specification covers all the requirements in sufficient detail to pass the reference test suite. This makes it all the more surprising that all models still fail at generating strong test suites and points to a gap in their capabilities.

Valid but Weak Tests The large gap between validity scores $\mathcal{V}$ (88%-96%) and mutation scores $\mathcal{M}$ (38%-56%), in particular, has a concerning implication for practitioners. While the models are capable of writing valid tests, giving the impression of well tested software, they fail to write discriminative tests that would actually surface specification violations. This directly limits the capability of current agents for autonomous software engineering and reflects commonly observed patterns of code agents writing large test suites yet not detecting failures in their implementations.

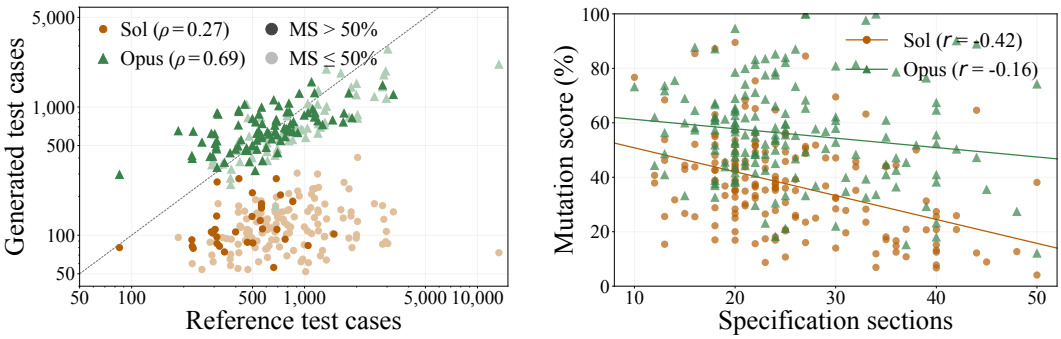
Cost-Strength Tradeoff Overall, we observe a clear logarithmic relationship between cost and mutation score $\mathcal{M}$ with more recent frontier models achieving the best performance at the highest prices. All evaluated models except for Gemini 3.8 Flash lie on the pareto frontier.

Reasoning Effort Ablating over reasoning effort for GPT 5.6 Sol, we see the same trend continue. A higher reasoning effort increases both cost and mutation score $\mathcal{M}$ from 31.3% at $0.45 per instance on low reasoning to 46.5% at $2.4 per instance on xhigh reasoning.

3.2 Additional Analysis

Model Test Suites are (Too) Small We investigate the observed low mutation scores comparing the number of generated tests with those in the reference test suites in Figure 6a. While all models, in expectation generate smaller test suites than the reference, the gap ($\Delta = -8.9\%$) is smallest for Opus with Sol ($\Delta = -79.9\%$) producing roughly 5x smaller test suites than the reference. This correlation is unsurprising as a high mutation score $\mathcal{M}$ hinges on exhaustively covering all requirements, which generally requires a large number of tests. We observe a general trend that (i) instances with fewer reference tests (and thus fewer requirements to cover), and (ii) a larger number of generated tests lead to a better $\mathcal{M}$. We provide a more comprehensive analysis in App. A.1.

Agents Struggle with Large Specifications In Figure 6b (full results in App. A.2), we visualize the relationship between the number of specification sections and test generation performance ($\mathcal{M}$). Across all models, we observe a negative correlation (Opus: $\rho = -0.16$, Sol: $\rho = -0.42$), indicating that models struggle to generate strong test suites for large requirement sets.



(a) Reference and generated test counts.

(b) Specification length and mutation score.

Figure 6: **Left:** Opus generates more tests than Sol with stronger correlation to the reference test count. **Right:** Both models struggle to generate strong test suites for larger specifications.

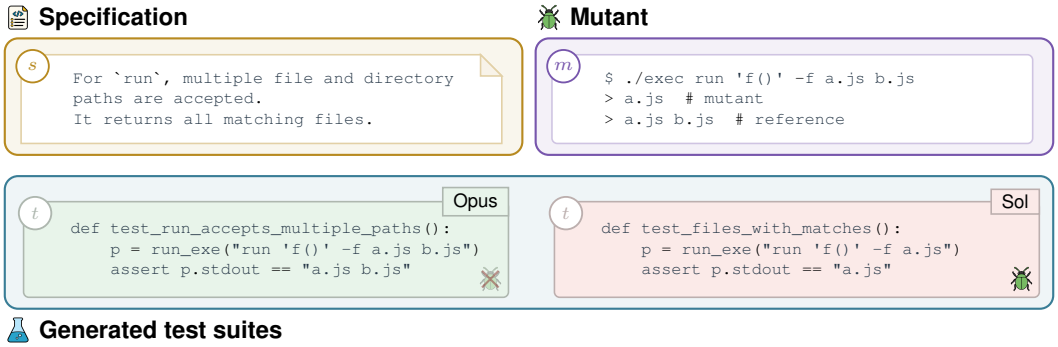


Figure 7: The specification requires `run` to accept multiple paths. The mutant ignores the second path, `b.js`. The Opus test checks both paths and kills the mutant. In contrast, the Sol test only checks one path and misses the mutation. Complete version of error analysis is provided in App. A.6.

Executable/Mutant Creation during Test Generation We have observed some agents to create implementations and mutants to help test suite generation, without them being instructed to do so. Here we investigate whether such an approach improves the resulting test suites. We design a rule-based parser to detect whether the test generation trajectory contains code generation attempts (e.g., write `workspace/executable`), or mutation attempts (e.g., shell action mentioning mutation).

Table 3: Executable and mutant creation attempt detected in test generation trajectories. Opus has the highest signal, which could explain its highest $\mathcal{M}$.

Model	Exec.	Mutants	$\mathcal{M}$
Opus	149	33	55.5
Gemini	102	12	37.9
Sol	4	0	37.7
DeepSeek	58	4	28.6

As shown in Table 3, Opus creates both executables and mutants most frequently, agreeing well with its strong mutation score $\mathcal{M}$. Yet, even Opus creates mutants in only 33 of 165 instances, indicating potential for mutant based test suite generation methods. Interestingly, however, mutant and executable generation do not seem to be necessary to generate strong test suites; Sol creates by far the fewest executables and no mutants at all, yet still achieves better $\mathcal{M}$ than DeepSeek. This allows Sol to create complete test suites in much fewer steps than any other model (Figure 10).

Error Analysis To provide a more in depth understanding of common failure modes, we show a small case study in Figure 7. We consider a mutant from the `ast-grep` instance, where the specification mentions that `run` (the pattern-search command) accepts multiple files. The mutant breaks this contract, ignoring the second item, `b.js`. While the Opus-generated test suite captures this requirement, generating a test that provides two files and successfully identifies the mutation, the Sol-generated test suite tests `run` only with a single input file and therefore fails to kill the mutant.

Table 6: Comparison to prior work. Inputs are implementations I, executables E, and specifications S. We distinguish whether the evaluation employs mutation score $\mathcal{M}$ and validity rate $\mathcal{V}$, and whether the evaluation target is mainly coding agents or LLMs.

Benchmark	# Repos.	Scope	Inputs	$\lvert T\rvert/N$	$\mathcal{M}$	$\mathcal{V}$	Agents
SWT-Bench (Mündler et al., 2024)	12	Issue	I	3	✗	✓	✓
TestGenEval (Jain et al., 2025)	11	File	I	57	✓	✓	✗
MultiFileTest (Wang et al., 2026)	21	Few-File	I	–	✗	✓	✗
SWE-Mutation (Sun et al., 2026)	53	Issue	I	62	✓	✓	✓
RESTestBench (Kogler et al., 2026)	3	Scenario	S, E	1	✓	✓	✗
TESTSUITEBENCH (ours)	165	Repository	S	944	✓	✓	✓

We observe this to be a common pattern where core functionality is tested but input combinations and edge cases are omitted from test suites. We provide additional examples in App. A.6.

Providing Reference Executables for Test Generation We assess whether providing the reference executable in addition to the specification and thus disambiguating any potential remaining uncertainties improves performance using GPT 5.6 Sol and showing results in Table 4. As expected, the validity rate $\mathcal{V}$ increases even further from 94.4% to 98.4% as the model can directly inspect the reference executable when constructing its tests. In contrast, however, the mutation score barely moves, from 37.7% to 38.1%. This indicates that a lack of clarity in the specification that could make the model hesitate to add invalid tests does not explain the low mutation scores observed across this analysis.

Table 4: Input ablation of test generation (Sol). Even with the access to the reference executable, mutation score remains low.

