XREPOTEST: Benchmarking Multilingual Repository-Level Unit Test Generation for Large Language Models

Anonymous ACL submission

Abstract

Large language models (LLMs) have shown promise for automated unit test generation, but existing evaluations largely rely on standalone settings and a narrow set of programming languages, overestimating real-world readiness. We introduce XREPOTEST, a multilingual, repository-level benchmark for unit test generation spanning five underexplored languages: Rust, Go, Julia, PHP, and Ruby. XREPOTEST evaluates tests under realistic repository constraints using a containerized execution framework and multiple context augmentation strategies, including file-level, LSP-based, and retrieval-based context. Beyond standard metrics such as test pass rate and coverage, we propose Invocation Rate (IR) to assess whether generated tests meaningfully exercise the intended functionality. Experiments with 14 state-of-the-art LLMs (e.g., Claude 4.5, GPT-5.2, DeepSeek V4-pro, and Qwen families) reveal a substantial gap between standalone and repository-level performance, as well as trade-offs between richer context and test reliability. Overall, XREPOTEST provides a challenging and informative benchmark to advance scalable, robust, and multilingual unit test generation in realistic software environments.

1 Introduction

Writing unit tests is a time-consuming yet critical part of software development (Yuan et al., 2023; Pan et al., 2025; Mohapatra, 2025); studies estimate it can take around 15% of a developer's time. This has motivated decades of research into automated unit test generation. Traditional approaches (Fraser and Arcuri, 2011; Lukasczyk and Fraser, 2022; Noller et al., 2018; Ognawala et al., 2019; Fraser and Arcuri, 2025) (e.g., search-based, symbolic, or random test generation) can produce tests with reasonable coverage, but these auto-generated tests often suffer from poor readability and trivial or ineffective assertions. Recently, large language models (LLMs) have emerged as a promising alternative, significantly accelerating unit test generation and often producing more natural, human-like test cases (Schäfer et al., 2023; Lemieux et al., 2023; Alagarsamy et al., 2024; Pan et al., 2025; Yuan et al., 2024; Watson et al., 2020; Tufano et al., 2022b; Dakhel et al., 2024; Siddiq et al., 2024; Bhatia et al., 2024; Li and Yuan, 2024; Serra et al., 2019; Yang et al., 2026; Le Hai et al., 2025; Zhang et al., 2024; Yang et al., 2024). Although LLMs have been widely adopted for unit test generation, their evaluation still faces significant challenges in practical applicability and multilingual adaptability. Existing test generation benchmarks often fail to reflect realistic repository-level constraints and remain biased toward a small set of programming languages, limiting our understanding of how LLMs perform in domain-specific and language-diverse software systems.

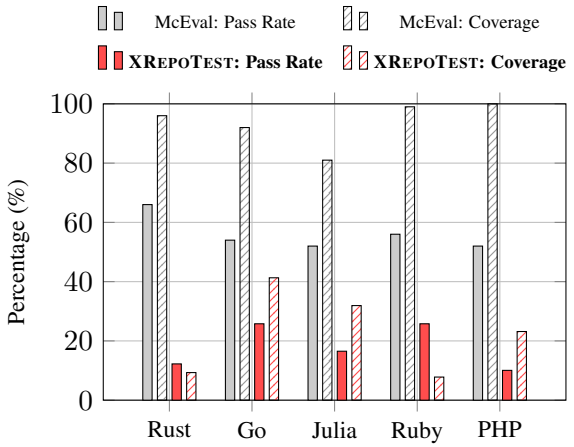


Figure 1: **The Performance Gap:** While LLMs achieve high pass rate and coverage on standalone settings (Gray), performance drops significantly in the repository-level **XREPOTEST** benchmark (Red).

First, various existing unit test generation benchmarks operate at the standalone, function-level, isolating code from the repository contexts in which

real-world tests are written and executed (Chai et al., 2025; Wang et al., 2025; Chen et al., 2021). In these settings, models are not required to interact with realistic build systems, dependency graphs, or framework conventions, leading to an inflated impression of readiness for practical deployment. Figure 1 illustrates this discrepancy: strong performance on standalone benchmarks (i.e., McEval (Chai et al., 2025)) can substantially overestimate model effectiveness when tests must be generated and executed within *real repositories*. In repository-level settings, a model must not only write syntactically plausible tests, but also resolve project-specific dependencies, follow framework conventions, satisfy build systems, and construct valid inputs and assertions under incomplete context. These requirements sharply reduce both feasibility and semantic correctness, revealing a persistent gap between standalone success and real-world unit test generation.

Second, although some recent frameworks and benchmarks consider repository-level test generation scenarios, they remain disproportionately centered on Java and Python (Schäfer et al., 2023; Pan et al., 2025; Jain et al., 2025a). This narrow language focus limits the evaluation of LLM-based test generation methods across diverse programming ecosystems, hindering the development and assessment of tools that are generalizable and practically useful in multilingual software environments (Celik and Mahmoud, 2025). Meanwhile, different programming languages exhibit distinct characteristics and domain-specific usage patterns. For instance, Rust has rapidly become a standard choice for memory-safe systems programming; Go underpins much of cloud-native infrastructure; Julia is driving advances in high-performance scientific computing; and PHP and Ruby continue to power a substantial portion of production web applications (Myklebust et al., 2025; Gäher et al., 2024; Collins et al., 2025). These languages also introduce unique technical challenges; such as Rust's ownership and borrowing model, that can invalidate generation strategies effective in more conventional language settings (Cheng et al., 2025).

To bridge these gaps, we introduce **XRepoTest**, a large-scale multilingual, repository-level benchmark for unit test generation spanning five high-impact yet underexplored languages: Rust, Go, Julia, PHP, and Ruby. XRepoTest provides rich repository context and systematically evaluates multiple context strategies, including retrieval-based, file-level, and LSP-based augmentation, revealing trade-offs between test effectiveness and invocation reliability. In addition, we propose **Invocation Rate (IR)** as a complementary metric to pass rate and coverage, enabling a more faithful assessment of whether generated tests truly exercise the intended functionality. Our comprehensive experiments reveal that in the standard setting, performance is primarily bottlenecked by API hallucinations; while enhancing context improves pass rates and coverage, it often introduces noise that reduces IR, highlighting a complex trade-off between test effectiveness and reliability. In summary, our contributions are as follows:

- **Benchmarking Multilingual Repo-level Test Generation:** We present XRepoTest, a curated benchmark of over $3,642$ focal functions from real-world Rust, Go, Julia, PHP, and Ruby repositories, designed to reflect diverse, real-world project code and testing conventions across these ecosystems.

- **Comprehensive Evaluation Framework:** We provide a containerized evaluation framework that automatically sets up and executes generated tests across all considered languages. Our framework enables reliable, execution-based evaluation using four standard metrics: test pass rate (TPR), compilation success rate (CSR), line coverage, and mutation score; and introduces a novel *invocation rate (IR)* to identify cases where tests pass but fail to meaningfully exercise the focal function, potentially introducing low-quality tests and technical debt.

- **Repository-aware Context:** To examine the influence of varying context, our framework and dataset provide rich metadata and tooling to support both static and dynamic context enhancement. Static context includes File-level context and LSP-based symbol resolution (e.g., argument types and dependency definitions), while dynamic context is enabled via retrieval-augmented approaches using both sparse and dense retrieval methods.

- **Extensive LLMs Evaluation:** We conduct comprehensive experiments on 14 state-of-the-art LLMs spanning families and scales, evaluating diverse context configurations. Our analysis reveals systematic strengths and limitations of cur-

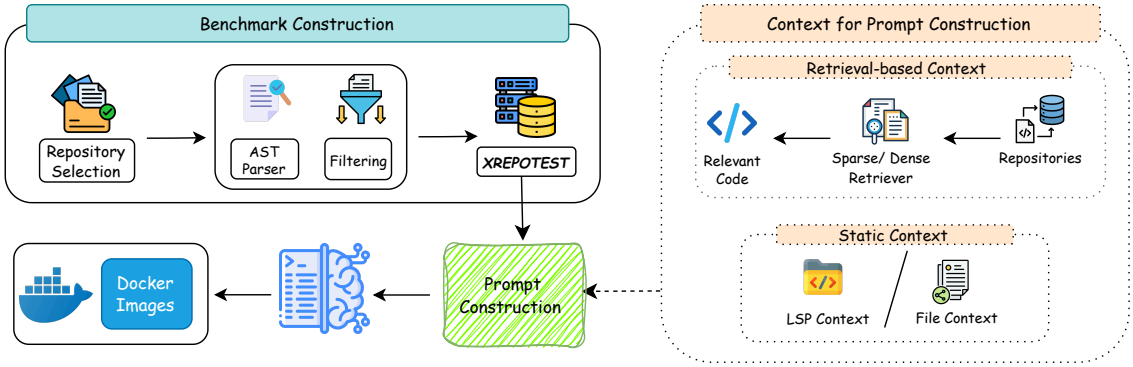


Figure 2: Overview of the XREPOTEST benchmark construction and context-aware evaluation workflow. **Left: Benchmark construction and evaluation.** Repositories are selected, parsed, and filtered to construct the benchmark, after which generated unit tests are executed in containerized, language-native environments for reproducible repository-level evaluation. **Right: Context for prompt construction.**

rent models and demonstrates how context design critically shapes the quality of generated tests.

2 Related Work

Automated unit test generation has a long history, spanning traditional search-based and heuristic techniques as well as more recent LLM-driven approaches. Classic generators can achieve reasonable coverage, but they often produce brittle, low-readability tests with trivial or uninformative assertions, limiting adoption in practice (Fraser and Arcuri, 2011; Lukasczyk and Fraser, 2022). In contrast, LLM-based systems can generate more natural test scaffolding and assertions, further incorporate execution-in-the-loop repair or analysis-guided prompting to improve validity (Chen et al., 2024; Pan et al., 2025; Rao et al., 2023; Ma et al., 2025; Gorinski et al., 2023; Xiong et al., 2024).

From an evaluation perspective, several high-quality benchmarks for unit test generation have incorporated repository context, such as Defects4J (René Just, 2014), Methods2Test (Tufano et al., 2022a), and TestGenEval (Jain et al., 2025a), but they remain largely centered on high-resource languages like Java and Python, limiting their applicability to other ecosystems. More recently, UniTSyn (He et al., 2024) introduced a multilingual test generation framework; however, its evaluation is adapted from standalone code generation benchmarks such as HumanEval-X (Zheng et al., 2023) and McEval (Chai et al., 2025), which abstract away repository-level constraints and thus provide limited insight into the practical effectiveness of test generation methods in real-world settings.

Gap for Multilingual Repository-level Evaluation. Despite the widespread use of many programming languages in practice, benchmarks for evaluating unit test generation at the repository level remain scarce outside a few high-resource ecosystems. A key barrier lies in constructing reliable execution environments, which requires language-specific knowledge of build systems, testing frameworks, and dependency configurations (Le Hai et al., 2025; Zhang et al., 2025a,b; Wang et al., 2025; Islam et al., 2024; Du et al., 2025). To address this gap, we present a large-scale multilingual, repository-level benchmark and data pipeline for test generation, enabling systematic and executable evaluation of test generation models across diverse programming languages.

3 Methodology

In this section, we describe the construction of XREPOTEST and its context-aware evaluation framework, followed by an overview of the dataset's scale, language coverage, and domain distribution. XREPOTEST is repository-level rather than standalone: each task targets a focal function, but generated tests are inserted into and validated within the original project environment. These details highlight the design choices that make XREPOTEST a robust benchmark for evaluating LLM-based unit test generation.

3.1 Dataset Construction

Figure 2 illustrates the end-to-end workflow of XREPOTEST, which consists of three conceptual stages. First, during **Benchmark Construction**, we select repositories and extract non-trivial focal

Benchmark	Language	Size	PF	FC	RC
MBPP (Austin et al., 2021)	Python	500	✗	✗	✗
HumanEval (Chen et al., 2021)	Python	164	✗	✗	✗
TestEval (Wang et al., 2025)	Python	210	✗	✗	✗
McEval (Chai et al., 2025)	40 languages	16,031	✗	✗	✗
SF110 (Fraser and Arcuri, 2011)	Java	182	✓	✗	✗
Defects4J (René Just, 2014)	Java	357	✓	✓	✗
TestPilot (Schäfer et al., 2023)	JavaScript	1,684	✓	✓	✗
TestGenEval (Jain et al., 2025a)	Python	1,210	✓	✓	✗
XREPOTEST	Rust, Go, Julia, PHP, Ruby	3,642	✓	✓	✓

Table 1: Comparison of test generation benchmarks by language, size, and support for project-level functions (**PF**), file context (**FC**), repository context (**RC**).

functions to form the core benchmark. Second, during **Prompt Construction**, each focal function is paired with contextual information drawn from well-defined context sources. This includes (i) *static context*, released as part of the benchmark, and optionally (ii) *retrieval-based (dynamic) context*, constructed on the fly at evaluation time. Finally, generated unit tests are executed and validated in **Containerized Environments** to ensure reproducible, repository-level evaluation. More detailed implementations of the pipeline components, including filtering rules, context augmentation, and containerized environments, are provided in Appendix B. Besides, the discussion on data contamination risk is provided in Appendix C.

Language Selection. We exclude Java and Python due to their extensive prior study and focus instead on PHP, Go, Ruby, and Rust, that are widely used languages in production, but remain underrepresented in repository-level test generation. We also include Julia, a relatively low-resource language that has seen growing adoption and research interest (Joel et al., 2024; Giagnorio et al., 2025).

Repository Selection. For each language, to ensure quality and relevance, we apply filtering criteria including: a minimum of 500 GitHub stars, a clearly defined project structure with build configuration files (e.g., `Cargo.toml`, `Project.toml`, `go.mod`, `composer.json`, or `Gemfile`) to minimize human intervention in environment setup. Additionally, for each language, we ensure that selected repositories span at least two distinct application domains to promote diversity in the dataset. Following this process, we select 6–10 repositories per language, which together form the foundation of our benchmark dataset (See additional per-repository statistics in Appendix A, Table A.1).

Focal Method Extraction. From collected repositories, we use `tree-sitter` (Brunsfeld, 2018) to parse source files into abstract syntax trees (ASTs), enabling language-agnostic traversal and structural analysis. From each AST, we extract top-level function and method definitions. To ensure functions' quality and validity, we apply systematic filtering rules that enforce: (i) minimum code length, (ii) presence of observable outputs (explicit `return` statements), (iii) exclusion of existing test code, and (iv) syntactic validity. These criteria help eliminate trivial or degenerate functions while retaining semantically meaningful *focal methods* suitable for unit test generation. Complete filtering rules are detailed in Table A.2 in the Appendix.

Context Augmentation. To enable realistic and flexible repository-level test generation, our framework supports both static and dynamic context augmentation. The released XREPOTEST benchmark provides precise, lightweight static context with **File-level context** (i.e. file content containing focal method) and **LSP-based context** that resolves argument definitions, type declarations, and relevant callee signatures via Language Server Protocol (LSP), inspired by recent studies (Jain et al., 2025a; Go et al., 2025). In addition, following repository-level code generation practices (Zhang et al., 2023; Wu et al., 2024; Wang et al., 2024), XREPOTEST supports optional **Retrieval-based augmentation**, in which sparse (BM25) or dense (UniXCoder - Guo et al. (2022)) retrievers dynamically select relevant code snippets (i.e., code chunks) from the repository to enrich the prompt.