Setting	$\mathcal{V}$	$\mathcal{M}$
Spec only	94.4%	37.7%
+ Executable	98.4%	38.1%

Comparison with Developer-Written Documentation We generate rich specifications rather than relying on developer documentation, which is typically brief: only 30 of our 165 repositories provide documentation beyond their README. Replacing our specifications with this documentation on these 30 instances, Sol's mutation score drops from 25.5% to 4.0% (Table 5), as it writes 40% fewer tests (2,274 vs. 3,789), of which fewer are valid (75.4% vs. 86.7%). Classifying the failing tests as in Section 2.3, we observe the original documentation induces three times as many underspecification failures per test.

Table 5: Performance with the developer specification (Sol). 30 instances with the official documentation are subsampled.

Setting	$\mathcal{V}$	$\mathcal{M}$
Official Docs	75.4%	4.0%
Our Spec	86.7%	25.5%

4 RELATED WORK

Repository-Level Coding Benchmarks Recently multiple coding benchmarks task agents with implementing full projects from scratch. NL2Repo-Bench (Ding et al., 2025) provides agents with an extremely detailed specification and ProgramBench (Yang et al., 2026) with reference executable. Both task the agent with (re)implementing the program. In contrast, we measure the ability to generate a test suite given only the specification.

Test Generation Benchmarks SWT-Bench (Mündler et al., 2024) measures whether agents can add regression tests reproducing user-reported bugs. SWE-Mutation (Sun et al., 2026) evaluates issue-directed test generation and program repair using both bug reproduction and mutation testing. TestGenEval (Jain et al., 2025) evaluates tests generation for individual source files, measuring test validity, coverage, and mutation scores. MultiFileTest (Wang et al., 2026) evaluates test generation from source code for 59 released projects from 21 repositories across Python, Java, and JavaScript, primarily through direct prompting and error-fixing loops; it also includes a four-project agent case study. RESTestBench (Kogler et al., 2026) evaluates individual test generation for specified scenarios against three REST services using manually constructed mutants.

We compare our work against these benchmarks in Table 6 and highlight two crucial differences. i) prior work always provides access to an implementation or executable and ii) prior work restricts test generation by focusing on specific behaviors, individual files, or concrete issues. As a result, the golden reference test suite size per task ($|T|/N$) and thus the expected size of generated test suites is orders of magnitudes smaller than ours.

5 Limitations and Future Work

Mutants as a Proxy for Test Strength To keep construction and evaluation tractable, our mutants target a sample of requirements, selected uniformly across the specification (Section 2.2). Gaps in untargeted requirements thus go unpenalized, but since agents cannot know which requirements are targeted, only suites covering the whole specification achieve a high expected score. Further, selecting mutants that are killed by a single reference test risks favoring mutants triggered only by that test's exact inputs. We mitigate this by instructing the mutation agent to mutate the logic implementing a requirement and removing mutants that hard-code behavior for specific inputs in a manual and agentic audit (Section 2.4). Finally, each mutant violates a single requirement by design and need not reflect the distribution of real-world faults.

Specification Realism We derive specifications from reference test suites and refine them until they suffice for test and code generation, making them far more detailed than typical developer documentation, with which mutation scores drop from 25.5% to 4.0% (Table 5). However, specifications sufficient to build software from scratch, such as product requirement documents, must be similarly detailed, making ours a reasonable proxy for this setting. Where our specifications are nonetheless more complete, our results may overestimate agents' performance, which would only strengthen our conclusion.

Reference Behavior as Ground Truth Our construction process aligns specifications, reference tests, and mutants with the reference implementation, so all of them inherit its bugs and incidental behavior. We manually removed obvious cases, such as crashes expected at specific source lines (App. B.3), but subtler ones may persist. We accept this trade-off, as grounding all artifacts in an existing implementation is what makes our construction scale.

Scope Building on ProgramBench, TestSuiteBench is limited to command-line applications in compiled languages, as ProgramBench provides agents the reference executable, which would expose the source code for interpreted languages. Since TestSuiteBench provides only the specification, our pipeline can be extended to interpreted languages such as Python and TypeScript in future work.

6 Conclusion

We introduced TestSuiteBench, a benchmark of 165 open-source applications that measures whether coding agents can turn a natural-language specification into a strong test suite, using mutants that each violate a single requirement. We find that frontier agents can build software they cannot verify: their tests are largely valid and their implementations pass most reference tests, yet even the strongest agent's suites kill just over half of the mutants. Giving agents the reference executable leaves mutation scores unchanged, pointing to test design rather than specification quality as the bottleneck: generated suites are often too small and miss the input combinations needed to expose violated requirements. Closing this gap is a prerequisite for fully autonomous development of new software, and TestSuiteBench makes progress toward it measurable.

Reproducibility Statement

We describe our benchmark construction process in detail in Section 2 and our evaluation methodology in Section 3. We also provide the full prompts used for construction and evaluation in App. D. The full benchmark data and the code required to reproduce it will be released upon publication.

AI Usage Statement

In this work, we used generative AI tools to generate synthetic data and to clean datasets: coding agents (Codex with GPT-5.6 Sol) construct and refine the reference test suites, specifications, and mutants of TestSuiteBench, as described in Section 2. We have not used generative AI tools to develop conceptual frameworks, propose or refine hypotheses, design or provide feedback on research methodology or experiments, implement methods, or interpret results, and the remaining required disclosure tasks (formulating mathematical claims, proofs, translation, and qualitative or thematic data analysis) are not applicable to this work. Additionally, we used generative AI tools to create and modify figures, to draft parts of the paper, and to polish the writing for readability. We have reviewed all AI-assisted work: agent-generated benchmark artifacts are validated by the automated checks of our construction pipeline and a manual quality check (Section 2.4), and all AI-assisted text and figures were reviewed and edited by the authors. We take responsibility for the final content of this work, including text, claims, and artifacts produced with the aid of generative AI.

References

Anthropic. Claude Opus 5 System Card. https://www.anthropic.com/system-cards, July 2026.

Nicholas Carlini. Building a c compiler with a team of parallel claudes, 2 2026. URL https://www.anthropic.com/engineering/building-c-compiler.

DeepSeek-AI, Anyi Xu, Bangcai Lin, Bing Xue, Bingxuan Wang, Bingzheng Xu, Bochao Wu, Bowei Zhang, Chaofan Lin, Chen Dong, Chenchen Ling, et al. Deepseek-v4: Towards highly efficient million-token context intelligence. *arXiv preprint arXiv:2606.19348*, 2026.

Xiang Deng, Jeff Da, Edwin Pan, Yannis Yiming He, Charles Ide, Kanak Garg, Niklas Lauffer, Andrew Park, Nitin Pasari, Chetan Rane, Karmini Sampath, Maya Krishnan, Srivatsa Kundurthy, Sean Hendryx, Zifan Wang, Vijay Bharadwaj, Jeff Holm, Raja Aluri, Chen Bo Calvin Zhang, Noah Jacobson, Bing Liu, and Brad Kenstler. Swe-bench pro: Can ai agents solve long-horizon software engineering tasks?, 2025. URL https://arxiv.org/abs/2509.16941.

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.

Google DeepMind. Gemini 3.8 Flash Model Card. https://deepmind.google/models/model-cards/gemini-3-8-flash/, September 2026.

Ruanqianqian Huang, Avery Reyna, Sorin Lerner, Haijun Xia, and Brian Hempel. Professional software developers don't vibe, they control: Ai agent use for coding in 2025, 2026. URL https://arxiv.org/abs/2512.14012.

Kush Jain, Gabriel Synnaeve, and Baptiste Rozière. Testgeneval: A real world unit test generation and test completion benchmark. In *International Conference on Learning Representations*, volume 2025, pp. 14947–14999, 2025.

Yue Jia and Mark Harman. An analysis and survey of the development of mutation testing. *IEEE transactions on software engineering*, 37(5):649–678, 2011.

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.

Leon Kogler, Stefan Hangler, Maximilian Ehrhart, Benedikt Dornauer, Roland Wuersching, and Peter Schrammel. Restestbench: A benchmark for evaluating the effectiveness of llm-generated rest api test cases from nl requirements. *arXiv preprint arXiv:2604.25862*, 2026.

Wilson Lin. Scaling long-running autonomous coding, January 2026. URL https://cursor.com/blog/scaling-agents.

Zimu Lu, Yunqiao Yang, Houxing Ren, Haotian Hou, Han Xiao, Ke Wang, Weikang Shi, Aojun Zhou, Mingjie Zhan, and Hongsheng Li. Webgen-bench: Evaluating llms on generating interactive and functional websites from scratch, 2025. URL https://arxiv.org/abs/2505.03733.

Niels Mündler, Mark Niklas Mueller, Jingxuan He, and Martin Vechev. SWT-bench: Testing and validating real-world bug-fixes with code agents. In *The Thirty-eighth Annual Conference on Neural Information Processing Systems*, 2024. URL https://openreview.net/forum?id=9Y8zUO11EQ.

Ahmad Mustafa, Wan MN Wan-Kadir, Noraini Ibrahim, Muhammad Arif Shah, Muhammad Younas, Atif Khan, Mahdi Zareei, and Faisal Alanazi. Automated test case generation from requirements: A systematic literature review. *Computers, Materials and Continua*, 67(2):1819–1833, 2021.

Christian Mürtz and Mark Niklas Müller. Agents in the wild - dashboard. https://insights.logicstar.ai, 2025. URL https://doi.org/10.5281/zenodo.15846865. Interactive web dashboard. Code available at https://github.com/logic-star-ai/insights.

OpenAI. GPT-5.6 System Card. https://deploymentsafety.openai.com/gpt-5-6, July 2026a.

OpenAI. GPT-6 Astra System Card. https://deploymentsafety.openai.com/gpt-6-astra, September 2026b.

Yuxuan Sun, Yuze Zhao, Yufeng Wang, Yao Du, Zhiyuan Ma, Jinbo Wang, Mengdi Zhang, Kai Zhang, and Zhenya Huang. Swe-mutation: Can llms generate reliable test suites in software engineering? In *Findings of the Association for Computational Linguistics: ACL 2026*, pp. 39651–39674, 2026.

Mark Vero, Niels Mündler, Victor Chibotaru, Veselin Raychev, Maximilian Baader, Nikola Jovanović, Jingxuan He, and Martin Vechev. Baxbench: Can llms generate correct and secure backends?, 2025. URL https://arxiv.org/abs/2502.11844.

Yibo Wang, Congying Xia, Wenting Zhao, Jiangshu Du, Chunyu Miao, Zhongfen Deng, Philip S Yu, and Chen Xing. Multifiletest: A multi-file-level llm unit test generation benchmark and impact of error fixing mechanisms. In *Findings of the Association for Computational Linguistics: ACL 2026*, pp. 28151–28172, 2026.

John Yang, Carlos E. Jimenez, Alexander Wettig, Kilian Lieret, Shunyu Yao, Karthik Narasimhan, and Ofir Press. Swe-agent: Agent-computer interfaces enable automated software engineering, 2024. URL https://arxiv.org/abs/2405.15793.

John Yang, Kilian Lieret, Jeffrey Ma, Parth Thakkar, Dmitrii Pedchenko, Sten Sootla, Emily McMilin, Pengcheng Yin, Rui Hou, Gabriel Synnaeve, et al. Programbench: Can language models rebuild programs from scratch? *arXiv preprint arXiv:2605.03546*, 2026.

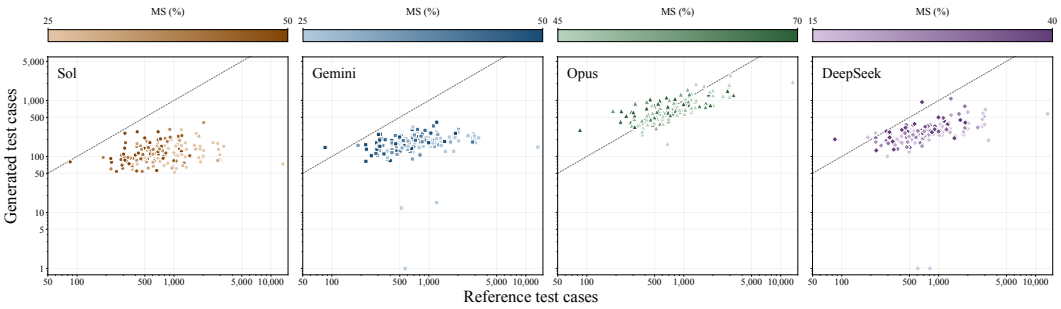


Figure 8: Correlation between number of reference and generated test functions.

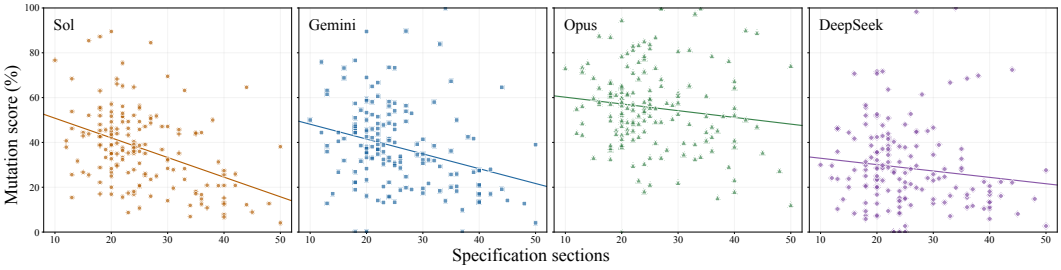


Figure 9: Correlation between the number of spec section and mutation score.

A Additional Experimental Results

A.1 Correlation between Reference and Generated Test Suite

In Figure 8, we provide a full version of Figure 6a, i.e., correlation analysis between the number of reference tests and the number of generated tests. We observe a general trend that the number of generated tests is generally lower than the reference test suite, with an exception of Opus generating similar number of tests as the reference test suite, and that the $\mathcal{M}$ is generally higher when the reference suite is smaller, and generated tests is closer to the reference test suite.

A.2 Correlation between Spec Length and Mutation Score

In Figure 9, we provide a full version of Figure 6b, i.e., correlation analysis between the spec length and the mutation score. Across all agents, we observe a weak negative correlation between the spec length and the mutation score, indicating that longer specifications with more specified requirements makes it more challenging for agents to generate strong test suites.

A.3 Score and Cost Distribution

Figure 10 shows the distribution of Reference Pass Rate ($\mathcal{R}$) for code generation, Validity Rate ($\mathcal{V}$) and Mutation Score ($\mathcal{M}$) for test generation, along with the cost, time, and number of steps for each setting.

A.4 Tool Use Over Trajectories

In Figure 11, we show the distribution of tool use over the trajectory of each model, by classifying the actions into `read` (reading file contents, such as `cat`), `search` (finding files or text, such as `ls`, `grep`),`write` (create or edit files), `execute` (run tests or other shell commands), and `other`.

We observe a general trend that first few steps are dominated by `read` and `search`, and majority of the later sections are dominated by `write` and `execute`. This is indicates that the model quickly digests the provided specification (usually a single file) within a few steps, and they start writing while

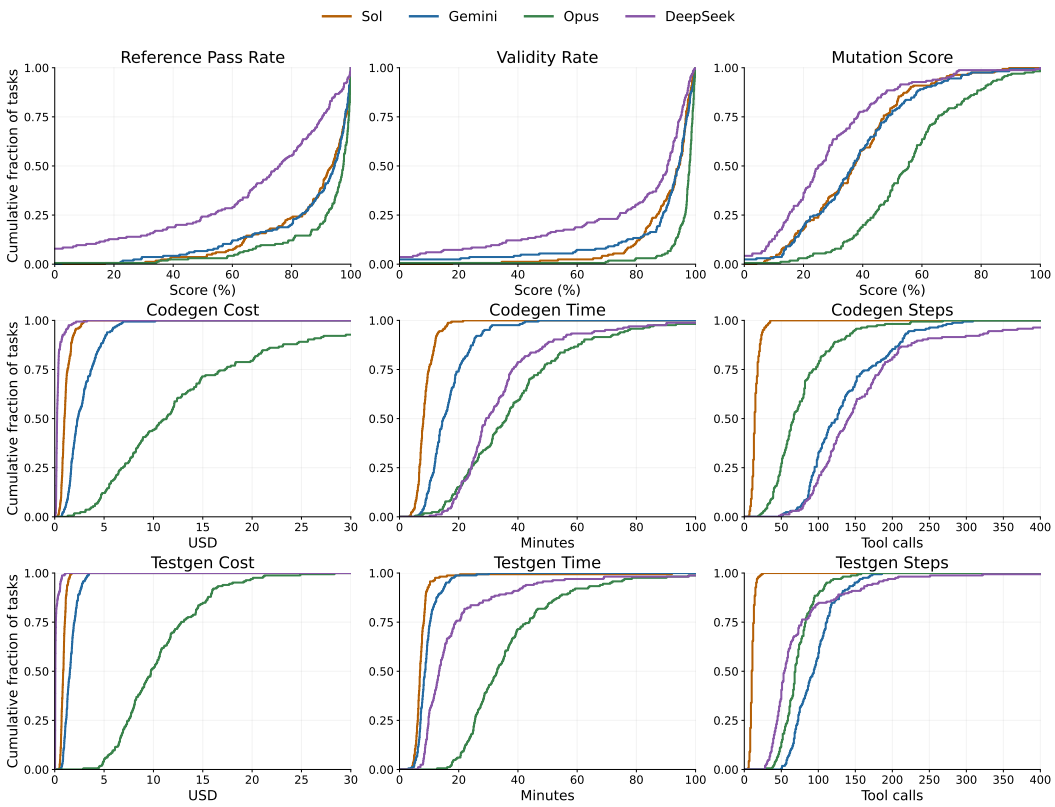


Figure 10: Distribution of mutation scores and costs across models.