Containerized Execution Environment. All generated unit tests are executed within Docker containers to standardize toolchains, dependency installation, and evaluation across languages. Each

Algorithm 1 Computation of Invocation Rate (IR)

Require: Set of generated unit tests T, focal method name f
Ensure: Invocation Rate IR

function COMPUTEIR(T, f)
  $N \leftarrow |T|$, $count \leftarrow 0$
  for all test method $t \in T$ **do**
    $AST \leftarrow$ `AST_parse`(t)
    $C \leftarrow$ `GetCallIdentifiers`(AST)
    $count \leftarrow count + 1$ **if** $f \in C$
  end for
  $IR \leftarrow \frac{count}{N}$
  return IR
end function

repository is evaluated using its native test framework (e.g., `cargo test`, `go test`, `Test.jl`, `PHPUnit`, `RSpec`).

Overall, our pipeline provides a flexible and extensible foundation for repository-level unit test generation. Although this work focuses on five programming languages, the framework is designed to scale to additional languages with minimal effort, requiring only the availability of a supported testing framework and appropriate configuration files to set up the execution environment.

3.2 Data Characteristics

Data Statistics. XREPOTEST spans six diverse application domains, reflecting language-specific usage patterns across the selected programming languages. We additionally ensure a balanced number of samples per language to support fair and comparable evaluation across multilingual settings. Detailed repository statistics and domain distributions are provided in Appendix A.

Comparison of Unit Test Generation Benchmarks. Table 1 compares existing unit test generation benchmarks in terms of language coverage, scale, and support context. Most prior benchmarks focus on standalone or file-level settings and are largely confined to high-resource languages such as Python and Java, with limited or no support for full repository context. In contrast, XREPOTEST uniquely combines multilingual coverage with repository-level evaluation, supporting five underexplored languages and providing comprehensive focal, file, and repository-aware context, thereby enabling more practical assessment.

Example: Unintended indirect invocation

```
[Helper Function]
func Circle[...](arr []T) []T {
    ...
    for doSort(...) { /* ... */ }
    ...
}

[Focal Method]
func doSort[...](arr []T, l, r int) bool {
    ...
}

[Generated Test]
func TestDoSort(t *testing.T) {
    var tests = []struct{ /* ... */ }
    for _, tt := range tests {
        got := Circle(tt.input)
        if !reflect.DeepEqual(got, tt.want
            ){
            t.Errorf("...")
        }
    }
}
```

4 Evaluation Metrics

Standard Metrics. To evaluate the effectiveness of generated unit tests, we adopt a set of metrics commonly used in prior work on automated test generation (Tufano et al., 2022a; Alagarsamy et al., 2024; Jain et al., 2025a; Chen et al., 2024; Lukasczyk and Fraser, 2022). **Test Pass Rate (TPR)** measures runtime correctness as the fraction of compiled tests that execute without exceptions or assertion failures. To assess semantic coverage, we report **Line Coverage (Cov)**, which measures the percentage of lines in the focal method exercised by the generated tests. We additionally compute Compilation Success Rate (CSR) and Mutation Score but omit them from the main text for readability; full metric definitions are provided in Appendix D.

Novel Proposed Metric. Standard evaluation metrics, such as TPR, CSR and Cov, often fail to distinguish between *accidental execution* and *intentional validation*. We observe that LLMs frequently produce "passing" tests that contain critical semantic flaws, notably *indirect invocations* where the focal method is executed as a side effect of a wrapper function rather than being explicitly targeted. Example 4 provides an illustrative example of this issue, where the test calls the wrapper `Circle` instead of the focal method `doSort`. In these scenarios, coverage metrics are inflated, recording the focal method as "covered" even though its specific logic is never directly exercised or asserted. To bridge this gap between *execution* and *intent*, we introduce the **Invocation Rate (IR)**. This metric

Model	Rust			Julia			Go			Ruby			PHP		
	TPR	Cov	IR	TPR	Cov	IR	TPR	Cov	IR	TPR	Cov	IR	TPR	Cov	IR
Small															
Qwen3 (8B)	0.32	0.21	64.88	0.88	0.97	73.83	0.42	1.13	47.17	0.00	1.39	**92.15**	0.62	2.13	22.91
Yi-Coder (9B)	1.50	1.71	**90.23**	4.82	19.16	90.50	2.41	6.48	91.36	1.48	9.37	86.81	3.56	17.68	98.76
Codestral (22B)	2.69	2.97	84.85	2.05	21.17	91.23	3.82	14.52	**99.58**	0.15	7.35	87.26	5.57	23.01	99.69
Medium															
Qwen3-Coder-Next (80B)	7.20	3.52	56.71	5.12	17.38	91.08	6.66	16.74	96.88	1.19	6.88	87.26	5.11	22.43	97.68
GPT-OSS (120B)	5.80	4.16	81.63	4.39	24.27	93.86	19.26	26.95	88.24	6.96	8.21	85.93	3.41	5.07	99.23
Llama-3.3 (70B)	3.33	3.04	88.94	3.07	25.42	93.27	2.69	7.96	98.73	0.44	7.66	87.85	2.01	21.75	**99.85**
Large															
DeepSeek V3.2	2.79	3.06	77.66	3.95	21.94	89.47	5.24	14.91	59.07	1.48	10.01	84.30	5.42	20.20	93.34
MiniMax-M2.7	4.30	3.63	66.49	4.39	18.12	86.11	4.67	13.07	81.16	1.63	7.30	87.41	5.57	17.25	97.68
Flagship															
Claude 4.5 Haiku	8.27	3.73	80.34	12.57	23.96	92.69	10.91	26.87	99.43	4.30	10.57	87.11	14.86	34.68	99.54
Claude 4.5 Sonnet	**12.78**	**11.56**	75.94	**19.88**	31.58	**94.01**	23.65	**43.53**	99.43	6.37	**14.25**	85.63	**26.93**	**41.76**	99.38
DeepSeek V4-pro	6.66	4.66	74.22	11.40	26.83	89.77	13.88	21.66	63.46	8.89	9.15	84.74	13.00	22.19	96.13
GLM-5	6.44	2.52	70.68	5.41	13.54	89.47	5.10	9.48	75.78	3.11	8.50	86.96	13.16	24.98	99.07
Kimi-k2.5 (Instruct)	1.40	1.17	74.76	2.63	8.70	87.43	2.27	2.33	75.21	2.52	5.02	78.96	3.25	4.02	88.08
GPT-5.2	12.24	9.31	77.87	16.52	**31.92**	91.81	**25.78**	41.30	98.44	**25.78**	7.80	88.89	10.06	23.17	99.38

Table 2: LLMs Performance on XREPOTEST. Best scores are highlighted. Full metrics are in the Appendix A.4.

explicitly verifies whether a generated test contains a direct call to the focal function under test. By distinguishing these misleading indirect invocations from genuine unit tests, IR serves as a strict indicator of "behavioral relevance", ensuring that reported high TPR and Cov scores actually reflect the model's ability to engage with the focal method. The IR implementation is detailed in Algorithm 1.

Together, these metrics provide a comprehensive paradigm on both the *syntactic soundness* and *execution effectiveness* to evaluate the generated unit tests across different test generation techniques.

5 Findings and Discussion

5.1 Performance across LLMs on XREPOTEST

In this experiment, we evaluate 14 state-of-the-art LLMs across model families and scales on XREPOTEST using minimal *focal/standard context* (the focal method and class signature), following prior studies (Tufano et al., 2022a, 2020; Yang et al., 2024). We first summarize cross-model trends and then analyze results across individual languages. All languages in XREPOTEST are evaluated under a unified protocol using greedy decoding. Details of the models and prompting templates are provided in Appendices H and J.

Overall trends across models. Table 2 indicates that unit test generation remains challenging across the five languages, even for frontier systems. Across languages, the strongest models tend to be instruction-following and more reliable in producing executable test scaffolding, but passing tests remains bottlenecked by correct setup and oracles. Model family effects are pronounced: frontier models (Claude 4.5 and GPT-5.2) dominate many of the highest scores per metric, while strong open-weight baselines (e.g., Qwen variants and GPT-OSS) remain competitive. We also observe that GPT-5.2 is consistently strong on several languages/metrics (e.g., Ruby TPR), but it does not uniformly dominate across the suite, emphasizing that test generation capability remains uneven. Notably, the best model differs by language and metric (e.g., coverage vs. pass rate), reinforcing that improvements are not uniform and that XREPOTEST exposes distinct capability gaps.

Language-specific performance and challenges: ***Rust.*** Results highlight a scaffolding bottleneck: models frequently fail to assemble correct imports/crates and satisfy strict type and ownership constraints. Even strong systems often reach the focal method but do not reliably produce behavior-checking tests. Under the standard setting, Claude 4.5 Sonnet achieves the best Rust performance (TPR 12.78%, Cov 11.56%, IR 75.94%), while GPT-5.2 reaches TPR 12.24% and Cov 9.31%.
Go. Models achieve higher executed coverage than Rust, but passing tests remain bottlenecked by correct oracles and setup, indicating semantic correctness is the primary limiter. GPT-5.2 leads on TPR (25.78%) while Claude 4.5 Sonnet achieves the highest Cov (43.53%), with both models maintain-

ing near-perfect IR (≥98%).

Julia. Performance is moderate: Claude 4.5 Sonnet achieves the best TPR (19.88%) and GPT-5.2 the best Cov (31.92%), while small models (e.g., Qwen3-8B at TPR 0.88%) lag far behind. IR is high for most models (≥87%), though smaller models such as Qwen3-8B (73.83%) are exceptions, indicating that focal method invocation is generally not the bottleneck — semantic correctness is.

Ruby. Results are notably split: GPT-5.2 leads on TPR (25.78%) but achieves only 7.80% Cov, while Claude 4.5 Sonnet leads on Cov (14.25%) with a much lower TPR (6.37%). This TPR–Cov divergence suggests models can exercise code paths without producing passing assertions, reflecting the difficulty of Ruby's DSL-heavy testing conventions.

PHP. Claude 4.5 Sonnet dominates with TPR 26.93% and Cov 41.76%, while GPT-5.2 (TPR 10.06%) and open-weight models trail significantly. IR is near-perfect (≥97%) for most models, though smaller models such as Qwen3-8B (22.91%) and Kimi-k2.5 (88.08%) are notable exceptions, making PHP the language where invocation is easiest for capable models but test correctness remains the challenge.

Unit Test Quality vs Software Engineering Capability. We observe strong positive correlations between SWE-bench (Jimenez et al., 2024) performance and TPR/Cov, suggesting partial transfer from general software engineering capability to test generation. However, the correlation is incomplete, motivating complementary metrics such as IR. Full analysis is in Appendix E.4.

5.2 Impact of Context Augmentation

Building on the earlier analyses, we observe that models excel in domains with standalone functions but struggle when broader project-level context is required (Figure 1). This missing context often leads to *API Hallucination* (Table 4), as the model cannot fully resolve dependencies or references. In this section, we investigate context augmentation strategies; more implementation details are described in Appendix B.

Retrieval-based Augmentation. Retrieval methods show mixed results across languages and models (Table 3). For Go, Julia, and PHP with Claude 4.5 Sonnet, both sparse and dense retrieval improve TPR and Cov over the standard setting, but these gains are not universal: GPT-5.2 and GPT-OSS show inconsistent improvements, and Ruby exhibits IR drops despite some TPR/Cov gains. This variability suggests retrieval effectiveness depends heavily on model architecture and language characteristics.

Language	Setting	Claude 4.5 Sonnet			GPT-5.2			GPT-OSS		
		TPR	Cov	IR	TPR	Cov	IR	TPR	Cov	IR
Rust	Standard	12.78	11.56	**75.94**	12.24	9.31	77.87	5.80	4.16	81.63
	Sparse	16.00	15.29	69.07	15.36	13.01	83.24	4.94	4.56	82.28
	Dense	17.19	15.52	71.75	13.86	12.40	**84.32**	4.51	4.94	**82.38**
	File	**21.16**	**18.93**	68.53	**17.94**	**17.43**	83.35	**8.06**	**6.67**	80.56
	LSP	13.53	13.24	72.18	11.49	7.04	83.78	6.44	4.98	82.28
Go	Standard	23.65	43.53	**99.43**	25.78	41.30	98.44	19.26	26.95	88.24
	Sparse	28.05	46.95	97.17	28.90	43.10	**99.15**	18.98	27.20	95.61
	Dense	30.31	49.17	97.59	32.15	46.42	98.87	17.71	27.90	95.75
	File	**34.14**	**52.63**	97.88	**37.68**	**58.11**	98.87	**25.92**	**39.99**	**96.74**
	LSP	25.35	43.42	97.03	28.05	42.50	98.44	17.42	27.31	96.60
Julia	Standard	19.88	31.58	94.01	16.52	31.92	91.81	4.39	24.27	93.86
	Sparse	19.15	31.35	94.01	11.40	30.53	93.86	6.43	27.32	93.27
	Dense	18.57	31.21	94.44	11.70	30.18	93.86	**10.09**	26.96	93.27
	File	**23.98**	**34.83**	94.15	**17.25**	**34.35**	93.71	9.50	**32.07**	92.25
	LSP	15.64	29.22	**95.03**	12.87	25.83	**94.01**	5.85	20.29	**94.74**
Ruby	Standard	6.37	14.25	**85.63**	**25.78**	7.80	88.89	6.96	8.21	85.93
	Sparse	10.37	18.86	71.11	19.85	11.15	87.41	17.14	13.56	**87.14**
	Dense	11.26	20.97	68.15	21.04	14.82	85.33	**18.57**	15.24	85.71
	File	**16.30**	**23.51**	53.19	13.04	**18.35**	60.89	13.71	**26.71**	64.57
	LSP	7.70	14.55	85.19	23.85	7.95	**89.19**	7.56	7.93	86.37
PHP	Standard	26.93	41.76	99.38	10.06	23.17	99.38	3.41	5.07	99.23
	Sparse	30.03	45.09	99.38	13.16	29.39	98.76	3.41	3.21	**99.85**
	Dense	**31.42**	45.14	99.38	16.10	30.73	**99.69**	4.95	4.63	99.23
	File	**31.42**	**47.25**	**99.54**	**22.91**	**38.64**	99.54	**6.35**	5.64	99.54
	LSP	26.78	44.62	99.07	11.76	26.68	99.38	4.49	**5.65**	99.69

Table 3: Unit test generation performance across context settings for three frontier models. Best scores per language (within each metric) are bolded and highlighted.

Static Context Augmentation. File-level context is the most consistently effective strategy, achieving the best results for Go, Julia, and Rust across all settings (e.g., Claude: 34.14% TPR, 52.63% Cov on Go; 23.98% TPR, 34.83% Cov on Julia; 21.16% TPR, 18.93% Cov on Rust). However, on Ruby, file-level context sharply reduces IR (from 85.63% to 53.19%) despite improving TPR and coverage — a systematic trade-off across all three evaluated models (IR drops of 21–32%), specific to Ruby's DSL-heavy, metaprogramming-rich ecosystem. Go and PHP show negligible IR changes (±2%) for the top two models under the same setting, though GPT-OSS shows a larger IR gain in Go (+8.5%). Julia IR changes are similarly small (≤2%) across all three models. LSP-based augmentation mirrors developer workflows but benefits are selective; models do not consistently leverage fine-grained dependency traces, and overly resolved context can become noisy.

Key Takeaways. File-level context is the most consistently effective augmentation, while retrieval and LSP-based approaches yield selective, language-dependent gains. Richer context can reduce IR, highlighting a trade-off between test ef-

Failure Mode	Rust		Go		Julia		Ruby		PHP	
	Std.	**Ctx.**	**Std.**	**Ctx.**	**Std.**	**Ctx.**	**Std.**	**Ctx.**	**Std.**	**Ctx.**
Syntactic & Compilation	3.94%	2.58%	24.98%	20.35%	15.55%	8.72%	1.68%	3.88%	17.23%	12.07%
Type System & Memory	11.31%	14.82%	13.27%	8.45%	9.02%	8.33%	4.64%	2.29%	4.80%	2.94%
API Hallucination	48.05%	36.09%	18.89%	14.35%	47.90%	50.10%	42.17%	41.88%	35.55%	40.61%
Logic & Assertion	26.42%	30.79%	19.97%	24.27%	13.94%	15.94%	38.42%	37.35%	28.95%	24.15%
Test Design & Mocking	3.29%	4.98%	0.38%	0.99%	0.00%	0.00%	4.40%	0.59%	1.08%	0.67%
No Error / Pass	6.98%	10.74%	22.51%	31.59%	13.59%	16.91%	8.69%	14.01%	12.39%	19.56%

Table 4: Failure mode distribution across languages under standard (**Std.**) and file-level context (**Ctx.**) settings, computed as the fraction of total samples assigned to each failure mode. Rows sum to 100% within each column; the "No Error / Pass" row captures samples that pass all checks without any failure.

fectiveness and invocation reliability. Overall, even the best models achieve at most ∼27% TPR across all languages under the standard setting, confirming XREPOTEST as a challenging benchmark. Statistical significance of key model comparisons is confirmed via bootstrap resampling and McNemar's test (Appendix G).

5.3 Failure Modes Analysis

To analyze failure modes, we collect test execution logs and apply systematic categorization through refinement and manual inspection. This yields five classes: *Syntactic & Compilation*, *API Hallucination*, *Type System & Memory*, *Logic & Assertion*, and *Test Design & Mocking* — capturing recurring breakdown points across (i) runnable scaffolding, (ii) valid API usage, (iii) type/memory constraints, and (iv) behavioral correctness. Full definitions and taxonomy are in Appendix I.2.

Analysis of Results: Table 4 reports failure modes normalized by total samples, so each column sums to 100%. *API Hallucination* is the dominant failure across most languages. In Rust, context augmentation reduces hallucinations but raises *Logic & Assertion* failures, indicating that richer context helps models produce runnable tests yet semantic correctness remains a bottleneck. In Julia, API hallucination slightly increases under context (47.90% → 50.10%), suggesting that additional context does not reliably ground Julia API usage and semantic correctness remains the primary challenge. Ruby's hallucination rate is largely unchanged by context. Go shows a more distributed profile: context reduces both hallucination and compilation errors, but *Logic & Assertion* rises, confirming that oracle correctness is the primary limiter once scaffolding issues are resolved. PHP is an exception — context reduces compilation failures but increases API hallucination, suggesting that richer context introduces more opportunities to misuse framework-specific APIs.

Overall, improving API grounding and oracle quality is key to better test success. More examples and discussion are in Appendix I.

6 Conclusion

In this work, we introduced XREPOTEST, a multilingual repository-level benchmark and evaluation framework for unit test generation that emphasizes realistic development settings, controlled context augmentation, and execution-based metrics. Covering multiple programming languages, diverse domains, and containerized environments, XREPOTEST enables systematic analysis of how modern LLMs generate, invoke, and validate tests under different languages. Our evaluation of 14 state-of-the-art LLMs reveals substantial challenges stemming from language-specific constraints, domain complexity, and semantic issues in generated tests. To better assess behavioral validity, we proposed the novel Invocation Rate (IR) metric, complementing standard metrics like test pass rate (TPR) and coverage. We further explored the impact of context augmentation strategies on test generation. While static file-level context provides the most reliable gains across several languages, whereas more sophisticated augmentations such as LSP- or retrieval-based context yield selective and language-dependent benefits. Together, these findings position XREPOTEST as a challenging and informative benchmark for advancing scalable and robust multilingual unit test generation in realistic repository contexts, while highlighting promising directions for future work in adaptive context selection, language-aware augmentation, and stronger semantic reasoning for test oracle construction.

7 Limitations

In this work, for comparability, we evaluate all models using a standardized prompt format and fixed decoding configuration. Alternative prompting strategies, decoding schemes, or tool-augmented settings may yield different absolute performance and could better exploit specific models. Similarly, retrieval-based augmentation is sensitive to retriever design choices (e.g., chunking strategy, indexing granularity, and relevance scoring), and our retrieval-based context results do not necessarily reflect an optimally tuned retrieval pipeline. Additionally, XREPOTEST focuses on five underexplored yet impactful languages (Rust, Go, Julia, PHP, and Ruby) and a curated set of real-world repositories. While this scope enables meaningful multilingual, repository-level evaluation, the results may not directly generalize to other programming languages, application domains, or testing frameworks. To partially address this, we release an extensible, containerized evaluation pipeline designed to facilitate future expansion to additional languages, repositories, and test ecosystems.

References

Sandhini Agarwal, Lama Ahmad, Jason Ai, Sam Altman, Andy Applebaum, Edwin Arbus, Rahul K Arora, Yu Bai, Bowen Baker, Haiming Bao, and 1 others. 2025. gpt-oss-120b & gpt-oss-20b model card. *arXiv preprint arXiv:2508.10925*.

Saranya Alagarsamy, Chakkrit Tantithamthavorn, and Aldeida Aleti. 2024. A3test: Assertion-augmented automated test case generation. *Information and Software Technology*, 176:107565.

Jacob Austin, Augustus Odena, Maxwell Nye, Maarten Bosma, Henryk Michalewski, David Dohan, Ellen Jiang, Carrie Cai, Michael Terry, Quoc Le, and 1 others. 2021. Program synthesis with large language models. *arXiv preprint arXiv:2108.07732*.

Shreya Bhatia, Tarushi Gandhi, Dhruv Kumar, and Pankaj Jalote. 2024. Unit test generation using generative ai: A comparative performance analysis of autogeneration tools. In *Proceedings of the 1st International Workshop on Large Language Models for Code*, pages 54–61.

Max Brunsfeld. 2018. Tree-sitter: a parser generator tool and an incremental parsing library. https://tree-sitter.github.io/.

Ruisheng Cao, Mouxiang Chen, Jiawei Chen, Zeyu Cui, Yunlong Feng, Binyuan Hui, Yuheng Jing, Kaixin Li, Mingze Li, Junyang Lin, Zeyao Ma, Kashun Shum, Xuwu Wang, Jinxi Wei, Jiaxi Yang, Jiajun Zhang, Lei Zhang, Zongmeng Zhang, Wenting Zhao, and Fan Zhou. 2026. Qwen3-coder-next technical report. *Preprint*, arXiv:2603.00729.

Arda Celik and Qusay H Mahmoud. 2025. A review of large language models for automated test case generation. *Machine Learning and Knowledge Extraction*, 7(3):97.

Linzheng Chai, Shukai Liu, Jian Yang, Yuwei Yin, Jiaheng Liu, Tao Sun, Ge Zhang, Changyu Ren, Hongcheng Guo, Noah Wang, and 1 others. 2025. Mceval: Massively multilingual code evaluation. In *The Thirteenth International Conference on Learning Representations (ICLR)*.

Mark Chen, Jerry Tworek, Heewoo Jun, Qiming Yuan, Henrique Ponde De Oliveira Pinto, Jared Kaplan, Harri Edwards, Yuri Burda, Nicholas Joseph, Greg Brockman, and 1 others. 2021. Evaluating large language models trained on code. *arXiv preprint arXiv:2107.03374*.

Yinghao Chen, Zehao Hu, Chen Zhi, Junxiao Han, Shuiguang Deng, and Jianwei Yin. 2024. Chatunitest: A framework for llm-based test generation. In *Companion Proceedings of the 32nd ACM International Conference on the Foundations of Software Engineering*, pages 572–576.

Zhi Chen and Lingxiao Jiang. 2024. Promise and peril of collaborative code generation models: Balancing effectiveness and memorization. In *Proceedings of the 39th IEEE/ACM International Conference on Automated Software Engineering*, pages 493–505.

Xiang Cheng, Fan Sang, Yizhuo Zhai, Xiaokuan Zhang, and Taesoo Kim. 2025. Rug: Turbo llm for rust unit test generation. In *2025 IEEE/ACM 47th International Conference on Software Engineering (ICSE)*, pages 634–634. IEEE Computer Society.

Zoe Collins, Luigi Di Paolo, Cathal O'Grady, Niall Ryan, Gerard Marks, Murat Yilmaz, Paul M Clarke, and Andrew McCarren. 2025. Programming language selection in software engineering: Results from an adapted mlr focused on go, haskell, python and rust. In *European Conference on Software Process Improvement*, pages 176–193. Springer.

Arghavan Moradi Dakhel, Amin Nikanjam, Vahid Majdinasab, Foutse Khomh, and Michel C Desmarais. 2024. Effective test generation using pre-trained large language models and mutation testing. *Information and Software Technology*, 171:107468.

DeepSeek-AI, Aixin Liu, Aoxue Mei, Bangcai Lin, Bing Xue, Bingxuan Wang, Bingzheng Xu, Bochao Wu, Bowei Zhang, Chaofan Lin, Chen Dong, Chengda Lu, Chenggang Zhao, Chengqi Deng, Chenhao Xu, Chong Ruan, Damai Dai, Daya Guo, Dejian Yang, and 245 others. 2025. Deepseek-v3.2: Pushing the frontier of open large language models. *Preprint*, arXiv:2512.02556.

Mingzhe Du, Anh Tuan Luu, Bin Ji, Xiaobao Wu, Yuhao Qing, Dong Huang, Terry Yue Zhuo, Qian Liu, and See-Kiong Ng. 2025. CodeArena: A collective evaluation platform for LLM code generation. In *Proceedings of the 63rd Annual Meeting of the Association for Computational Linguistics (Volume 3: System Demonstrations)*, pages 502–512, Vienna, Austria. Association for Computational Linguistics.

Gordon Fraser and Andrea Arcuri. 2011. Evosuite: automatic test suite generation for object-oriented software. In *Proceedings of the 19th ACM SIGSOFT symposium and the 13th European conference on Foundations of software engineering*, pages 416–419.

Gordon Fraser and Andrea Arcuri. 2025. A retrospective on whole test suite generation: On the role of sbst in the age of llms. *IEEE Transactions on Software Engineering*.

Lennard Gäher, Michael Sammler, Ralf Jung, Robbert Krebbers, and Derek Dreyer. 2024. Refinedrust: A type system for high-assurance verification of rust programs. *Proceedings of the ACM on Programming Languages*, 8(PLDI):1115–1139.

Alessandro Giagnorio, Alberto Martin-Lopez, and Gabriele Bavota. 2025. Enhancing code generation for low-resource languages: No silver bullet. In *2025 IEEE/ACM 33rd International Conference on Program Comprehension (ICPC)*, pages 478–488. IEEE.

GLM-5-Team, :, Aohan Zeng, Xin Lv, Zhenyu Hou, Zhengxiao Du, Qinkai Zheng, Bin Chen, Da Yin, Chendi Ge, Chenghua Huang, Chengxing Xie, Chenzheng Zhu, Congfeng Yin, Cunxiang Wang, Gengzheng Pan, Hao Zeng, Haoke Zhang, Haoran Wang, and 168 others. 2026. Glm-5: from vibe coding to agentic engineering. *Preprint*, arXiv:2602.15763.

Gwihwan Go, Quan Zhang, Chijin Zhou, Zhao Wei, and Yu Jiang. 2025. Lsprag: Lsp-guided rag for language-agnostic real-time unit test generation. *arXiv preprint arXiv:2510.22210*.

Philip Gorinski, Matthieu Zimmer, Gerasimos Lampouras, Derrick Goh Xin Deik, and Ignacio Iacobacci. 2023. Automatic unit test data generation and actor-critic reinforcement learning for code synthesis. In *Findings of the Association for Computational Linguistics: EMNLP 2023*, pages 370–384, Singapore. Association for Computational Linguistics.

Daya Guo, Shuai Lu, Nan Duan, Yanlin Wang, Ming Zhou, and Jian Yin. 2022. Unixcoder: Unified cross-modal pre-training for code representation. In *Proceedings of the 60th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pages 7212–7225.

Yifeng He, Jiabo Huang, Yuyang Rong, Yiwen Guo, Ethan Wang, and Hao Chen. 2024. Unitsyn: A large-scale dataset capable of enhancing the prowess of large language models for program testing. In *Proceedings of the 33rd ACM SIGSOFT International Symposium on Software Testing and Analysis*, pages 1061–1072.

Md. Ashraful Islam, Mohammed Eunus Ali, and Md Rizwan Parvez. 2024. MapCoder: Multi-agent code generation for competitive problem solving. In *Proceedings of the 62nd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pages 4912–4944, Bangkok, Thailand. Association for Computational Linguistics.

Kush Jain, Gabriel Synnaeve, and Baptiste Roziere. 2025a. Testgeneval: A real world unit test generation and test completion benchmark. In *The Thirteenth International Conference on Learning Representations*.

Naman Jain, King Han, Alex Gu, Wen-Ding Li, Fanjia Yan, Tianjun Zhang, Sida Wang, Armando Solar-Lezama, Koushik Sen, and Ion Stoica. 2025b. Livecodebench: Holistic and contamination free evaluation of large language models for code. In *The Thirteenth International Conference on Learning Representations*.

Carlos E Jimenez, John Yang, Alexander Wettig, Shunyu Yao, Kexin Pei, Ofir Press, and Karthik R Narasimhan. 2024. Swe-bench: Can language models resolve real-world github issues? In *ICLR*.

Sathvik Joel, Jie Wu, and Fatemeh Fard. 2024. A survey on llm-based code generation for low-resource and domain-specific programming languages. *ACM Transactions on Software Engineering and Methodology*.

Nam Le Hai, Dung Manh Nguyen, and Nghi DQ Bui. 2025. On the impacts of contexts on repository-level code generation. In *Findings of the Association for Computational Linguistics: NAACL 2025*, pages 1496–1524.

Caroline Lemieux, Jeevana Priya Inala, Shuvendu K Lahiri, and Siddhartha Sen. 2023. Codamosa: Escaping coverage plateaus in test generation with pre-trained large language models. In *2023 IEEE/ACM 45th International Conference on Software Engineering (ICSE)*, pages 919–931. IEEE.

Kefan Li and Yuan Yuan. 2024. Large language models as test case generators: Performance evaluation and enhancement. *arXiv preprint arXiv:2404.13340*.

Anton Lozhkov, Raymond Li, Loubna Ben Allal, Federico Cassano, Joel Lamy-Poirier, Nouamane Tazi, Ao Tang, Dmytro Pykhtar, Jiawei Liu, Yuxiang Wei, and 1 others. 2024. Starcoder 2 and the stack v2: The next generation. *arXiv preprint arXiv:2402.19173*.

Stephan Lukasczyk and Gordon Fraser. 2022. Pynguin: Automated unit test generation for python. In *Proceedings of the ACM/IEEE 44th International Conference on Software Engineering: Companion Proceedings*, pages 168–172.

Zeyao Ma, Xiaokang Zhang, Jing Zhang, Jifan Yu, Sijia Luo, and Jie Tang. 2025. Dynamic scaling of unit tests for code reward modeling. In *Proceedings of the 63rd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pages 6917–6935.

Partha Sarathi Mohapatra. 2025. Artificial intelligence-driven test case generation in software development. In *Intelligent Assurance: Artificial Intelligence-Powered Software Testing in the Modern Development Lifecycle*, volume 4, page 38. Deep Science Publishing.

Thor Myklebust, Christian Askeland, and Espen Helle. 2025. Enhancing software safety through programming languages: A study of rust. In *Conference paper, ESREL, Norway*.

Yannic Noller, Rody Kersten, and Corina S Păsăreanu. 2018. Badger: complexity analysis with fuzzing and symbolic execution. In *Proceedings of the 27th ACM SIGSOFT international symposium on software testing and analysis*, pages 322–332.