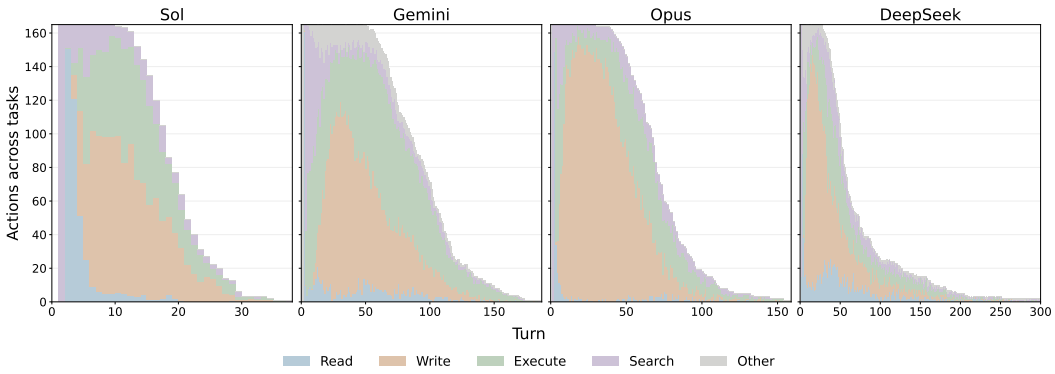


Figure 11: Distribution of tool use across agent trajectories.

executing it to check the correctness of their written code. Overall, `write` dominates the trajectory, with Opus having 60% of its actions being `write`.

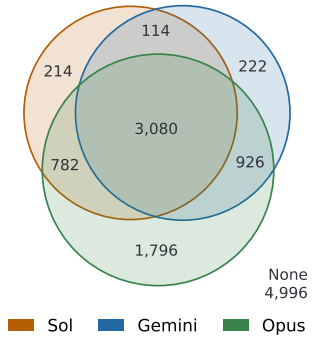


Figure 12: Overlap in mutants killed by test suites generated by different models.

A.5 More Ablation

Overlap of Killed Mutants We consider three representative models (Opus, Sol, Gemini) and analyze the overlap in the mutants they kill (Figure 12). Out of 12,130 mutants, only 3,080 (25%) are killed by all three models, and the two weaker models each kill over 200 mutants that neither other model kills. Combining the three suites increases the number of killed mutants from 6,584 (54%) for Opus alone to 7,134 (59%), leaving 4,996 mutants (41%) that no model kills. Combining agents thus yields only modest gains, and we leave more effective ways to combine them to future work.

A.6 More Error Analysis

Figure 13 extends the error analysis in Section 3.2 with four mutants from `ast-grep` and `cmatrix`

Figure 13a is the full version of Figure 7. Figure 13b is another mutant killed only by Opus. The specification explicitly lists `-ab` as a valid grouping of short options, and the mutant rejects exactly this argument. Opus tests `-ab` before `-h` and kills the mutant, whereas Sol tests only the longer grouping `-abBnbBnnV`, which the mutant still accepts.

The other two mutants survive both suites, because they break the specification only when two specified behaviors are combined. In Figure 13c, the mutant ignores JavaScript `await` expressions only when `run` is given no path. Opus tests nested JavaScript and Python `await` separately, each on an explicit file, and Sol tests only nested JavaScript read from standard input. In Figure 13d, the mutant rejects `-C black` only when it is followed by `-h`. Both suites test `black`, but Opus combines it with rendering and Sol with `-l`. Covering each specified behavior individually is thus insufficient; strong test suites must also exercise their combinations.

B Benchmark Construction Detail

B.1 Details on Filtering

1. Excessive Resources. We limit the execution resources for a test environment run to 2 CPU cores, 60 GB RAM and 1 h. We evaluate reference executable E against T and filter all tests that fail under these constraints.

2. Internet Access. Reliance on internet access tends to make tests flaky, we therefore exclude repositories whose reference test suites require internet access. To detect test cases that require internet, we execute the reference test suite T against E without network access.

3. Flaky tests. Whenever a test fails on I in any step of the pipeline, we exclude it.

(a) `ast-grep`: multiple paths

s
```
For `run` and `scan`, omitted paths mean
the current directory, and multiple file
and directory paths are accepted. [...]
```

m
```
$ ./exec run -p 'hit()' -l js \
    --files-with-matches a.js b.js
> a.js  # mutant
> a.js b.js  # reference
```

t Opus
```
def test_run_accepts_multiple_paths(work):
    r = run_exe(
        ["run", "-p", "console.log($A)",
         "-l", "js", "--json=compact",
         "a.js", "b.js"], work)
    files = sorted(m["file"] for m in r.json())
    assert files == ["a.js", "b.js"], r
```

t Sol
```
def test_files_with_matches(run, path):
    write(path / "a.js", "hit(); hit();")
    p = run(
        "run", "-p", "hit()", "-l", "js",
        "--files-with-matches", "a.js")
    assert p.stdout == "a.js\n"
```

(b) `cmatrix`: grouped short options

s
```
Short options may be separate or grouped,
so forms such as -a -b, -ab, -abc, -lba,
and -abnV are valid. [...]
```

m
```
$ ./exec -ab -h
> exit 1 (no output)  # mutant
> <help text>  # reference
```

t Opus
```
@pytest.mark.parametrize("args", [
    ["-a", "-h"], ["-b", "-h"], ["-ab", "-h"],
    ...,
])
def test_flags_may_precede_help(args):
    res = H.run_plain(args)
    H.expect_help(res)
```

t Sol
```
result = invoke("-abBnbBnnV", term=None)
assert result.returncode == 0
assert b"CMatrix version 2.0" in result.stdout
```

(c) `ast-grep`: JavaScript `await`

s
```
JavaScript is supported. [...]
Each AST node in each selected source is
considered, including nodes at arbitrary
nesting depth. [...]
```

m
```
$ cat a.js
async function f() { await g(); }
$ ./exec run -p 'await $X' -l js \
    --json=compact
> []  # mutant
> [{"text":"await g()"}]  # reference
```

t Opus
```
def test_arbitrary_nesting_depth(work):
    write(work / "a.js", NESTED_JS)
    r = run_exe(
        ["run", "-p", "console.log($A)",
         "-l", "js", "--json=compact",
         "a.js"], work)
    assert len(r.json()) == 1, r

def test_python_await_syntax(work):
    write(work / "a.py", AWAIT_PY)
    r = run_exe(
        ["run", "-p", "await $X",
         "-l", "py", "--json=compact",
         "a.py"], work)
    assert r.json()[0]["text"] == "await g()"
```

t Sol
```
def test_deep_tree_has_no_shallow_limit(run):
    source = "(" * 180 + "target" + ")" * 180
    p = run(
        "run", "-p", "target", "-l", "js",
        "--stdin", "--json=compact",
        stdin=source)
    assert compact_json(p)[0]["text"] \
        == "target"
```

(d) `cmatrix`: `-C black` before `-h`

s
```
-C COLOR accepts black. [...]
Recognized non-terminating flags may
precede help. [...]
```

m
```
$ ./exec -C black -h
> Invalid color selection [...]  # mutant
> <help text>  # reference
```

t Opus
```
VALID = ["green", "red", ..., "black"]

@pytest.mark.parametrize("color", VALID)
def test_valid_color_is_accepted(color):
    with H.start(["-C", color]) as p:
        H.assert_renders(p)
        assert p.stderr() == b""
```

t Sol
```
@pytest.mark.parametrize("color", [
    "green", "red", ..., "black", "ReD"])
# ...
result = invoke("-C", color, "-l")
assert result.returncode == 0
```

Figure 13: **Error analysis on `ast-grep` and `cmatrix`.** Each panel shows the specification (s), the mutant (m), and the relevant Opus and Sol tests (t). A crossed-out mutant icon marks a test that kills the mutant. Opus kills the mutants in (a) and (b); both suites miss those in (c) and (d).