Saahil Ognawala, Fabian Kilger, and Alexander Pretschner. 2019. Compositional fuzzing aided by targeted symbolic execution. *arXiv preprint arXiv:1903.02981*.

Rangeet Pan, Myeongsoo Kim, Rahul Krishna, Raju Pavuluri, and Saurabh Sinha. 2025. Aster: Natural and multi-language unit test generation with llms. In *2025 IEEE/ACM 47th International Conference on Software Engineering: Software Engineering in Practice (ICSE-SEIP)*, pages 413–424. IEEE.

Nikitha Rao, Kush Jain, Uri Alon, Claire Le Goues, and Vincent J Hellendoorn. 2023. Cat-lm training language models on aligned code and tests. In *2023 38th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 409–420. IEEE.

Michael D. Ernst René Just, Darioush Jalali. 2014. Defects4j: a database of existing faults to enable controlled testing studies for java programs. In *Proceedings of the 2014 International Symposium on Software Testing and Analysis*, pages 437 – 440. Association for Computing Machinery.

Max Schäfer, Sarah Nadi, Aryaz Eghbali, and Frank Tip. 2023. An empirical evaluation of using large language models for automated unit test generation. *IEEE Transactions on Software Engineering*, 50(1):85–105.

Domenico Serra, Giovanni Grano, Fabio Palomba, Filomena Ferrucci, Harald C Gall, and Alberto Bacchelli. 2019. On the effectiveness of manual and automatic unit test generation: ten years later. In *2019 IEEE/ACM 16th International Conference on Mining Software Repositories (MSR)*, pages 121–125. IEEE.

Mohammed Latif Siddiq, Joanna Cecilia Da Silva Santos, Ridwanul Hasan Tanvir, Noshin Ulfat, Fahmid Al Rifat, and Vinícius Carvalho Lopes. 2024. Using large language models to generate junit tests: An empirical study. In *Proceedings of the 28th International Conference on Evaluation and Assessment in Software Engineering*, EASE '24, page 313–322, New York, NY, USA. Association for Computing Machinery.

Kimi Team, Tongtong Bai, Yifan Bai, Yiping Bao, S. H. Cai, Yuan Cao, Y. Charles, H. S. Che, Cheng Chen, Guanduo Chen, Huarong Chen, Jia Chen, Jiahao Chen, Jianlong Chen, Jun Chen, Kefan Chen, Liang Chen, Ruijue Chen, Xinhao Chen, and 307 others. 2026. Kimi k2.5: Visual agentic intelligence. *Preprint*, arXiv:2602.02276.

Michele Tufano, Shao Kun Deng, Neel Sundaresan, and Alexey Svyatkovskiy. 2022a. Methods2test: A dataset of focal methods mapped to test cases. In *Proceedings of the 19th International Conference on Mining Software Repositories*, pages 299–303.

Michele Tufano, Dawn Drain, Alexey Svyatkovskiy, Shao Kun Deng, and Neel Sundaresan. 2020. Unit test case generation with transformers and focal context. *arXiv preprint arXiv:2009.05617*.

Michele Tufano, Dawn Drain, Alexey Svyatkovskiy, and Neel Sundaresan. 2022b. Generating accurate assert statements for unit test cases using pretrained transformers. In *Proceedings of the 3rd ACM/IEEE International Conference on Automation of Software Test*, pages 54–64.

Wenhan Wang, Chenyuan Yang, Zhijie Wang, Yuheng Huang, Zhaoyang Chu, Da Song, Lingming Zhang, An Ran Chen, and Lei Ma. 2025. Testeval: Benchmarking large language models for test case generation. In *Findings of the Association for Computational Linguistics: NAACL 2025*, pages 3547–3562.

Yanlin Wang, Yanli Wang, Daya Guo, Jiachi Chen, Ruikai Zhang, Yuchi Ma, and Zibin Zheng. 2024. Rlcoder: Reinforcement learning for repository-level code completion. In *2025 IEEE/ACM 47th International Conference on Software Engineering (ICSE)*, pages 165–177. IEEE Computer Society.

Yue Wang, Hung Le, Akhilesh Deepak Gotmare, Nghi DQ Bui, Junnan Li, and Steven Hoi. 2023. Codet5+: Open code large language models for code understanding and generation. In *The 2023 Conference on Empirical Methods in Natural Language Processing*.

Cody Watson, Michele Tufano, Kevin Moran, Gabriele Bavota, and Denys Poshyvanyk. 2020. On learning meaningful assert statements for unit test cases. In *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering*, pages 1398–1409.

Di Wu, Wasi Uddin Ahmad, Dejiao Zhang, Murali Krishna Ramanathan, and Xiaofei Ma. 2024. Repoformer: selective retrieval for repository-level code completion. In *Proceedings of the 41st International Conference on Machine Learning*, pages 53270–53290.

Weimin Xiong, Yiwen Guo, and Hao Chen. 2024. The program testing ability of large language models for code. In *Proceedings of the 2024 Conference on Empirical Methods in Natural Language Processing: Industry Track*, pages 23–34, Miami, Florida, US. Association for Computational Linguistics.

Ruijie Xu, Zengzhi Wang, Run-Ze Fan, and Pengfei Liu. 2024. Benchmarking benchmark leakage in large language models. *arXiv preprint arXiv:2404.18824*.

Chen Yang, Junjie Chen, Bin Lin, Ziqi Wang, and Jianyi Zhou. 2026. Advancing code coverage: Incorporating program analysis with large language models. *ACM Trans. Softw. Eng. Methodol.*, 35(5).

Lin Yang, Chen Yang, Shutao Gao, Weijing Wang, Bo Wang, Qihao Zhu, Xiao Chu, Jianyi Zhou, Guangtai Liang, Qianxiang Wang, and 1 others. 2024. On the evaluation of large language models in unit test generation. In *Proceedings of the 39th IEEE/ACM International Conference on Automated Software Engineering*, pages 1607–1619.

Zhiqiang Yuan, Mingwei Liu, Shiji Ding, Kaixin Wang, Yixuan Chen, Xin Peng, and Yiling Lou. 2024. Evaluating and improving chatgpt for unit test generation. *Proceedings of the ACM on Software Engineering*, 1(FSE):1703–1726.

Zhiqiang Yuan, Yiling Lou, Mingwei Liu, Shiji Ding, Kaixin Wang, Yixuan Chen, and Xin Peng. 2023. No more manual tests? evaluating and improving chatgpt for unit test generation. *arXiv preprint arXiv:2305.04207*.

Fengji Zhang, Bei Chen, Yue Zhang, Jacky Keung, Jin Liu, Daoguang Zan, Yi Mao, Jian-Guang Lou, and Weizhu Chen. 2023. Repocoder: Repository-level code completion through iterative retrieval and generation. In *Proceedings of the 2023 Conference on Empirical Methods in Natural Language Processing*, pages 2471–2484.

Linghao Zhang, Junhao Wang, Shilin He, Chaoyun Zhang, Yu Kang, Bowen Li, Jiaheng Wen, Chengxing Xie, Maoquan Wang, Yufan Huang, Elsie Nallipogu, Qingwei Lin, Yingnong Dang, Saravan Rajmohan, Dongmei Zhang, and Qi Zhang. 2025a. DI-BENCH: Benchmarking large language models on dependency inference with testable repositories at scale. In *Findings of the Association for Computational Linguistics: ACL 2025*, pages 10134–10153, Vienna, Austria. Association for Computational Linguistics.

Zehua Zhang, Ati Priya Bajaj, Divij Handa, Siyu Liu, Arvind S Raj, Hongkai Chen, Hulin Wang, Yibo Liu, Zion Leonahenahe Basque, Souradip Nath, Vishal Juneja, Nikhil Chapre, Yan Shoshitaishvili, Adam Doupé, Chitta Baral, and Ruoyu Wang. 2025b. Buildbench: Benchmarking llm agents on compiling real-world open-source software. *Preprint*, arXiv:2509.25248.

Zhe Zhang, Xingyu Liu, Yuanzhang Lin, Xiang Gao, Hailong Sun, and Yuan Yuan. 2024. Llm-based unit test generation via property retrieval. *arXiv preprint arXiv:2410.13542*.

Qinkai Zheng, Xiao Xia, Xu Zou, Yuxiao Dong, Shan Wang, Yufei Xue, Lei Shen, Zihan Wang, Andi Wang, Yang Li, and 1 others. 2023. Codegeex: A pre-trained model for code generation with multilingual benchmarking on humaneval-x. In *Proceedings of the 29th ACM SIGKDD Conference on Knowledge Discovery and Data Mining*, pages 5673–5684.

Appendix

A Detailed Statistics of XRepoTest

Table A.1 provides a summary of XRepoTest. The benchmark includes 3,642 focal methods across five languages: Julia (684), Rust (931), Go (706), PHP (646), and Ruby (675). Despite having fewer samples, Rust and Go exhibit significantly longer average focal lengths, reflecting their more verbose function implementations. From the Table, we also observe the domain distribution of the benchmark across the five languages. Julia primarily focuses on scientific computing tasks owing to its design and use cases, with additional coverage in data structures and algorithms (DSA). Rust is largely oriented toward developer tools with additional coverage in DSA, scientific computing, and web infrastructure. Go spans developer tools, web/framework, and data structure domains. PHP contributes functions from testing, utilities, and web development domains, while Ruby emphasizes testing frameworks and web development with smaller representation in utilities and web security.

B Implementation Details

This section summarizes implementation-level details that are omitted from the main paper for brevity.

Focal Function Filtering Rules. To ensure that extracted functions are non-trivial and representative of real-world unit testing targets, we apply systematic filtering criteria during the focal method extraction phase. Table A.2 summarizes these rules.

LSP-based Context. To enrich prompts with type-aware context, we run a Language Server Protocol[1] (LSP) analysis for each repository. Our pipeline implements a lightweight LSP client that speaks the standard JSON-RPC message format and issues the typical LSP lifecycle calls (e.g., starting/initializing the server, opening the focal file, and querying symbol and type information). Concretely, we use LSP to retrieve three kinds of information centered on the focal method.

- First, for each focal-method argument, we resolve its definition (e.g., local binding, field, imported symbol, or type alias) and then collect the corresponding type declarations as well as any directly referenced dependent types (e.g., nested structs/classes, trait/interface bounds, and enum/union variants when applicable). This material is attached as "argument context".
- Second, we extract the function/method calls appearing in the focal method body and resolve the definitions of the callees (including overload-/dispatch targets when available). We then collect the signatures and relevant type declarations for these invoked functions and their receiver/-parameter types, providing the model with type-accurate information about the APIs that the focal method depends on.
- Third, we focus on identifiers that appear in the focal method's control-flow conditions (e.g., predicates in `if`/`while` branches and guards). For these tokens (variables, fields, constants, or function calls), we use LSP definition/type-definition and reference queries to recover their declaring sites and types, effectively retrieving "references for the focal method" that are most likely to constrain feasible tests (e.g., branch-specific preconditions).

Retrieval-augmented Context. We implement a retrieval-augmented context construction pipeline that enriches each XRepoTest function with repository context, following prior repository-level code generation works (Zhang et al., 2023; Wu et al., 2024; Wang et al., 2024). For each repository, we preprocess all source files into chunks/overlapping sliding windows of $W = 50$ lines. The stride s is computed as:

$$s = \left\lfloor \frac{W}{\text{slice_size}} \right\rfloor = \left\lfloor \frac{50}{10} \right\rfloor = 5 \text{ lines} \quad (1)$$

This creates 90% overlap between consecutive windows. Empty windows are dropped and duplicate windows (identical text) are merged while preserving multiple metadata entries (file path and line span).

We retrieve windows using the focal method code itself as the query. Specifically, we use the focal function (the complete function implementation) as the query string. This approach captures local implementation patterns and enables the retriever to find semantically and lexically similar code snippets from the repository that share similar logic, structure, or domain-specific patterns with the target function.

[1] https://microsoft.github.io/language-server-protocol/

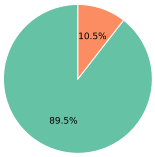
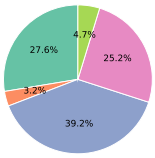
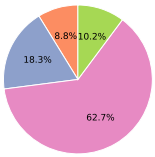
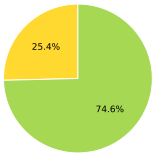
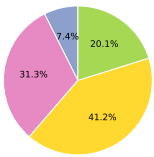
Language (#Sample)	Repo	Size	Avg. Focal Lines	Domain	Domain Distribution
Julia (684)	DataFrames.jl	267	26.99	Data Manipulation	
	Distributions.jl	105	19.89	Probability Distribution	
	StatsBase.jl	99	22.63	Statistical Computation	
	Turing.jl	81	24.83	Bayesian modelling	
	DataStructures.jl	72	25.79	Data structure and algorithms	
	QuantEcon.jl	60	26.03	Quantitative Economics	
Rust (931)	nom	278	22.86	Parser Combinator	
	Burn	257	47.49	Deep learning	
	hyper	235	27.99	HTTP Library	
	alacritty	54	49.59	Terminal emulator	
	uuid	44	21.84	UUID Generation	
	TheAlgorithms/Rust	30	34.30	Data structure and algorithms	
	ripgrep	18	40.17	Command-line search tool	
	starship	15	62.53	Terminal Prompt	
Go (706)	client_golang	262	28.54	Monitoring Instrumentation	
	Hugo	179	46.40	Static Site Generator	
	Ollama	114	49.81	Local LLM runner	
	zap	72	21.92	Logging Library	
	TheAlgorithms/Go	62	31.31	Data structure and algorithms	
	cli	12	33.33	Command line tool	
	Cobra	3	47.67	Library for building CLIs	
	chi	2	48.50	HTTP router	
PHP (646)	Faker	124	22.55	Testing & Mocking	
	Carbon	110	28.15	Date & Time Utilities	
	flysystem	110	17.48	Filesystem Abstraction	
	csv	94	19.80	CSV Processing	
	PHP-Parser	49	23.82	Code Parsing	
	monolog	45	24.69	Logging Library	
	PHPMailer	43	37.12	Email Library	
	mockery	40	18.38	Testing & Mocking	
	uuid	31	21.97	UUID Generation	
Ruby (675)	capybara	155	16.58	Web Testing	
	rspec-core	123	16.92	Testing Framework	
	rom	97	17.09	Data Mapping	
	grape	97	16.42	REST-like API Framework	
	hanami	81	19.38	Web Framework	
	shoryuken	50	17.76	Message Queue Worker	
	hashie	28	16.79	Hash Extensions	
	httparty	26	20.65	HTTP Client	
	dotenv	11	18.00	Configuration	
	pundit	7	13.29	Authorization	

Table A.1: Per-repository statistics with per-language domain distributions in XREPOTEST. There are 6 domains across the benchmark: Data Structure and Algorithm, Scientific Computing, Developer Tool, Web/Framework, Testing & Mocking, Utilities.

Retrieval is performed over split code chunks using two retrievers: BM25 as a sparse retriever and UniXCoder (Guo et al., 2022) as a dense retriever. After that, we attach the top-k ($k = 10$) retrieved contexts by each retriever to the input prompt.

- BM25 retrieval (BM25Okapi with $k_1 = 1.5$, $b = 0.75$) uses whitespace tokenization and ranks chunks by lexical relevance:

Rule	Description	Purpose
Minimum number of lines	Function body must contain at least 10 lines of code	Avoid trivial or uninformative functions (e.g., empty methods or those offering trivial coverage)
Return statement	Function must include an explicit `return` or output expression	Ensure testable behavior and observable outputs
Non-test function	Exclude functions already written as unit tests	Ensuring only production code is selected as focal functions
Valid parsing	Function must be parsable by language grammar (no syntax errors)	Guarantee valid function and clean dataset for model evaluation

Table A.2: Filtering rules used to select focal functions for unit test generation.

$$\mathrm{BM25}(q, w) = \sum_{t \in q} \mathrm{IDF}(t) \cdot \frac{f(t,w)(k_1+1)}{f(t,w) + k_1\left(1 - b + b\frac{|w|}{\mathrm{avgdl}}\right)} \quad (2)$$

where $f(t, w)$ is the term frequency, $|w|$ is the chunk length in tokens, and avgdl is the average window length.

- UniXcoder encodes the query and chunks (max length 512 tokens) with attention-masked mean pooling and ranks by cosine similarity:

$$\mathrm{sim}(q, w) = \frac{\mathbf{e}_q \cdot \mathbf{e}_w}{\|\mathbf{e}_q\|\|\mathbf{e}_w\|} \quad (3)$$

where $\mathbf{e}_q$ and $\mathbf{e}_w$ are the embeddings.

Containerized provisioning and execution. We evaluate each repository inside a Docker container to standardize toolchains and ensure reproducible execution. Containers provide (i) the language runtime/toolchain and package manager, (ii) the project dependencies, (iii) the language's native test runner, and (iv) coverage instrumentation where available. At evaluation time, the repository is mounted into the container, tests are integrated following language conventions, and we run the corresponding test command (e.g., `cargo test`, `go test`, `Test.jl`, `phpunit`, `rspec`) to collect execution outcomes and metrics. We use the following standard test runners and entrypoints: Go's `go test`[2], Rust's `cargo test`[3], Julia's `Test.jl`[4], PHP's PHPUnit [5], and Ruby's RSpec [6].

To robustly handle the often noisy or partially structured outputs produced by LLMs, we adopt a lightweight post-processing strategy that extracts generated test code from fenced code blocks (e.g., "'language... "'). The extracted tests are then integrated into the target repository following language- and framework-specific conventions. For example, Go tests are written to files with the `_test.go` suffix, while Rust test functions are appended to the corresponding source files containing the focal functions. Depending on the language, tests may be injected into existing files or written into newly created test files.

After integrating the generated tests, we execute an automated evaluation pipeline. First, the project is compiled to detect syntactic errors and unresolved dependencies. If compilation succeeds, the test suite is executed and code coverage is measured using language-specific tooling. The entire evaluation process is containerized with Docker to ensure reproducibility and consistency across environments. To support future research on LLM-based test generation, we release our evaluation framework publicly.

[2] https://pkg.go.dev/testing
[3] https://doc.rust-lang.org/cargo/commands/cargo-test.html
[4] https://docs.julialang.org/en/v1/stdlib/Test/
[5] https://phpunit.de/
[6] https://rspec.info/

C Data Contamination Risk

Large language models (LLMs) are typically trained on large-scale corpora collected from diverse online sources, with limited transparency regarding data curation and preprocessing. Consequently, constructing a benchmark that is entirely free from training-data leakage is widely considered impractical. Recent benchmarks therefore acknowledge this limitation and focus on mitigating and characterizing potential contamination rather than claiming strict guarantees, commonly adopting strategies such as temporal filtering to reduce overlap with model pretraining data (Jain et al., 2025a,b; Wang et al., 2025).

From our experiments in Section 5, the consistently low performance across models suggests that memorization is unlikely to be a dominant factor in XREPOTEST. To further assess the reliability of the benchmark, we follow prior works (Xu et al., 2024; Chen and Jiang, 2024) and conduct a perplexity-based analysis to estimate whether models may have previously observed code artifacts related to XREPOTEST. The underlying assumption is that content encountered during pretraining is likely to be assigned systematically lower perplexity than genuinely unseen code. In particular, our analysis focuses on repository test code, as generated unit tests are evaluated against existing project tests. Since aligned focal–test pairs are not directly available, we extract all test functions from the repositories included in XREPOTEST and compute their perplexity under the evaluated models. These scores serve as a proxy for assessing whether repository-specific test code may have been memorized during training. We compute perplexity scores using two code models (Yi-Coder-9B and Codestral-22B) across all five benchmark languages (Go, Rust, Ruby, PHP, Julia).

Perplexity measures the uncertainty of a language model when predicting the next token in a sequence, expressed as the exponentiated average negative log-likelihood:

$$\text{PPL}(\mathbf{X}) = \exp\left(-\frac{1}{t}\sum_{i=0}^{t}\log p_\theta(x_i|x_{<i})\right) \quad (4)$$

where $\mathbf{X} = [x_0, x_1, ..., x_t]$ denotes a tokenized sequence. Lower perplexity indicates the model is confident in predicting the test code, suggesting it has likely encountered similar patterns during training (potential data leakage).

From Table A.3, we observe substantial variation in perplexity across languages. Ruby shows notably higher perplexity (26.94–32.74) compared to Go, Rust, PHP, and Julia (3.52–6.11), suggesting models have significantly less exposure to Ruby test code during pretraining. These high perplexity values across all languages (3.52–32.74) indicate that models have not memorized the dataset, suggesting a low risk of data leakage.

D Evaluation Metrics

This section provides detailed formulations for the evaluation metrics used in XREPOTEST. All metrics are normalized by the total number of samples in the benchmark, enabling direct cross-model and cross-language comparison.

Test Pass Rate (TPR). TPR measures the proportion of test suites that execute successfully without runtime errors or assertion failures. Crucially, we adopt an *all-or-nothing* criterion: a test suite passes only when **all** test cases within it pass. Formally:

$$\text{TPR} = \frac{\sum_{i=1}^{N} \mathbf{1}[\text{all tests pass in suite}_i]}{N} \quad (5)$$

where N is the total number of samples (focal functions) and $\mathbf{1}[\cdot]$ is the indicator function. This strict definition prevents inflated pass rates from partially correct test suites and ensures that only fully executable, assertion-compliant test code contributes to TPR.

Invocation Rate (IR). IR quantifies the proportion of all generated test suites that successfully invoke the focal function. To compute IR, we first parse each generated test suite using Tree-sitter. For test suites that are syntactically valid, we perform static analysis to detect whether any test function contains a direct call to the focal function; test suites that cannot be parsed are treated as failing to invoke the function. Formally:

$$\text{IR} = \frac{\sum_{i=1}^{N} \mathbf{1}[\text{suite}_i \text{ invokes focal function}]}{N} \quad (6)$$

where N is the total number of generated test suites. By computing IR over the total set rather than just the parsable subset, this metric penalizes both syntax errors and logical failures to call the target method, providing a comprehensive measure of the model's targeting capability

Model	Our Benchmark						External Benchmarks		
	Go	Rust	Ruby	PHP	Julia	Avg	McEval	HumanEval	TestGenEval
Yi-Coder-9B	3.76	5.78	26.94	4.53	4.51	9.10	2.74	1.95	5.44
Codestral-22B	3.52	6.11	32.74	4.81	4.40	10.32	2.91	3.07	6.92

Table A.3: Perplexity of different models on XREPOTEST across 5 languages. Higher perplexity indicates less likelihood of data leakage.

Line Coverage (Cov). Line coverage measures the proportion of executable lines in the focal function that are exercised by the generated tests. We employ macro-averaged line coverage, computed as follows. For each sample, we execute the generated test suite with language-specific coverage instrumentation (`llvm-cov` for Rust, `go test -coverprofile` for Go, `Coverage.jl` for Julia, PHPUnit coverage for PHP, SimpleCov for Ruby). From the coverage report, we extract two quantities within the focal function's start and end line boundaries: the set of *executable* lines L_{total} (lines that the instrumentation marks as trackable, regardless of whether they were exercised), and the subset of *covered* lines L_{covered} (executable lines that were actually executed during the test run).

The coverage metric is then computed as:

$$\text{Cov} = \frac{1}{N} \sum_{i=1}^{N} \frac{L_{\text{covered},i}}{L_{\text{total},i}} \tag{7}$$

where $L_{\text{total},i}$ and $L_{\text{covered},i}$ are both obtained from the same coverage run for focal function i. This macro-averaged formulation weights each focal function equally, ensuring that coverage is not dominated by samples with disproportionately many executable lines. Coverage for samples with no available coverage data (e.g., due to compilation failure) is treated as 0%.

Mutation Score (MS). Following the methodology of TestGenEval (Jain et al., 2025a), Mutation score assesses the fault-detection capability of generated tests by measuring their ability to distinguish correct code from systematically injected faults (mutants). For each sample i, if mutation testing is executed, we apply a set of mutation operators (e.g., arithmetic operator replacement, relational operator flip, constant modification) to the focal function, generating mutants M_i. Each mutant is then tested: if the test suite detects the mutation (via assertion failure or runtime error), the mutant is considered *killed*; otherwise, it survives. We define the per-sample mutation score as $\text{ms}_i = \frac{|M_{i,\text{killed}}|}{|M_i|}$ when $|M_i| > 0$; if mutation testing is not available for sample i (e.g., due to compilation/test failures or missing mutation data), we set $\text{ms}_i = 0$. The reported **Mutation Score (MS)** averages this value over *all* samples:

$$\text{MS} = \frac{1}{N} \sum_{i=1}^{N} \text{ms}_i \tag{8}$$

where N is the total number of samples.

Mutation@Pass Score (MS@Pass). To isolate test oracle quality conditional on successful execution, we also report **Mutation@Pass Score (MS@Pass)**, computed *only* on samples where all tests pass (TPR$= 1$) *and* mutation testing is available ($|M_i| > 0$). Formally:

$$\text{MS@Pass} = \frac{1}{N_{\text{pass}}} \sum_{i \in \text{pass}} \frac{|M_{i,\text{killed}}|}{|M_i|} \tag{9}$$

where pass denotes the set of samples satisfying these criteria and $N_{\text{pass}} = |\text{pass}|$.

Aggregate Reporting. All metrics are reported as percentages normalized by their respective denominators (N for TPR, Cov, IR, and MS; N_{pass} for MS@Pass). This normalization enables fair comparison across models, languages, and context configurations, and ensures that metrics are interpretable as proportions of the total evaluation scope.

E Additional Experiment Results

E.1 Full Results of LLMs Performance

This section presents the complete set of experimental results for all models. Table A.4 summarizes the evaluation metrics, including test pass rate (TPR), line coverage (Cov), and invocation rate (IR). CSR values are omitted from this table and available upon request. Together, these metrics offer a comprehensive assessment of model performance, capturing both the syntactic correctness and the semantic quality of the generated tests across different programming languages.

Table A.4: Full standard-setting results on XREPOTEST. Values denote **CSR | TPR | Cov | IR**.

Model	**Rust**				**Julia**				**Go**				**Ruby**				**PHP**			
	CSR	TPR	Cov	IR	CSR	TPR	Cov	IR	CSR	TPR	Cov	IR	CSR	TPR	Cov	IR	CSR	TPR	Cov	IR
Small																				
Yi-Coder (9B)	26.42	1.50	1.71	90.23	94.44	4.82	19.16	90.50	9.63	2.41	6.48	91.36	98.67	1.48	9.37	86.81	99.38	3.56	17.68	98.76
Qwen3 (8B)	23.09	0.32	0.21	64.88	12.57	0.88	0.97	73.83	1.27	0.42	1.13	47.17	14.07	0.00	1.39	92.15	7.28	0.62	2.13	22.91
Codestral (22B)	27.82	2.69	2.97	84.85	96.49	2.05	21.17	91.23	20.25	3.82	14.52	99.58	99.26	0.15	7.35	87.26	98.76	5.57	23.01	99.69
Medium																				
GPT-OSS (120B)	29.86	5.80	4.16	81.63	70.91	4.39	24.27	93.86	31.02	19.26	26.95	88.24	97.19	6.96	8.21	85.93	51.86	3.41	5.07	99.23
Llama-3.3 (70B)	28.03	3.33	3.04	88.94	97.81	3.07	25.42	93.27	10.62	2.69	7.96	98.73	98.22	0.44	7.66	87.85	95.05	2.01	21.75	99.85
Qwen3-Coder-Next (80B)	39.63	7.20	3.52	56.71	90.50	5.12	17.38	91.08	20.11	6.66	16.74	96.88	91.41	1.19	6.88	87.26	91.95	5.11	22.43	97.68
Large																				
DeepSeek V3.2	28.14	2.79	3.06	77.66	63.74	3.95	21.94	89.47	17.99	5.24	14.91	59.07	84.30	1.48	10.01	84.30	68.73	5.42	20.20	93.34
MiniMax-M2.7	31.15	4.30	3.63	66.49	72.08	4.39	18.12	86.11	17.00	4.67	13.07	81.16	89.33	1.63	7.30	87.41	78.79	5.57	17.25	97.68
Flagship																				
DeepSeek V4-pro	30.93	6.66	4.66	74.22	73.10	11.40	26.83	89.77	25.50	13.88	21.66	63.46	83.70	8.89	9.15	84.74	85.29	13.00	22.19	96.13
GLM-5	30.72	6.44	2.52	70.68	51.17	5.41	13.54	89.47	10.34	5.10	9.48	75.78	86.67	3.11	8.50	86.96	75.23	13.16	24.98	99.07
Kimi-k2.5 (Instruct)	24.70	1.40	1.17	74.76	39.04	2.63	8.70	87.43	3.26	2.27	2.33	75.21	41.33	2.52	5.02	78.96	15.48	3.25	4.02	88.08
Claude 4.5 Haiku	34.69	8.27	3.73	80.34	93.86	12.57	23.96	92.69	31.73	10.91	26.87	99.43	99.11	4.30	10.57	87.11	98.92	14.86	34.68	99.54
Claude 4.5 Sonnet	41.46	12.78	11.56	75.94	98.25	19.88	31.58	94.01	52.27	23.65	43.53	99.43	99.56	6.37	14.25	85.63	98.76	26.93	41.76	99.38
GPT-5.2	38.78	12.24	9.31	77.87	84.21	16.52	31.92	91.81	47.59	25.78	41.30	98.44	98.22	25.78	7.80	88.89	97.68	10.06	23.17	99.38

E.2 Domain-Based Performance Analysis

To understand how model performance varies across different application domains, we analyze test pass rates across repository categories. We categorize each benchmark repository into one of six domains: *Scientific Computing*, *Data Structure & Algorithm*, *Developer Tool*, *Web/Framework*, *Testing & Mocking*, and *Utilities*. Figure A.1 presents a comprehensive comparison using spider charts. The left subplot shows ten models in standard mode (baseline setting without context enrichment), while the right subplot compares three representative models (GPT-5.2, Claude 4.5 Sonnet, and GPT-OSS 120B) in file context mode. Each axis represents a unique (language, domain) pair, and the radial distance indicates the test pass rate achieved by each model.

The visualization reveals several key insights. First, performance exhibits significant variation across domains even within the same language, suggesting that domain-specific characteristics (e.g., mathematical abstractions in scientific computing vs. string manipulation in utilities) present distinct challenges for test generation. Second, models show different strengths across domains: while proprietary frontier models (GPT-5.2, Claude 4.5 Sonnet) generally achieve higher test pass rates, open-source and smaller models demonstrate competitive performance in certain language-domain combinations. Comparing standard mode with file context mode reveals that contextual enrichment can significantly improve performance, particularly for domains requiring understanding of complex type relationships or API usage patterns.

A consistent pattern across models is that *Data Structures & Algorithms* domains achieve the strongest performance. This likely reflects the prevalence of such problems in existing code generation and evaluation benchmarks (Chen et al., 2021; Austin et al., 2021; Lozhkov et al., 2024; Wang et al., 2023, 2024; Jain et al., 2025b; Le Hai et al., 2025), leading to greater model familiarity with their structures and testing patterns. In contrast, more specialized or application-specific domains exhibit lower and more variable performance, indicating that they remain challenging and underexplored in current test generation research. Finally, the clustering of low-performing language–domain pairs across all models suggests inherent difficulty that persists regardless of model architecture or context scope, potentially driven by domain-specific complexity or limitations in automated test oracles.