B.2 Optimal Selection of Mutant Targets

Here, we detail the process of selecting the mutation target (Section 2.2). Each mutant targets a pair (s, t), where $s \in S$ is a specification section and $t \in T$ is a reference-test function that evaluates behavior described by s. We add an edge (s, t) whenever this relationship holds.

We augment the resulting bipartite graph with a source α and a sink ω. The edge capacities are

$$u(\alpha, s) = 6, \qquad u(s, t) = 1, \qquad u(t, \omega) = 1.$$

Thus, each specification section receives at most six target tests, while each test can be assigned to at most one mutant. Every unit-carrying path $\alpha \to s \to t \to \omega$ represents one selected target pair.

To choose among maximum-cardinality assignments, we assign cost

$$w(s, t) = d(t),$$

where $d(t)$ is the number of specification sections linked to t. We then solve a min-cost max-flow problem. Consequently, the solver favors section-specific tests over tests shared by many sections and uses the test identifier to break remaining ties.

B.3 Findings in Manual Quality Check

After the spec iteration (Section 2.3), we found 20 instances where the spec contained a specific location of an error (e.g., `stdout should include 'panicked at src/main.rs:24:40'`). All mutants could be trivially killed by a single test function as the error location differs in each mutant. In such a case, we replace it with an obfuscated `<error location>` and remove the test function from the reference suite if there are functions that check such particular property. Further, we found 14 instances where the spec mentions behaviors that are related to tests flagged to be skipped (e.g., requiring internet). We drop these behaviors from the spec in such cases.

C Experimental Setup

Models and Agent Harnesses We consider GPT-5.6 Sol (OpenAI, 2026a), GPT-5.6 Luna (OpenAI, 2026a) GPT-6 Astra (OpenAI, 2026b), Claude Opus 5 (Anthropic, 2026), Gemini-3.8 Flash (Google DeepMind, 2026), DeepSeek V4 Flash (DeepSeek-AI et al., 2026). Unless otherwise indicated, we use default reasoning effort and vendor-recommended agent harnesses

We use default reasoning effort for our main experiment. We use vendor-default harnesses (Codex, Claude Code, Gemini CLI, DeepSeek Harness) unless noted otherwise. Each model runs inside a container with 2 CPUs and 256 GB RAM, with a 6-hour limit per instance (each for code and test generation).

Solution Generation For each task, we use the same container for code generation and test generation. The container is preinstalled with libraries used by the reference I, T, and is the same across instances except for a few instances that have unique libraries which are e.g., incompatible with the global library version setup. The internet is disabled and the only accessible point is the API provider's server endpoint (e.g., `openai.com`).

For code generation, following ProgramBench (Yang et al., 2026), we ask the agent to generate `build.sh` that writes `executable`. The test suite then checks behaviors of the executable instead of the code, allowing unified grading of language-agnostic implementation by the agent.

The test suite generation task asks agents to write a suite that generates its results in JUnit XML format, which is supported by various libraries in various languages, including pytest. While this design conceptually allows cross-language test generation, we in practice find that models struggle to generate the XML themselves. We therefore gave a hint that `pytest -junitxml` can generate the result in the expected format; all observed submissions then used pytest.

Evaluation We use the same container for evaluation as the solution generation container. The generated code is evaluated against the reference suite, and the generated test is evaluated against

reference code and mutants. Each code-test pair is evaluated in a separate container (each with 2 CPUs, 60 GB RAM), with internet being disabled.

As we described in Section 2.1, our primary focus is **Mutation Score ($\mathcal{M}$)** (fraction of mutants killed by the generated suite) of the test generation setup. Along with this, we calculate **Validity Rate ($\mathcal{V}$)** (fraction of generated test suite that passed reference implementation) for test generation, and **Reference Pass Rate ($\mathcal{R}$)** (fraction of reference test suite that passed the implementation) for code generation.

To avoid the trivial test generation described in Section 2.1, we run the test suite on reference I three times, and only use valid tests when calculating $\mathcal{M}$. This evaluation design is explicitly mentioned in the prompt, saying that the test should be "break them down into smaller, focused tests rather than writing one huge test function" so that gradable part will not be massively skipped for $\mathcal{M}$ calculation because only part of it fails.

D PROMPT

D.1 CONSTRUCTION STEPS

Specification-initialization prompt

You are given two directories and must clarify a task specification.

Workspace.

- `specification/`: contains the current task specification. A program and its compiled executable will be created based solely on this specification.
- `test-branches/`: contains the test suite and, when provided, `skipped_tests.json`. Treat every listed function as absent: it is excluded from evaluation and must not influence the specification.

Task.
Update the contents of `specification/` so that it provides enough information for someone to implement an executable that passes all tests in `test-branches/`, without relying on any information outside the specification.

Workflow.

1. Prepare a new file `spec.md`.
2. For each tested behavior, create a section titled `## Spec {i}`, and thoroughly explain the required behavior to the point where an executable can be created based on the document that can pass the test.

Rules.

- Make sure each section is titled `## Spec {i}`.
- The spec may have at least 10 sections, and at most 50 sections. Effectively aggregate similar requirements into a single section.
- The spec should be an abstracted description of the expected behaviors, i.e., what will be tested. Do not simply enumerate over all test functions.
- The required artifact that will be made based on the spec is exactly `executable`. If `eval/run.sh` of the test builds another command from repository source, that command has to be harness-provided. Do not assign its behavior to executable, and write the spec as if they are simply provided and available.
- Write a self-contained specification that does not rely on information outside `specification/`.

Output.
Leave the clarified specification in `specification/spec.md`.

Mutant-suite initialization prompt

You are creating a mutant suite.

Workspace.

- `specification/`: contains the task specification. We aim to perform mutation testing to detect executables that violate the specification.
- `repo/`: contains the ground-truth implementation and `build.sh`, which creates an executable binary.
- `reference-tests/`: contains the immutable active reference-test sources used to construct the plan.
- `mutant_plan.json`: the authoritative list of `(target_spec, target_test)` assignments.
- `mutants.json`: a manifest pre-seeded with exactly the planned mutant IDs and targets.

Task.

Create exactly the mutants in `mutant_plan.json`. Each mutant must violate its assigned specification rule in the narrowest way that is expected to make its assigned test fail while every other active reference-test function continues to pass.

Workflow.

1. Read every assignment in `mutant_plan.json` and inspect its exact `target_spec` and `target_test` source.
2. For every assignment, make exactly one copy of `repo/` whose directory name equals `mutant_id`.
3. Modify source code or build inputs so the assigned specification rule is violated specifically at the behavior asserted by `target_test`. Leave a `[MUTATED]` comment identifying that scope.
4. Keep each copied repository `build.sh` compatible with the original repository build contract.
5. After creating all mutants and updating `mutants.json`, stop. The script will compile the repositories after your turn and will resume this same session with a build report if repairs are needed.

Rules.

- The plan is immutable: do not add, remove, rename, merge, or retarget assignments.
- Create every planned repository exactly once and create no unplanned `repo_*` repositories.
- A valid mutant is intended to have the singleton kill set `{target_test}`. Avoid changes that affect other reference tests, unrelated crashes, global timeouts, or broad output corruption.
- Do not detect the test environment, test name, harness process, or fixture inputs merely to force the assigned test to fail.
- Mutate the logic that produces the behavior: do not be based on literal argument, filename, or payload strings copied from `target_test`, and do not post-process a correct result back into a wrong one. A mutant must be a rule a re-implementer could plausibly get wrong across all inputs in its class.
- Include every generated `repo_*` repository exactly once in `/workspace/mutants.json`.
- Avoid exact full error-message assertions unless they are important; exit codes and functional effects are more important.
- Do not create wrapper/proxy executables that delegate to `/workspace/executable`, `./executable`, or another reference executable. Mutants must be self-contained outputs of their repository build, because evaluation installs each candidate as `/workspace/executable`.
- Do not replace `build.sh` with a script whose main purpose is to write a shell wrapper instead of building the mutated implementation.
- Do not compile or run the mutants yourself: do not invoke `build.sh`, `make`, or a parallel-build helper. The script owns compilation so resource limits and logs remain deterministic.

Output.

- Leave every generated mutant in a numbered `repo_*` repository.
- After defining all repositories, edit `/workspace/mutants.json` so its top-level `mutants` array contains one entry per generated mutant.

Here are the entries you have to fill; the rest are pre-seeded:

- `mutation`: required. The behavior summary and the mutation details – which specification rule is violated, and how the source change violates it.

Leave every other field – `id`, `assignment_id`, `candidate_index`, `target_spec`, `target_test`, and `target_test_description` – exactly as seeded. Do not add, remove, or reorder entries.