E.3 Mutation Results

Table A.5 reports mutation-testing results for all 14 evaluated models across Go, Rust, and Ruby. Both Mutation Score (MS) and Mutation@Pass Score (MS@Pass) are included, providing complementary views on fault-detection capability and test oracle quality.

Mutation testing meaningfully differentiates model capabilities beyond pass rate. GPT-5.2 achieves the highest MS across all three languages (Go: 19.62%, Rust: 4.14%, Ruby: 6.29%), while Claude 4.5 Sonnet ranks second in Go MS (18.87%). Smaller models often exhibit low MS despite moderate pass rates, confirming that passing repository tests does not necessarily imply strong fault-detection capability.

MS@Pass scores are consistently higher than MS across all models and languages, showing that test oracles are effective when tests execute successfully — the primary bottleneck is test executability rather than assertion quality. The ranking patterns remain broadly consistent with our overall evaluation: stronger repository-level models also tend to achieve higher MS, reinforcing the benchmark's internal validity.

Regarding language coverage: we investigated mutation testing for all five languages, but ecosystem-specific tooling limitations constrained full coverage. Ruby's mutation framework struggles with dynamic dispatch and metaprogramming-heavy code, reducing the number of valid mutants; Ruby MS values should therefore be interpreted as lower-bound estimates. Julia's available mutation libraries are incompatible with the required language version and dependency environment. PHP's existing mutation tools operate at file-level granularity only, which would introduce substantial noise and invalidate focal-function-level analysis. Despite these constraints, mutation testing across three ecosystems (Go, Rust, Ruby) already represents broader multilingual coverage than most prior unit test generation benchmarks. Notably, even Jain et al. (2025a) highlights mutation score computation as a key technical contribution, despite focusing solely on Python — a language with comparatively mature mutation tooling. Implementing mutation analysis across multiple heterogeneous ecosystems introduces substantially greater engineering complexity, further strengthening the benchmark's practical and technical scope.

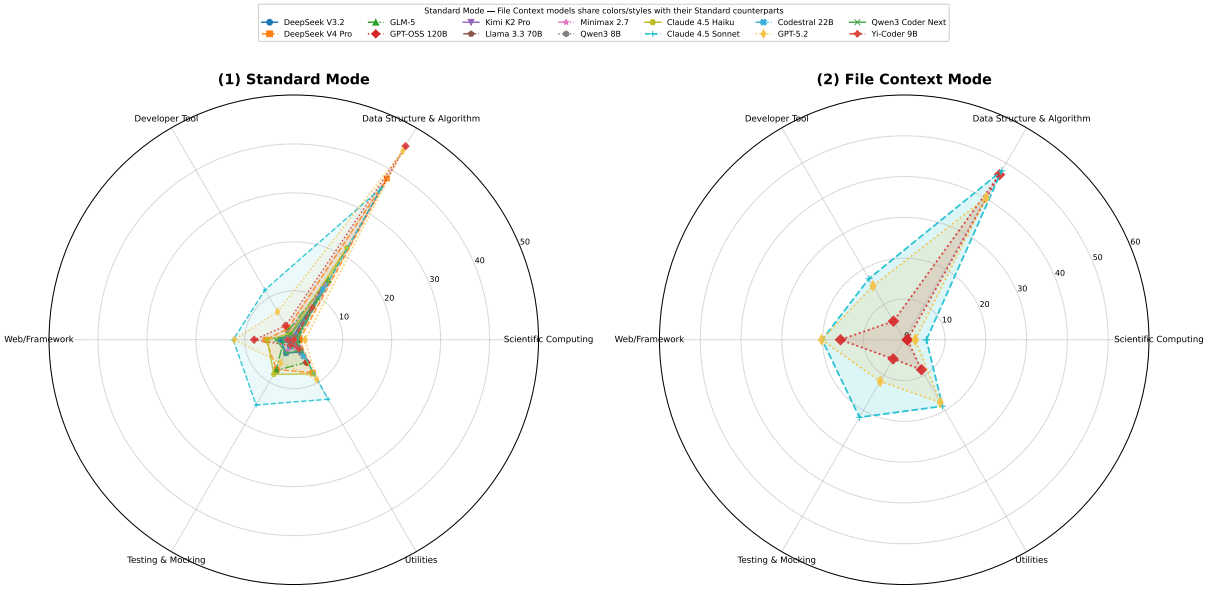


Figure A.1: Test pass rate comparison across language-domain combinations. Left (1): Ten models in standard mode including Qwen3-Coder-Next 80B, GPT-OSS 120B, Claude 4.5 Haiku, Claude 4.5 Sonnet, DeepSeek V3.2, GPT-5.2, DeepSeek V4-pro, GLM-5, MiniMax-M2.7, and Kimi-k2.5. Right (2): Three LLMs in file context mode showing the impact of contextual enrichment.

E.4 Correlation Analysis

We present two complementary correlation analyses: (i) correlations among XRepoTest's evaluation metrics across all evaluated models, and (ii) correlations between XRepoTest metrics and external code-generation performance on SWE-bench.

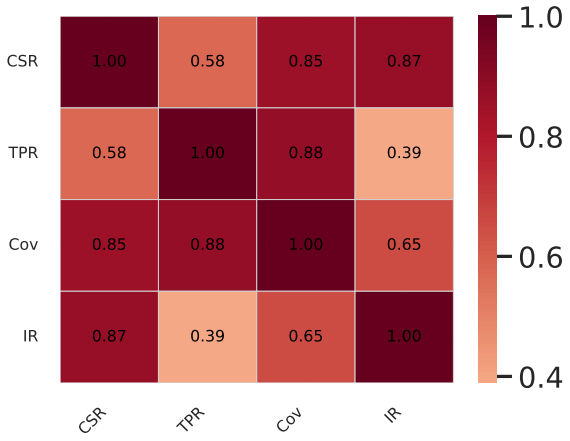


Figure A.2: Correlation matrix illustrating relationships between evaluation metrics on XRepoTest.

Correlations among XRepoTest metrics. Figure A.2 reports the pairwise Pearson correlations between code success rate (CSR), test pass rate (TPR), line coverage (Cov), and invocation rate (IR) across all 14 evaluated models on XRepoTest.

Two patterns stand out. First, CSR and Cov are strongly correlated ($r = 0.85$), and TPR and Cov are equally strongly correlated ($r = 0.88$), indicating that models which more reliably produce compilable/integrable tests also tend to achieve substantially higher coverage. CSR and TPR show moderate correlation ($r = 0.57$).

Second, and most importantly, IR captures a dimension that is clearly not redundant with the other metrics: it shows strong positive correlations with CSR ($r = 0.87$) and moderate positive correlation with Cov ($r = 0.65$), but only weak positive correlation with TPR ($r = 0.39$). This pattern indicates that while high IR generally accompanies high compilability and coverage, the relationship with test pass rates is much looser. A test suite can achieve high IR by consistently invoking the focal function, yet still exhibit low coverage or weak assertions—consistent with the presence of trivial or weak tests that pass without deeply exercising behavior.

Correlation with SWE-bench. To connect unit test generation and broader software engineering capability, we analyze correlations between model performance on XRepoTest and their SWE-bench Verified scores (Jimenez et al., 2024). SWE-bench has emerged as a widely used yardstick for evaluating LLMs on realistic software en-

Model	Go		Rust		Ruby	
	MS	MS@Pass	MS	MS@Pass	MS	MS@Pass
Small						
Yi-Coder (9B)	1.59	70.01	0.38	58.79	0.15	**98.44**
Qwen3 (8B)	0.39	**92.08**	0.00	0.00	0.00	0.00
Codestral (22B)	3.16	82.55	1.14	75.71	0.00	0.00
Medium						
GPT-OSS (120B)	15.04	78.63	2.72	79.00	1.25	84.44
Llama-3.3 (70B)	2.35	87.29	1.21	66.05	0.00	0.00
Qwen3-Coder-Next (80B)	4.73	74.26	0.98	50.69	0.29	66.31
Large						
DeepSeek V3.2	3.86	77.93	1.04	**87.70**	0.14	93.59
MiniMax-M2.7	3.82	81.64	0.74	57.22	0.01	9.68
Flagship						
DeepSeek V4-pro	11.42	83.96	2.75	73.22	1.67	94.13
GLM-5	3.62	79.91	1.22	60.00	0.43	71.81
Kimi-k2.5	1.34	63.07	0.49	76.19	0.01	6.25
Claude 4.5 Haiku	9.15	85.01	1.72	51.52	1.28	78.68
Claude 4.5 Sonnet	18.87	82.24	3.26	61.88	0.55	93.25
GPT-5.2	**19.62**	76.96	**4.14**	75.62	**6.29**	78.68

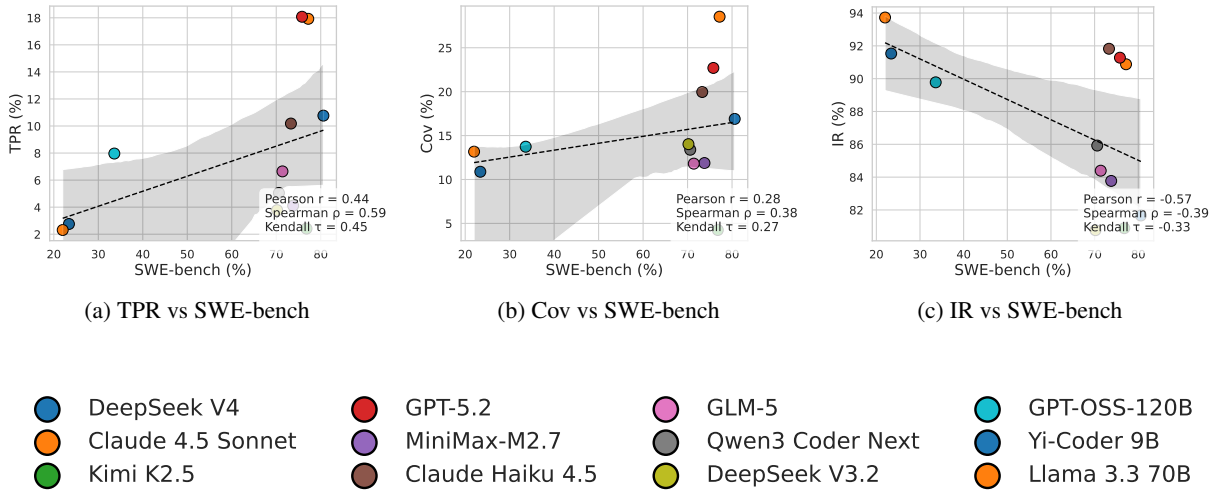
Table A.5: Mutation-testing results (MS and MS@Pass, %) across Go, Rust, and Ruby for all 14 evaluated models under `standard` context. Ruby MS values should be interpreted as lower-bound estimates due to tooling limitations with dynamic dispatch and metaprogramming-heavy code. Julia and PHP are excluded due to incompatible mutation libraries and file-level-only tooling granularity, respectively. Best scores are bold

gineering tasks, where a model must produce code changes that resolve real repository issues under test-based verification. We use it here as an external proxy for general software engineering capability.

We analyze correlations across 12 models with verified SWE-bench scores (ranging from 22.0% for Llama 3.3 70B to 80.6% for DeepSeek V4 Pro). Figure A.3 reports the Pearson, Spearman, and Kendall correlation coefficients.

Two patterns emerge. First, TPR shows the strongest relationship with SWE-bench: moderate positive Pearson ($r = 0.44$), and notably higher Spearman ($\rho = 0.59$) and Kendall ($\tau = 0.46$) correlations. This indicates that while the linear relationship is moderate, the rank-order relationship is more pronounced—models that perform better on SWE-bench tend to rank higher on TPR, though the relationship is not strictly linear. Cov shows a weaker positive trend (Spearman $\rho = 0.38$). CSR shows essentially no linear relationship with SWE-bench (Pearson -0.12), though a slight positive rank-order tendency (Spearman 0.13).

Second, IR shows a notable negative trend with SWE-bench across all three methods (Pearson $r = -0.57$, Spearman $\rho = -0.39$, Kendall $\tau = -0.33$). We emphasize that these correlations are computed over only 12 models, which limits statistical power and the precision of the estimates; accordingly, we present this as an observational pattern rather than a conclusive finding. Subject to this caveat, the negative trend is suggestive of a structural misalignment: software engineering capability as measured by patch synthesis may not transfer to—and could even interfere with—the specific constraint of targeting a focal method. One plausible explanation is that models with higher general software engineering capability may over-complicate test harnesses or hallucinate complex setups that miss direct invocation, whereas models with lower capability may rely on simpler, more



(a) TPR vs SWE-bench (b) Cov vs SWE-bench (c) IR vs SWE-bench

Figure A.3: Correlation between unit test generation metrics on XREPOTEST and SWE-bench performance. Pearson, Spearman, and Kendall coefficients are reported in each subfigure.

direct invocation patterns. This divergence hints that "can fix a bug" (SWE-bench) may be a distinct capability from "can strictly invoke a specific function," though confirming this would require a larger set of models or controlled experiments.

E.5 SWE-bench Verified Score Sources

Notes on evaluation scaffolds. Most scores are reported using the **mini-SWE-agent** scaffold (Jimenez et al., 2024) or equivalent standardized agent loop. The Qwen3-Coder-Next and DeepSeek V3.2 scores are from the Qwen3-Coder-Next technical report (Cao et al., 2026), which evaluates across multiple scaffolds (SWE-Agent, MiniSWE-Agent, OpenHands); we report the SWE-Agent result for consistency.

For reproducibility, all scores should be treated as provider-reported or independently replicated using the official SWE-bench evaluation harness (`https://github.com/swe-bench/SWE-bench`).

E.6 Benchmark Hardness Analysis

To understand whether different models solve the same set of problems or exhibit unique strengths, we conduct a benchmark hardness analysis focusing on three representative flagship models: Claude Sonnet 4.5, GPT-OSS-120B, and GPT-5.2. We consider a sample as successfully solved if the generated test passes and achieves non-zero line coverage. This criterion ensures that the test not only compiles and executes without errors but also exercises the focal function meaningfully.

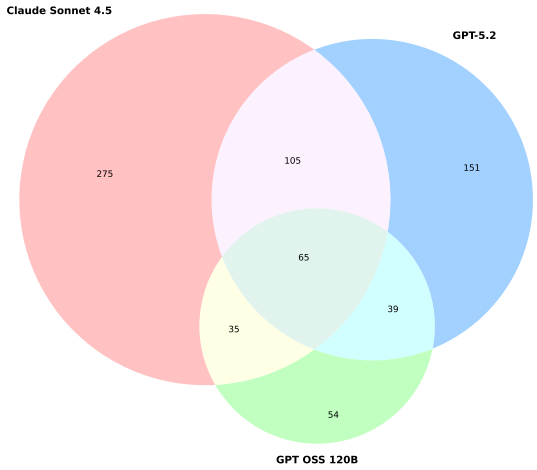


Figure A.4: Three-way Venn diagram showing overlap in successfully solved samples (test passes + coverage > 0) across three LLMs under `file_context` mode for all languages. Each region indicates the number of samples uniquely solved by specific model combinations, revealing both shared capabilities and model-specific strengths.

Figure A.4 presents a three-way Venn diagram illustrating the overlap in successfully solved samples across all five languages (Rust, Julia, Go, PHP, Ruby) under the `file_context` mode. The diagram reveals several key insights:

- **Shared Core.** A substantial intersection exists where all three models successfully solve the same problems, indicating a set of relatively easier samples that are accessible to state-of-the-art models regardless of their spe-

Retriever	w_s	k	Codestral	Yi-Coder
BM25	30	5	18.02	7.80
	30	10	17.84	7.41
	30	15	18.54	5.31
	50	5	17.64	8.41
	50	10	18.23	6.56
	50	15	18.59	5.31
	70	5	17.65	8.13
	70	10	18.30	7.28
	70	15	18.54	5.16
Dense	30	5	18.62	8.14
	30	10	16.68	8.30
	30	15	18.39	6.34
	50	5	17.71	8.97
	50	10	16.86	8.22
	50	15	17.66	5.80
	70	5	18.14	9.05
	70	10	16.04	8.49
	70	15	18.67	6.00

Table A.6: Retrieval parameter ablation: line coverage (%) across window size (ws) and top-k configurations for BM25 and Dense (UniXCoder) retrievers on Go.

cific architectures or training regimes.

- **Model-Specific Strengths.** Each model demonstrates unique capabilities, successfully solving problems that the other two fail on. These exclusive regions highlight architectural or training differences that enable specific models to handle certain code patterns, domains, or complexity characteristics more effectively.

- **Pairwise Complementarity.** The pairwise intersections (problems solved by exactly two models) suggest that model combinations could potentially achieve higher coverage than any single model, motivating ensemble or multi-model generation strategies for practical test generation systems.

This hardness analysis provides empirical evidence that benchmark difficulty is not uniform across models, different models exhibit complementary strengths, and a significant portion of the benchmark remains challenging for all evaluated systems, preserving discriminative power for future model development.

F Retrieval Parameter Ablation

To assess sensitivity to retrieval hyperparameters, we conducted an ablation study varying window size ($ws \in \{30, 50, 70\}$) and top-k ($k \in \{5, 10, 15\}$) for both BM25 and Dense (UniXCoder) retrievers. We report average line coverage across two representative models: Codestral and Yi-Coder, evaluated on Go.

Across all models, performance remains stable under moderate parameter variation. For Yi-Coder under BM25, coverage fluctuates between 5.16% and 8.41% across nine configurations ($\leq$3.3% absolute range). Codestral shows even narrower variation under BM25 (17.64%–18.59%, $\leq$1.0% absolute range). Similar patterns hold under dense retrieval.

The chosen configuration (ws=50, k=10), consistent with common practice in prior RAG-based code generation work (Zhang et al., 2023; Wu et al., 2024; Wang et al., 2024), consistently falls in the median-to-upper tier without being the peak for any single model. This avoids overfitting to a particular architecture while providing a representative and robust evaluation setting. These results confirm that retrieval performance is not highly sensitive to moderate parameter changes, supporting the validity of our chosen configuration.

Note that static augmentation contexts (file-level and LSP-based) are deterministic and invariant across runs; the ablation above applies only to retrieval-based augmentation.

G Statistical Significance Analysis

We conducted two complementary statistical analyses to validate the ranking stability of model performance across languages and context modes: bootstrap resampling (10,000 iterations) for confidence interval estimation and McNemar's chi-squared test for pairwise model comparisons. Across 25 language/mode combinations (5 modes $\times$ 5 language groups), we analyzed result files covering all 14 models.

G.1 Bootstrap Resampling

For each language/mode, we performed non-parametric bootstrap resampling to compute 95% confidence intervals for the top-3 models' pass rates. Table A.8 reports the mean pass rate and 95% CI for each. Models whose CI does not overlap with the runner-up are considered *statistically distinguishable*.

When TPR values are low (single-digit percentages for Rust, and below 10% for several other language/settings), small absolute differences may not be statistically reliable. We explicitly acknowledge that adjacent models with marginal TPR differences may be statistically indistinguishable. However, larger gaps between performance tiers remain robust, as demonstrated below.

Across all 25 settings, the top model's CI does not overlap with the runner-up in 10 cases (notably PHP and Julia across all five modes, and Go under standard mode). The coefficient of variation (CV) serves as a stability metric—higher CV indicates less stable estimates relative to the mean. Most models exhibit CV < 0.15, indicating reasonable stability; higher variance (CV > 0.20) is observed for weaker models in low-TPR settings (e.g., GPT-OSS 120B in PHP under RAG modes), where small absolute pass counts amplify bootstrap variance.

G.2 McNemar's Test Results

We applied McNemar's chi-squared test to all pairwise top-model comparisons within each language/mode setting. The test evaluates whether two models have statistically different pass/fail patterns, accounting for the paired nature of the data (same samples tested by both models). Table A.7 reports all significant comparisons across all 25 settings; all p-values are corrected using Holm-Bonferroni correction for multiple comparisons.

Of 90 top-model pairwise comparisons across all settings, 56 (62%) are statistically significant at $\alpha = 0.05$. The most consistent winner is **claude-sonnet-4-5**, achieving the highest rank in 21/25 settings and winning 32 significant pairwise comparisons. GPT-5.2 shows 19 significant wins, primarily in Go and Ruby. In Rust/standard, claude-sonnet-4-5 and gpt-5.2 are statistically indistinguishable (χ^2=0.22, p=0.64), though both significantly outperform claude-haiku-4.5.

Model Tier Separation (Rust File Context). In Rust under file-level context, the proprietary SOTA tier (Claude Sonnet 4.5: 21.16%, 95% CI [18.6%, 23.8%]) is statistically distinct from the leading open-weight baseline (GPT-OSS-120B: 8.06%, CI [6.3%, 9.9%]), with non-overlapping confidence intervals. Pairwise McNemar's test confirms this separation is highly significant (χ^2=89.63, p=2.87e-21 with Bonferroni correction).

Context Augmentation Significance. The improvement from Standard to File-Level Context for Claude Sonnet 4.5 (12.78% $\rightarrow$ 21.16%) is confirmed via McNemar's test (χ^2=17.65, p=1.53e-05), with 42 additional passing tests versus only 6 additional failures. This pattern holds across languages: PHP (26.95% $\rightarrow$ 31.43%) and Go (23.66% $\rightarrow$ 34.14%) show similarly significant gains, confirming that repository-aware context provides a decisive, non-random advantage.

G.3 Summary

The combined bootstrap and McNemar analyses confirm that **claude-sonnet-4-5** is the most consistent top performer (21/25 settings ranked first, 32 significant pairwise wins), with GPT-5.2 as the primary challenger in Go and Ruby. Overall, 56 of 90 pairwise comparisons (62%) reach significance, and the context augmentation effect is confirmed across all five languages.

Language/Mode	Model A	Model B	χ^2	p-value	Sig.
		Rust			
Rust / Standard	Claude 4.5 Sonnet	GPT-5.2	0.22	0.6396	ns
Rust / Standard	Claude 4.5 Sonnet	Claude 4.5 Haiku	11.56	0.0007	***
Rust / Standard	GPT-5.2	Claude 4.5 Haiku	8.09	0.0045	**
Rust / File Ctx	Claude 4.5 Sonnet	GPT-OSS 120B	89.63	2.87e-21	***
Rust / File Ctx	Claude 4.5 Sonnet	GPT-5.2	10.37	0.0013	**
Rust / LSP Ctx	Claude 4.5 Sonnet	GPT-OSS 120B	29.54	5.49e-08	***
Rust / RAG BM25	Claude 4.5 Sonnet	GPT-OSS 120B	64.61	9.13e-16	***
Rust / RAG Dense	Claude 4.5 Sonnet	GPT-OSS 120B	85.82	1.98e-20	***
Rust / RAG Dense	Claude 4.5 Sonnet	GPT-5.2	9.23	0.0024	**
		Go			
Go / Standard	GPT-5.2	GPT-OSS 120B	11.03	0.0009	***
Go / Standard	Claude 4.5 Sonnet	GPT-OSS 120B	6.32	0.0119	*
Go / File Ctx	GPT-5.2	GPT-OSS 120B	31.84	1.67e-08	***
Go / File Ctx	Claude 4.5 Sonnet	GPT-OSS 120B	18.46	1.73e-05	***
Go / LSP Ctx	GPT-5.2	GPT-OSS 120B	26.46	2.69e-07	***
Go / LSP Ctx	Claude 4.5 Sonnet	GPT-OSS 120B	15.96	6.47e-05	***
Go / RAG BM25	GPT-5.2	GPT-OSS 120B	21.97	2.78e-06	***
Go / RAG BM25	Claude 4.5 Sonnet	GPT-OSS 120B	18.89	1.39e-05	***
Go / RAG Dense	GPT-5.2	GPT-OSS 120B	44.76	2.22e-11	***
Go / RAG Dense	Claude 4.5 Sonnet	GPT-OSS 120B	31.21	2.32e-08	***
		Julia			
Julia / Standard	Claude 4.5 Sonnet	Claude 4.5 Haiku	18.39	1.80e-05	***
Julia / File Ctx	Claude 4.5 Sonnet	GPT-OSS 120B	40.29	2.19e-10	***
Julia / File Ctx	Claude 4.5 Sonnet	GPT-5.2	7.56	0.0060	**
Julia / LSP Ctx	Claude 4.5 Sonnet	GPT-OSS 120B	31.44	2.06e-08	***
Julia / RAG BM25	Claude 4.5 Sonnet	GPT-OSS 120B	50.27	1.34e-12	***
Julia / RAG BM25	Claude 4.5 Sonnet	GPT-5.2	14.11	1.72e-04	***
Julia / RAG Dense	Claude 4.5 Sonnet	GPT-OSS 120B	15.74	7.27e-05	***
Julia / RAG Dense	Claude 4.5 Sonnet	GPT-5.2	14.08	1.75e-04	***
		PHP			
PHP / Standard	Claude 4.5 Sonnet	Claude 4.5 Haiku	38.50	5.48e-10	***
PHP / Standard	Claude 4.5 Sonnet	GLM-5	45.29	1.70e-11	***
PHP / File Ctx	Claude 4.5 Sonnet	GPT-OSS 120B	130.91	2.59e-30	***
PHP / File Ctx	Claude 4.5 Sonnet	GPT-5.2	15.59	7.85e-05	***
PHP / LSP Ctx	Claude 4.5 Sonnet	GPT-OSS 120B	113.61	1.59e-26	***
PHP / LSP Ctx	Claude 4.5 Sonnet	GPT-5.2	50.92	9.63e-13	***
PHP / RAG BM25	Claude 4.5 Sonnet	GPT-OSS 120B	152.30	5.46e-35	***
PHP / RAG BM25	Claude 4.5 Sonnet	GPT-5.2	66.65	3.24e-16	***
PHP / RAG Dense	Claude 4.5 Sonnet	GPT-OSS 120B	149.74	1.98e-34	***
PHP / RAG Dense	Claude 4.5 Sonnet	GPT-5.2	52.48	4.34e-13	***
		Ruby			
Ruby / Standard	GPT-5.2	DeepSeek V4-pro	76.96	1.75e-18	***
Ruby / Standard	GPT-5.2	GPT-OSS 120B	95.07	1.84e-22	***
Ruby / File Ctx	Claude 4.5 Sonnet	GPT-OSS 120B	10.59	0.0011	**
Ruby / LSP Ctx	GPT-5.2	Claude 4.5 Sonnet	68.21	1.47e-16	***
Ruby / LSP Ctx	GPT-5.2	GPT-OSS 120B	80.13	3.50e-19	***
Ruby / RAG BM25	GPT-5.2	Claude 4.5 Sonnet	29.18	6.58e-08	***
Ruby / RAG BM25	GPT-5.2	GPT-OSS 120B	13.78	2.06e-04	***
Ruby / RAG Dense	GPT-5.2	Claude 4.5 Sonnet	29.90	4.55e-08	***
Ruby / RAG Dense	GPT-5.2	GPT-OSS 120B	13.34	2.60e-04	***

Table A.7: McNemar's chi-squared test results for all significant pairwise model comparisons across all language/-mode combinations. Holm-Bonferroni corrected p-values. Significance levels: *** $p < 0.001$, ** $p < 0.01$, * $p < 0.05$, ns = not significant.

	Go			Julia			PHP			Ruby			Rust		
	TPR	**CI**	**CV**	**TPR**	**CI**	**CV**	**TPR**	**CI**	**CV**	**TPR**	**CI**	**CV**	**TPR**	**CI**	**CV**
	Standard														
GPT-5.2	25.78	[22.7, 29.0]	0.064	16.52	[13.7, 19.4]	0.087	10.06	[7.9, 12.5]	0.118	25.78	[22.5, 29.0]	0.065	12.25	[10.2, 14.4]	0.088
Claude 4.5 Sonnet	23.66	[20.5, 26.8]	0.067	19.85	[16.8, 22.8]	0.076	26.95	[23.5, 30.3]	0.064	6.37	[4.6, 8.3]	0.148	12.78	[10.6, 15.0]	0.086
GPT-OSS 120B	19.27	[16.4, 22.2]	0.078	4.39	[3.0, 5.9]	0.152	3.41	[2.0, 4.8]	0.208	6.97	[5.2, 8.9]	0.140	5.80	[4.3, 7.5]	0.129
	File Context														
GPT-5.2	37.69	[34.0, 41.4]	0.049	17.28	[14.5, 20.2]	0.084	22.93	[19.7, 26.2]	0.072	13.02	[10.5, 15.6]	0.099	17.94	[15.5, 20.4]	0.070
Claude 4.5 Sonnet	34.14	[30.7, 37.7]	0.052	23.98	[20.8, 27.2]	0.068	31.43	[27.9, 35.0]	0.058	16.26	[13.5, 19.1]	0.088	21.16	[18.6, 23.8]	0.063
GPT-OSS 120B	25.93	[22.8, 29.2]	0.064	9.49	[7.3, 11.8]	0.119	6.34	[4.5, 8.2]	0.151	13.71	[10.3, 17.4]	0.135	8.06	[6.3, 9.9]	0.111
	LSP Context														
GPT-5.2	28.03	[24.8, 31.4]	0.061	12.84	[10.4, 15.4]	0.099	11.76	[9.3, 14.2]	0.108	23.84	[20.6, 27.0]	0.068	11.50	[9.5, 13.5]	0.091
Claude 4.5 Sonnet	25.34	[22.2, 28.6]	0.064	15.64	[13.0, 18.4]	0.088	26.77	[23.4, 30.2]	0.066	7.69	[5.8, 9.8]	0.134	13.55	[11.4, 15.8]	0.083
GPT-OSS 120B	17.41	[14.6, 20.3]	0.083	5.84	[4.1, 7.6]	0.154	4.48	[2.9, 6.2]	0.183	7.55	[5.6, 9.6]	0.136	6.45	[4.9, 8.1]	0.124
	RAG BM25														
GPT-5.2	28.89	[25.6, 32.2]	0.059	11.42	[9.2, 13.9]	0.106	13.17	[10.7, 15.8]	0.101	19.84	[16.9, 23.0]	0.078	15.37	[13.1, 17.7]	0.077
Claude 4.5 Sonnet	28.05	[24.8, 31.4]	0.060	19.16	[16.2, 22.1]	0.078	30.02	[26.5, 33.6]	0.061	10.38	[8.1, 12.7]	0.111	15.98	[13.6, 18.4]	0.075
GPT-OSS 120B	18.98	[16.1, 22.0]	0.078	6.42	[4.7, 8.3]	0.146	3.41	[2.0, 4.8]	0.208	17.14	[13.1, 21.1]	0.119	4.93	[3.5, 6.3]	0.144
	RAG Dense														
GPT-5.2	32.13	[28.8, 35.6]	0.055	11.68	[9.2, 14.2]	0.106	16.12	[13.3, 19.0]	0.090	21.06	[18.1, 24.1]	0.075	13.85	[11.7, 16.1]	0.082
Claude 4.5 Sonnet	30.32	[26.9, 33.7]	0.057	18.57	[15.8, 21.5]	0.079	31.43	[27.9, 35.1]	0.059	11.27	[8.9, 13.8]	0.109	17.18	[14.8, 19.5]	0.071
GPT-OSS 120B	17.70	[14.9, 20.5]	0.081	10.08	[7.9, 12.4]	0.114	4.95	[3.3, 6.7]	0.173	18.56	[14.6, 22.6]	0.112	4.52	[3.2, 5.9]	0.150

Table A.8: Bootstrap 95% CI for top models across all language/mode combinations (5 languages $\times$ 5 modes). TPR values are percentages. Within each mode block, only models appearing in the top-3 for at least 3 out of 5 languages are shown (Standard block); other blocks show the top-3 models consistent across all languages. CV = coefficient of variation.