Mutant-polishing prompt

You are repairing ProgramBench mutants for `{task}` after fixed-reference-suite mutation evaluation (attempt `{attempt}`).

Workspace.

- `specification/`: the current task specification.
- `reference-tests/`: the immutable extracted test sources.
- `skipped_tests.json`: functions excluded from the fixed reference suite; do not use them as repair evidence.
- `repair_required.json`: the repair evidence.
- `mutants/`: only the affected repositories: `{list of ids}`.

Task.
Try your best to make every listed mutant fail its assigned `target_test` while minimizing failures of other active tests and retaining its initial scope. An exact singleton kill set is desirable, but it is not required.

Rules.

- Modify only repositories already present under `mutants/`.
- Keep every mutant ID and its original `target_spec` and `target_test` unchanged. Do not add, remove, rename, or merge mutants.
- Preserve the behavioral topic of each original mutation. Strengthen or simplify how that same behavior is violated; do not switch to a different spec behavior merely because another test is easy to fail.

- For every edited mutant, the assigned `target_test` must fail. Aim for every other active reference-test function to pass.
- Prefer the smallest source-level change that creates a focused, deterministic violation in the original scope.
- Inspect only relevant reference tests. Never modify tests, the specification, or `repair_required.json`; host-side copyback excludes them.
- Rebuild each edited repository with its existing `build.sh` and leave a self-contained readable `executable`.
- Do not create wrapper/proxy executables, delegate to a reference executable, detect the test environment, key behavior on test names, or introduce unrelated crashes/timeouts.
- If the original change is invalid, replace it with a valid focused violation of the same `target_spec`.
- Some immutable tests may exercise indistinguishable behavior. If further focus would require detecting the test harness or violating these rules, leave that mutant completely unchanged and explain why in `repair_outcome.json`.

Output.

- Leave each repaired mutant in its existing repository under `mutants/`.
- Map each mutant ID to the strengthened violation in its retained scope.
- Write `repair_outcome.json` with two objects: `changed_mutants`, mapping each edited mutant ID to a concise summary, and `kept_as_is`, mapping each unedited mutant ID to the reason further focus is impractical. Every listed mutant ID must occur in exactly one object. This file is authoritative.

Accuracy-classification prompt

You are attributing candidate-implementation failures for a task.

Workspace.

- `/workspace/spec/`: the current specification for the task.
- `/workspace/failures.json`: tests of the fixed reference suite that the candidate implementation failed, one record each. Every record is a candidate failure: the same suite passes on the reference implementation.
- `/workspace/reference-tests/`: the fixed reference suite itself, read-only: test sources, conftest fixtures and golden files. Read the failing test there when the traceback is truncated or the expectation lives in a fixture. Traceback paths are the eval container's: a file quoted under `/workspace/eval/` is staged here at `/workspace/reference-tests/<branch>/eval/`.
- `/workspace/executable`: the reference implementation. Run it to check what the real behavior is.

Each record carries the test `name`, `branch`, `status`, the assertion `message`, the pytest `traceback`, and `relevant_specs` (the specification sections the test was linked to, when known). Any output quoted inside a traceback is output of the candidate implementation, which the test invoked as `/workspace/executable`. The fixed suite passes on the reference implementation, so what a test asserts is the real behavior, and a record always shows the candidate deviating from it.

Task.

There are `{failure_count}` failures in `failures.json`, all from test branch `{branch}`. For each failure, determine whether the specification or candidate executable is at fault.

Classification criteria.

- `executable`: the specification legitimately defines the behavior and the candidate implementation is wrong. Recommended action: `ignore`.
- `spec_underspecification`: the specification is missing or ambiguous with respect to the failed behavior. Recommended action: `polish_spec`.
- `spec_contradiction`: the specification is misleading or contradictory with respect to the failed behavior. Recommended action: `polish_spec`.
- `error`: the failure is an artifact of the environment rather than a judgement about the candidate. Recommended action: `ignore`.
- `unknown`: the failure record and specification do not provide enough information. Recommended action: `manual_review`.

Choose `spec_underspecification` or `spec_contradiction` only when the failure is clearly due to the specification. `spec_contradiction` requires a conflict you can show, not one you infer: you must be able to quote the specification sentence that states something different from what the test asserts. If the specification is merely silent, vague, or less precise than the test expectation, that is `spec_underspecification`. A test expectation the specification simply fails to mention is never a contradiction. As long as the most natural interpretation should lead to the tested behavior, it is not the spec's fault, but it is the executable's fault, even if the expected behavior is not stated explicitly.

Rules.

- Decide from the record, the specification, and what `/workspace/executable` does when you run it. If they leave unclear what the test expected, use `unknown` rather than guessing.
- Judge every record from its own traceback. Tests in one file routinely fail for different reasons, so never carry a verdict from one record to the next. For each record copy the failing assert line into `failing_assertion` and the candidate output that made it fail into `candidate_observed`, both verbatim from that record's traceback. A verdict whose copied evidence is not found in its own record is rejected and the record is re-judged.
- In a pytest diff, `-` lines are the assert's right operand and `+` lines its left one. A `CompletedProcess(...)` repr is the candidate's output. These rules can be different if the test is written in other frameworks. Read the whole failing assertion, not only its first clause, and check which flag or argument the test actually passed.

- Before emitting `spec_contradiction`, reconstruct the failing invocation from the traceback, run it on `/workspace/executable`, and state in the rationale the quoted specification claim, the test expectation, and the reference output that confirms the test. Copy the sentence from `/workspace/spec/` as it appears. If no such sentence can be quoted, or the invocation cannot be run here, then the fault is not a contradiction: use `spec_underspecification` when the specification is merely silent or vague, and `executable` otherwise. Put that sentence in `spec_quote`; a contradiction whose `spec_quote` is not a literal substring of the specification is rejected and the verdict is discarded, so never paraphrase, trim or reflow it.
- `{network_note}`
- Preserve each failure's `name` and `branch` exactly so the result can be matched to its input.

Output.
Write `classifier_result.json` containing exactly one JSON object with this shape:

```
{
  "failures": [
    {
      "name": "same failure name from failures.json",
      "branch": "same branch from failures.json",
      "fault": "executable|spec_underspecification|spec_contradiction|error|unknown",
      "subreason": "short_snake_case_reason providing detail beyond fault",
      "confidence": 0.0,
      "rationale": "a few sentences that explicitly name every relevant section as Spec N and cover (i)
which specification number provides relevant information, (ii) what the test evaluates in that regard, (iii)
why the executable failed, and (iv) why it is classified as
executable/spec_underspecification/spec_contradiction/error/unknown",
      "failing_assertion": "the failing assert line, copied verbatim from this record's traceback",
      "candidate_observed": "the candidate output or status that made it fail, copied verbatim from this
record's traceback",
      "spec_quote": "for spec_contradiction only: the conflicting sentence copied verbatim from
/workspace/spec/, otherwise empty",
      "recommended_action": "ignore|polish_spec|manual_review"
    }
  ]
}
```

Checkpointing.
Persist classifications incrementally to `classifier_result.json` in the workspace. After each compact group, atomically rewrite that valid JSON object with every completed entry so far. If time or output budget runs out, leave as many completed entries as possible.

Final response.
Do not return failures inline. When finished, return only: `{"output_path": "classifier_result.json"}` Do not add markdown or other text. The checkpoint file is authoritative.

Precision-classification prompt

You are classifying generated tests that failed on the ground-truth executable.

Workspace.

- `/workspace/spec/`: the current specification used to generate the test suite.
- `/workspace/test/`: the Codex-generated test suite produced from `/workspace/spec/`.
- `/workspace/generated_test_failures.json`: generated tests that failed on the ground-truth executable.
- `/workspace/executable`: the ground-truth executable itself. Run it to see what the real behavior is.

Task.
There are `{failure_count}` generated-test failures in `generated_test_failures.json`. For each generated-test failure, decide whose fault it is.

Classification criteria.

- `test`: the generated test exercised the executable but asserted something the specification does not entail: it misread a sentence, demanded an exact incidental value, or depends on timing. Recommended action: ignore for spec/mutant polishing.
- `spec_underspecification`: the specification is missing or ambiguous about the behavior, causing the generated test to make an unsupported assumption. Recommended action: polish the specification.
- `spec_contradiction`: the specification is misleading or contradicts the reference behavior, causing the generated test to encode the wrong behavior. Recommended action: polish the specification.
- `error`: the failure says nothing about the specification or the behavior: the test never reached the executable, its own fixture crashed or was written wrongly (an unexecutable fake tool, a missing input file), or the sandbox interfered. The executable was evaluated with networking disabled; a failure whose proximate cause is the missing network is `error`, never evidence about the specification or the executable. A package install that could not reach an index also belongs here. Recommended action: ignore for spec/mutant polishing.
- `unknown`: there is not enough information to decide.

Choose `spec_underspecification` or `spec_contradiction` only when the failure is clearly due to the specification. If the most natural interpretation should not result in this error, it is the test's fault, even if the expected behavior is not stated explicitly. Never classify an environment failure as a specification fault.

Rules.

- Explore only the relevant specification and generated tests. Do not do broad exploration.
- Run `/workspace/executable` on the failing invocation before deciding. A failure that never reached the executable is `error`; one whose expectation the executable in fact satisfies is `test`.
- Before emitting `spec_contradiction`, copy the misleading specification sentence into `spec_quote` exactly as it appears in `/workspace/spec/`, and give the observed reference output that conflicts with it in the rationale. A `spec_quote` that is not a literal substring of the specification is rejected and the verdict is discarded, so never paraphrase, trim or reflow it.
- Before emitting `test`, copy into `spec_quote` the specification sentence that shows the test's expectation is not entailed: the sentence the test misread, or the one that states the behavior the reference actually has. The same literal-substring check applies. If no sentence addresses the behavior the test assumed, the specification is silent about it: that is `spec_underspecification`, not `test`, unless the test's own record shows it never exercised the executable.
- Each record carries `timing_sensitive` (the test sleeps, polls or waits with a fixed deadline) and `flaky_siblings_in_file` (tests in the same file that passed and failed across reference trials). When both are nonzero and the specification describes the behavior as asynchronous, consolidated or best-effort, the failure is a race in the test, not a contradiction: classify it as `test` unless the reference fails the invocation deterministically when you replay it.
- You may repair a broken fixture (for example a fake tool with a malformed shebang) to replay the test and learn what the reference does. When your verdict rests on such a replay, set `replayed_with_fix` to true and describe the repair in `fix_description`; the recorded failure itself is then also an `error`, and the spec finding is what the polish step consumes.
- Prefer a specification fault only when clarifying the specification would plausibly prevent this kind of generated test failure. Use `spec_underspecification` for missing or ambiguous guidance and `spec_contradiction` for misleading or contradictory guidance. Prefer `test` when the test asserts behavior not entailed by the spec, exact incidental values, timing-sensitive behavior, or behavior contradicted by the ground truth.

Output.
Return exactly one JSON object with this shape:

```
{
"test_failures": [
    {
      "name": "same name from generated_test_failures.json",
      "fault": "spec_underspecification|spec_contradiction|test|error|unknown",
      "subreason": "short_snake_case_reason",
      "confidence": 0.0,
      "rationale": "a few sentences that explicitly name every relevant section as Spec N and cover (i)
which specification number provides relevant information, (ii) what the test evaluates in that regard, (iii)
why the executable fails, and (iv) why it is classified as
spec_underspecification/spec_contradiction/test/error/unknown",
      "spec_quote": "for spec_contradiction and test: the sentence copied verbatim from /workspace/spec/,
otherwise empty",
      "replayed_with_fix": false,
      "fix_description": "when replayed_with_fix is true: what you repaired to replay the test, otherwise
empty",
      "recommended_action": "polish_spec|ignore|manual_review"
    }
  ]
}
```

Return one entry for every case in `generated_test_failures.json`, in the same order.

Checkpointing.
Write the object to `/workspace/classifier_result.json`. Persist incrementally: after each compact group, atomically rewrite that valid JSON object with every completed entry so far. If time or output budget runs out, leave as many completed entries as possible. Every entry needs its own `subreason` and rationale; never reuse one verdict across a group.

Final response.
Do not return the entries inline. When finished, return only: `{"output_path": "classifier_result.json"}` Do not add markdown or other text. The checkpoint file is authoritative.

Recall-classification prompt

You are classifying ProgramBench specification clarity and generated-test adequacy using survived immutable mutants.

Workspace.

- `/workspace/spec/`: the current specification for the task.
- `/workspace/test/`: an agent-generated test suite produced from `/workspace/spec/`.
- `/workspace/mutants/`: the relevant mutant repositories or executables.
- `/workspace/mutant_cases.json`: survived mutants to classify.
- `/workspace/executable`: the ground-truth executable. Run it to see what the real behavior is.

Task.

There are `{mutant_count}` mutant cases in `mutant_cases.json`. For each mutant case, decide whether the specification needs clarification.

Evidence fields.

- `n_generated_tests_observed`: all generated tests observed in the evaluation matrix.
- `n_generated_tests_passing_ground_truth`: generated tests accepted as valid because they pass on the ground-truth executable.
- `n_generated_tests_passing_both_ground_truth_and_mutant`: valid generated tests that also pass on this mutant.
- `generated_tests_passing_ground_truth_but_failing_mutant`: valid generated tests that kill this mutant.
- `generated_tests_passing_ground_truth`: the valid generated-test baseline; this is not a list of tests that pass on the mutant.

Classification criteria.

- `spec_underspecification`: a survived mutant exposes behavior that the specification leaves missing or ambiguous, so future generated suites need clearer instructions. Recommended action: `polish_spec`.
- `spec_contradiction`: a survived mutant exposes behavior for which the specification is misleading or contradicts the reference behavior. Recommended action: `polish_spec`.
- `test`: a survived mutant violates behavior already stated clearly in the specification, so this generated suite is weak or wrong. Recommended action: `ignore`.
- `unknown`: there is not enough information to decide. Recommended action: `manual_review`.

Choose `spec_underspecification` or `spec_contradiction` only when the failure is clearly due to the specification. If the most natural interpretation should not result in this error, it is the test's fault, even if the expected behavior is not stated explicitly.

Rules.

- Explore only the relevant specification, generated tests, and immutable mutant files. Do not do broad exploration.
- Never recommend changing, rebuilding, removing, strengthening, or narrowing a mutant.

Output.
Return exactly one JSON object with this shape:

```
{
  "mutants": [
    {
      "mutant_id": "same mutant_id from mutant_cases.json",
      "fault": "test|spec_underspecification|spec_contradiction|unknown",
      "subreason": "short_snake_case_reason",
      "confidence": 0.0,
      "rationale": "a few sentences that explicitly name the mutant's target_spec as Spec N, name any other
relevant sections as Spec N, and cover (i) which specification number provides relevant information, (ii)
what the test evaluates in that regard, (iii) what the scope of the mutant is, (iv) why the mutant survived,
and (v) why it is classified as test|spec_underspecification|spec_contradiction|unknown",
      "recommended_action": "ignore|polish_spec|manual_review"
    }
  ]
}
```

Return one entry for every case in `mutant_cases.json`, in the same order.

Checkpointing.
Write the object to `/workspace/classifier_result.json`. Persist incrementally: after each compact group, atomically rewrite that valid JSON object with every completed entry so far. If time or output budget runs out, leave as many completed entries as possible. Every entry needs its own `subreason` and rationale; never reuse one verdict across a group.

Final response.
Do not return the entries inline. When finished, return only: `{"output_path": "classifier_result.json"}` Do not add markdown or other text. The checkpoint file is authoritative.

D.2 Evaluation

Below are the prompts used for evaluating the models. These are also used for the **Mock Run** step during the spec improvement iteration (Section 2.3).

Code-generation prompt

You are reimplementing a program from its specification.

Workspace.
The workspace contains the program specification.

Task.
Reimplement a new, original codebase from scratch that satisfies the specification and produces the required executable.

Rules.

- You may read bundled documentation and the specification.
- Do not search the internet, clone repositories, or download the original project from package registries.

- Do not run package-manager updates or installs, curl/wget, networked git commands, or any other command that fetches task dependencies.
- Use only tools and dependencies already present when the task starts. Installing something in the live container does not make it available on a fresh grading checkout.

Output.

- Create `./compile.sh`.
- `./compile.sh` must be executable and must build `./executable` in the workspace root on a fresh checkout.
- Only `./executable` is retained when tests run; the rest of the workspace is removed. It must be self-contained and must not depend on sibling files.

Test-generation prompt

You are creating a test suite.

Workspace.
The workspace contains a program specification.

Task.
Create a comprehensive test suite under `/workspace/eval`.

Evaluation design.

- The test suite will go through mutation testing. It will be evaluated against fresh workspaces containing the program under test at `/workspace/executable`.
- The specification is available only while authoring the suite.
- Tests must invoke `/workspace/executable`, or compute it robustly relative to `eval`.
- Do not hard-code `./executable` inside tests that run after `run.sh` has changed into `/workspace/eval`.

Rules.

- Cover documented functional behavior thoroughly. The test suite must kill any mutants that go against the spec, while it has to pass any implementation as long as it follows the spec.
- Pay particular attention to explicitly numbered or sectioned requirements in the specification. Cover each documented behavior with at least one meaningful test.
- When creating tests, please break them down into smaller, focused tests rather than writing one huge test function; tests that kill the ground truth are not used when calculating the mutation score.

Output.

- `/workspace/eval/run.sh` must exist and be executable.
- `/workspace/eval/run.sh` must always write valid JUnit XML to `/workspace/eval/results.xml`.
- `/workspace/eval/run.sh` must run the whole generated suite, even after individual test failures.
- The report must have a `<testsuites>` root containing `<testsuite>` elements (or a single `<testsuite>` root) and one named `<testcase>` per test. Represent non-passing outcomes with a nested `<failure>`, `<error>`, or `<skipped>` element.
- For pytest suites, this is achieved simply by running `pytest -junitxml=results.xml` from `/workspace/eval`.
- JUnit XML must escape XML metacharacters and must not contain raw control characters or binary stdout/stderr; encode such output with `repr`, base64, or another XML-safe textual representation.
- When performing parametrized testing, always provide short, descriptive `ids=` values. Never allow large strings, bytes, or object representations to become test IDs.
- Put all helper files, fixtures, scripts, and generated test assets under `/workspace/eval`.

E List of Instances

We list all TestSuiteBench instances in Table 7.

Table 7: All 165 TestSuiteBench instances with the number of specification sections $|S|$, mutants $|M|$, and reference test functions $|T|$.

Repository	$\|S\|$	$\|M\|$	$\|T\|$	Repository	$\|S\|$	$\|M\|$	$\|T\|$
abishekvashok/cmatrix	16	48	502	agourlay/zip-password-finder	20	60	679
ajeetdsouza/zoxide	13	39	531	alecthomas/chroma	20	57	186
alexpovel/srgn	33	99	1,807	altdesktop/i3-style	20	59	518
anordal/shellharden	15	45	1,085	antonmedv/fx	32	91	2,026
antonmedv/walk	26	70	412	ariga/atlas	31	92	1,302
arq5x/bedtools2	44	131	1,053	arthursonzogni/json-tui	21	63	754

Repository	$\|S\|$	$\|M\|$	$\|T\|$	Repository	$\|S\|$	$\|M\|$	$\|T\|$
ast-grep/ast-grep	23	69	863	astro/deadnix	20	60	600
bellard/quickjs	48	142	3,029	bensadeh/tailspin	21	63	610
blacknon/hwatch	23	68	791	blake3-team/blake3	16	47	645
bootandy/dust	24	72	521	boyter/scc	40	116	462
brocode/fblog	22	66	975	burntsushi/ripgrep	40	117	1,991
burntsushi/xsv	27	78	1,048	byron/dua-cli	24	72	675
canop/broot	14	41	530	canop/rhit	24	72	817
cheat/cheat	25	73	294	chirlu/sox	40	105	1,097
chmln/handlr	18	50	698	chmln/sd	22	65	810
clog-tool/clog-cli	20	60	569	cmatsuoka/figlet	27	81	872
codesnap-rs/codesnap	22	66	563	cordx56/rustowl	20	57	554
crowdagger/crowbook	42	125	790	cslarsen/jp2a	20	59	624
cweill/gotests	22	64	596	dalance/amber	25	74	563
dandavison/delta	40	117	932	direnv/direnv	23	67	812
doxygen/doxygen	16	41	85	drew-alleman/datasurgeon	21	62	499
ducaale/xh	18	52	1,149	duckdb/duckdb	13	39	2,825
dundee/gdu	23	67	1,140	ecumene/rust-sloth	18	52	374
ekzhang/bore	12	27	377	eliukblau/pixterm	20	55	423
elkowar/pipr	13	38	399	eradman/entr	18	54	571
esubaalew/run	38	106	1,194	eudoxia0/hashcards	20	60	1,014
facebook/zstd	37	111	2,023	facebookresearch/fasttext	26	74	308
filosottile/age	27	76	673	foriequal0/git-trim	24	67	479
gabotechs/dep-tree	35	105	785	git-bahn/git-graph	13	39	467
go-critic/go-critic	24	68	312	gromacs/gromacs	50	148	1,189
guumaster/hostctl	21	61	954	hairyhenderson/gomplate	36	104	2,873
halitechallenge/halite	18	51	224	hooklift/gowsdl	24	69	371
htop-dev/htop	20	56	686	hush-shell/hush	33	98	1,200
incu6us/goimports-reviser	24	71	505	ip7z/7zip	40	120	1,043
ismaelgv/rnr	25	72	675	isona/dirble	30	86	685
ivanceras/svgbob	26	75	470	jarun/nnn	29	51	477
jesseduffield/lazygit	42	112	796	jhspetersson/fselect	25	75	2,971
johanneskaufmann/html-to-markdown	27	81	855	jonas/tig	30	82	1,480
jrnxf/thokr	20	42	322	junegunn/fzf	32	96	1,856
kaushiksrini/parqeye	26	64	470	kisielk/errcheck	20	54	296
konradsz/igrep	20	58	354	ksxgithub/parallel-disk-usage	26	77	467
kyoheiu/felix	24	71	497	lfos/calcurse	22	64	640
lh3/seqtk	23	67	429	lua/lua	35	105	1,337
luajit/luajit	28	84	2,945	lymphatus/caesium-clt	25	75	561
madler/pigz	25	75	830	mfridman/tparse	23	68	413
mgdm/htmlq	21	63	1,453	mgechev/revive	50	105	450
mibk/dupl	18	54	373	mikefarah/yq	36	108	1,997
miserlou/loop	22	66	696	mookid/diffr	19	57	606
multiprocessio/dsq	30	89	542	nachoparker/dutree	18	53	497
naggie/dstask	16	48	1,270	nikolassv/bartib	20	60	716
noborus/ov	40	114	1,804	noborus/trdsql	36	108	1,306
nukesor/pueue	24	61	414	nuta/nsh	34	102	1,881
o2sh/onefetch	30	87	1,142	oppiliappan/eva	18	54	913
orf/gping	22	62	310	parcel-bundler/lightningcss	39	115	2,827
peco/peco	27	58	564	pemistahl/grex	24	71	1,311
pier-cli/pier	10	30	692	pls-rs/pls	20	60	316
psampaz/go-mod-outdated	15	45	284	quinn-rs/quinn	20	59	478
raviqqe/muffet	20	56	291	rbakbashev/elfcat	24	72	560
rcoh/angle-grinder	18	54	1,038	rhysd/kiro-editor	25	74	529
riquito/tuc	23	69	1,195	robertdavidgraham/masscan	39	116	2,533
rochacbruno/marmite	44	127	663	rs/curlie	20	55	687
rs/jplot	21	57	579	rust-ethereum/ethabi	13	39	997
rust-lang/mdbook	40	120	1,097	rvben/rumdl	40	119	3,263
sayanarijit/xplr	22	65	417	sclevine/yj	24	72	764
segmentio/chamber	25	72	1,691	sharkdp/fd	31	90	1,235
sharkdp/hexyl	19	57	906	sharkdp/hyperfine	22	66	289
sharkdp/pastel	30	87	1,114	shashwatah/jot	22	65	744
sibprogrammer/xq	28	83	772	sigoden/argc	36	108	995
simeg/eureka	18	54	344	sirwart/ripsecrets	18	54	610
sitkevij/hex	18	53	823	skeema/skeema	30	88	1,673
sqlite/sqlite	45	135	13,298	sstadick/hck	20	59	852
stacked-git/stgit	34	101	1,341	stathissideris/ditaa	25	75	608
stranger6667/jsonschema	27	81	1,860	svenstaro/genact	33	87	227
svenstaro/miniserve	21	60	304	thezoraiz/ascii-image-converter	20	60	462
tomarrell/wrapcheck	20	59	417	tomnomnom/gron	18	54	223
trasta298/keifu	32	81	255	tree-sitter/tree-sitter	37	110	1,203
typst/typst	35	105	1,622	unhappychoice/gittype	25	75	685
universal-ctags/ctags	15	42	544	wfxr/code-minimap	20	58	311
wfxr/csview	21	63	335	wgunderwood/tex-fmt	19	57	455
wintermute-cell/ngrrram	18	47	302	xampprocky/tokei	22	64	726
y2z/monolith	29	87	709	yaa110/nomino	22	66	312
yassinebridi/serpl	22	65	416	yoav-lavi/melody	35	104	1,205
ys-l/flamelens	12	29	224				