H Model Details

In our experimental study, we evaluate a diverse set of large language models spanning both proprietary and open-source families. Specifically, we consider proprietary models from OpenAI and Anthropic, including GPT-5.2 and GPT-OSS (Agarwal et al., 2025), as well as Claude 4.5.

Our evaluation further covers a broad range of open-source and open-weight models, including GLM-5 (GLM-5-Team et al., 2026), DeepSeek V3.2 (DeepSeek-AI et al., 2025) and DeepSeek V4-pro, Kimi-k2.5 (Team et al., 2026), MiniMax-M2.7, and several models from Alibaba's Qwen family, including Qwen3 and Qwen3-Coder-Next (Cao et al., 2026). We also evaluate Yi, Codestral, and Llama-3.3. Complete model identifiers and release details are provided in Table A.9

I Failure Mode Details

I.1 Category Definitions

We use the following definitions for the five failure classes reported in Section 5.3.

1. Syntactic & Compilation Tests that fail at compile or interpretation time due to syntax errors, missing imports, or type declaration issues. This is especially prominent in Go, reflecting challenges with package visibility and interface satisfaction. Examples include missing semicolons (PHP), incorrect closure syntax (Rust), block precedence issues (Ruby), and using Python keywords in Julia code ("def" instead of "function").

2. Type System & Memory Failures due to type mismatches, ownership violations, or memory safety issues. This is most visible in Rust due to its strict ownership model (borrow checker violations, lifetime errors), and less prominent in dynamic languages. In Go, this includes interface conversion panics and missing type assertion checks. In PHP and Ruby, this manifests as strict type mismatches and `NoMethodError` on `nil`.

3. API Hallucination Cases where models invent non-existent libraries, methods, or APIs. This is the dominant failure mode in several under-represented languages, indicating that models frequently generate plausible-looking but fictitious API calls. Examples include fabricating `testify` assertions in Go (e.g., `assert.NotError`), inventing trait methods in Rust, calling `len()` instead of `length()` in Julia, and using deprecated PHPUnit annotations.

4. Logic & Assertion Tests that compile and run but fail due to incorrect logic or assertions. This is especially prominent in PHP, where syntactic correctness is comparatively easier to achieve but semantic understanding of expected behavior remains challenging. Examples include unwrapping `Result` expecting failure (Rust), checking `err != nil` without calling `t.Fatal` (Go), using `==` for floating point comparisons instead of `isapprox` (Julia), and confused argument order in PHPUnit assertions.

5. Test Design & Mocking Failures from missing fixtures, incorrect mocking, or improper test setup, reflecting the complexity of framework conventions and dependency injection patterns. The primary indicator is tests that pass all checks but achieve zero focal coverage (`covered_lines == 0`), suggesting the focal function was never invoked. Examples include mocking the class under test instead of external dependencies, hardcoded database IDs without data setup, and global state pollution without cleanup.

I.2 Error Classification Methodology

To systematically analyze failure modes in LLM-generated tests, we developed an automated error classification system that processes test execution logs, compilation diagnostics, and coverage statistics across all five languages. The classifier employs a rule-based approach with language-specific pattern matching and a hierarchical decision procedure that prioritizes error categories based on the test lifecycle stage.

Hierarchical Classification Decision Tree. Classification follows a priority-ordered decision tree that mirrors the test execution lifecycle. Importantly, the decision logic differs between **static languages (Rust, Go)** and **dynamic languages (Julia, PHP, Ruby)**, reflecting fundamental differences in when type checking and API resolution occur:

1. **Preprocessing Failures:** If test code is empty or logs indicate "no tests generated", classify as **Category 1** (Syntactic & Compilation), signaling that the LLM output was malformed enough to be filtered during test extraction.

2. **Compilation Failures:** When `check.compilation == False`, the clas-

Provider	Model ID	Short Name	Link
OpenAI	gpt-5.2	GPT-5.2	https://openai.com/index/introducing-gpt-5-2/
	openai-gpt-oss-120b	GPT-OSS 120B	https://huggingface.co/openai/gpt-oss-120b
Anthropic	claude-haiku-4-5-20251001	Claude 4.5 Haiku	https://www.anthropic.com/news/claude-haiku-4-5
	claude-sonnet-4-5-20250929	Claude 4.5 Sonnet	https://www.anthropic.com/news/claude-sonnet-4-5
Zhipu AI	zai-org/GLM-5	GLM-5	https://huggingface.co/zai-org/GLM-5
DeepSeek	deepseek-ai/DeepSeek-V3.2	DeepSeek V3.2	https://huggingface.co/deepseek-ai/DeepSeek-V3.2
	deepseek-ai/DeepSeek-V4-pro	DeepSeek V4-pro	https://huggingface.co/deepseek-ai/DeepSeek-V4-pro
Moonshot AI	moonshotai/Kimi-k2.5-Instruct	Kimi-k2.5	https://huggingface.co/moonshotai/Kimi-K2.5
MiniMax	MiniMaxAI/MiniMax-M2.7	MiniMax-M2.7	https://huggingface.co/MiniMaxAI/MiniMax-M2.7
Alibaba	Qwen/Qwen3-8B	Qwen3 8B	https://huggingface.co/Qwen/Qwen3-8B
	Qwen/Qwen3-Coder-Next	Qwen3-Coder-Next 80B	https://huggingface.co/Qwen/Qwen3-Coder-Next
01.AI	01-ai/Yi-Coder-9B-Chat	Yi-Coder 9B	https://huggingface.co/01-ai/Yi-Coder-9B-Chat
Mistral AI	mistralai/Codestral-22B-v0.1	Codestral 22B	https://huggingface.co/mistralai/Codestral-22B-v0.1
Meta	meta-llama/Llama-3.3-70B-Instruct	Llama-3.3 70B	https://huggingface.co/meta-llama/Llama-3.3-70B-Instruct

Table A.9: Details of the LLMs evaluated in our experimental study.

sification strategy diverges by language type:

For static languages (Rust, Go), where compilation includes type checking and API resolution, we sequentially check:

- **Category 3** (API Hallucination) – highest priority, as undefined symbols indicate fabricated APIs
- **Category 2** (Type System & Memory) – type errors and ownership violations
- **Category 1** (Syntactic & Compilation) – fallback for parse errors and malformed syntax

For dynamic languages (Julia, PHP, Ruby), where compilation only validates syntax (not types or API existence), we **directly classify as Category 1**:

- **Category 1** (Syntactic & Compilation) – parse errors and malformed syntax
- Type checking and API resolution are deferred to runtime; Categories 2 and 3 are not checked during compilation failures

3. **Runtime Failures:** When compilation succeeds but `check.tests == False`, we again distinguish by language type:

   For static languages (Rust, Go):

   - **Category 2** (Type System & Memory, runtime) – runtime type errors and panics
   - **Category 4** (Logic & Assertion) – assertion failures and incorrect test logic

   For dynamic languages (Julia, PHP, Ruby), where type and API errors manifest at runtime:

   - **Category 3** (API Hallucination) – undefined functions, methods, or constants
   - **Category 2** (Type System & Memory, runtime) – runtime type errors, argument mismatches
   - **Category 4** (Logic & Assertion) – assertion failures and incorrect test logic

4. **Design Issues:** When all checks pass (`check.compilation == True` and `check.tests == True`), we apply coverage-based heuristics:

Table A.10: Failure modes by language with and without context (per-model). Values are percentages of total samples assigned to each failure mode; rows sum to 100% within each column, with the "No Error / Pass" row capturing samples that pass all checks.

Model	Failure Mode	Rust		Go		Julia		Ruby		PHP	
		Std.	Ctx.	Std.	Ctx.	Std.	Ctx.	Std.	Ctx.	Std.	Ctx.
Claude 4.5 Sonnet	Syntactic & Compilation	1.29%	2.36%	14.02%	19.12%	1.75%	1.61%	0.44%	0.89%	1.24%	1.86%
	Type System & Memory	14.18%	14.93%	13.60%	8.36%	9.36%	8.04%	5.93%	2.67%	6.66%	3.72%
	API Hallucination	43.07%	30.08%	21.53%	11.61%	59.65%	53.95%	49.78%	32.74%	24.46%	29.72%
	Logic & Assertion	28.68%	31.47%	27.20%	26.77%	9.36%	12.43%	37.48%	47.41%	40.71%	33.28%
	Test Design & Mocking	4.40%	6.44%	0.00%	0.42%	0.00%	0.00%	0.74%	0.74%	0.31%	0.46%
	No Error / Pass	8.38%	14.72%	23.65%	33.72%	19.88%	23.97%	5.63%	15.55%	26.62%	30.96%
GPT-5.2	Syntactic & Compilation	0.97%	1.50%	19.69%	14.59%	15.79%	8.04%	1.78%	7.41%	2.32%	3.72%
	Type System & Memory	11.71%	13.96%	13.88%	8.50%	6.87%	7.02%	4.74%	2.52%	5.42%	4.18%
	API Hallucination	48.55%	33.62%	19.12%	12.32%	37.43%	46.93%	23.41%	38.81%	47.06%	38.08%
	Logic & Assertion	26.53%	32.98%	21.53%	26.91%	23.39%	20.76%	44.30%	38.07%	35.14%	31.11%
	Test Design & Mocking	3.87%	5.59%	0.71%	1.42%	0.00%	0.00%	9.19%	0.59%	2.48%	0.77%
	No Error / Pass	8.38%	12.35%	25.07%	36.26%	16.52%	17.25%	16.58%	12.60%	7.58%	22.14%
GPT-OSS (120B)	Syntactic & Compilation	9.56%	3.87%	41.22%	27.34%	29.09%	16.52%	2.81%	2.86%	48.14%	30.65%
	Type System & Memory	8.06%	15.57%	12.32%	8.50%	10.82%	9.94%	3.26%	1.14%	2.32%	0.93%
	API Hallucination	52.52%	44.58%	16.01%	19.12%	46.64%	49.42%	53.33%	65.43%	35.14%	54.02%
	Logic & Assertion	24.06%	27.93%	11.19%	19.12%	9.06%	14.62%	33.48%	16.57%	10.99%	8.05%
	Test Design & Mocking	1.61%	2.90%	0.42%	1.13%	0.00%	0.00%	3.26%	0.29%	0.46%	0.77%
	No Error / Pass	4.19%	5.16%	18.84%	24.79%	4.39%	9.50%	3.86%	13.71%	2.95%	5.58%

- If `covered_lines == 0` (or ≤ 1 for Ruby), classify as **Category 5** (Test Design & Mocking), indicating the test executes but does not invoke the focal function.
- Otherwise, no error is reported (`has_error = False`).

I.3 API Hallucination Sub-category Analysis

To provide more actionable diagnostic insights, we decomposed all API Hallucination failures into four sub-categories:

1. **Phantom Library** — importing non-existent packages;
2. **Non-existent API** — calling fictitious functions within valid packages;
3. **Signature Mismatch** — incorrect arguments or types for real APIs;
4. **Other** — primarily deprecated APIs.

Table A.11 reports the distribution of these subtypes within the API Hallucination category, comparing standard (**Std**) and context-augmented (**Ctx**) settings. Values represent the percentage share within the hallucination category (columns sum to 100%).

Key findings. Three patterns emerge from this finer-grained analysis.

Context redistributes rather than uniformly reduces hallucination. Across languages, context augmentation alters the *composition* of hallucination rather than simply decreasing it. In Rust and PHP, Phantom Library errors remain dominant under context (Rust: 77.9% $\rightarrow$ 72.9%; PHP: 88.8% $\rightarrow$ 92.4%), suggesting that broader repository exposure can amplify incorrect assumptions about dependency structure. In contrast, Go exhibits a sharp shift: Phantom Library errors drop dramatically (23.2% $\rightarrow$ 3.9%) while Non-existent API errors increase markedly (71.2% $\rightarrow$ 85.9%). Here, context successfully anchors correct package imports, but models overgeneralize by inventing plausible methods on valid types.

Hallucination is primarily an ecosystem-grounding issue. Signature Mismatch remains secondary across most languages (<10%, except Julia at 21–33%), indicating that failures stem less from type-level reasoning and more from incorrect assumptions about available libraries and APIs. Techniques that constrain generation using repository-aware metadata (dependency graphs, import trees, module registries) may therefore be more impactful than improvements in generic type reasoning alone.

Language-dependent structural mechanisms. Ruby and PHP show extremely high Phantom Library dominance under both settings (>76%), con-

Sub-category	Rust		Go		Julia		Ruby		PHP	
	Std	Ctx	Std	Ctx	Std	Ctx	Std	Ctx	Std	Ctx
Phantom Library	77.9	72.9	23.2	3.9	32.8	38.8	76.9	77.5	88.8	92.4
Non-existent API	16.8	19.0	71.2	85.9	34.4	39.2	19.3	18.5	10.0	6.7
Sig. Mismatch	4.5	7.0	5.5	10.2	32.8	21.9	3.2	2.8	0.9	0.1
Other	0.7	1.0	0.0	0.0	0.1	0.1	0.6	1.1	0.3	0.8

Table A.11: Breakdown of API Hallucination failures into sub-categories (%). Values represent the distribution within the hallucination category; columns sum to 100%.

sistent with dynamic or framework-heavy ecosystems that encourage speculative dependency invention. Julia exhibits a more balanced distribution across sub-types, with Signature Mismatch notably higher (21–33%) than other languages, reflecting its multiple-dispatch system. Go's strong shift from package-level to method-level hallucination reflects its explicit import system, which reduces dependency invention but does not prevent API overgeneralization. These patterns suggest that evaluation strategies and mitigation techniques should be language-aware: dependency validation may be critical for dynamic ecosystems, while API-level verification may yield larger gains for statically structured languages.

I.4 Failure Mode Breakdown by Model

Table A.10 presents a comprehensive breakdown of failure modes across all evaluated models and languages, comparing standard and context-enriched prompting strategies. All values are normalized by the total number of benchmark samples (not only the error subset), avoiding the misleading scenario where a reduction in error count paradoxically increases a category's share. Each column sums to 100%, with the *No Error / Pass* row capturing samples that pass all checks without any failure. The table shows the distribution of samples across five primary failure mode categories plus the no-error class: Syntactic & Compilation errors, Type System & Memory issues, API Hallucination, Logic & Assertion failures, and Test Design & Mocking problems. This granular view reveals model-specific strengths and weaknesses, and demonstrates how context enrichment affects the distribution of failure modes across different languages.

I.5 Illustrative Examples

We provide one concrete example per failure category. Each example is taken from Go under `standard` mode with GPT-5.2, using the generated test code and execution logs produced by our evaluation pipeline.

Category 1: Syntactic & Compilation

Focal function: `Encrypt` in `Go-master/cipher/transposition/transposition.go`. The generated test attempted to monkey-patch the package-level helper function `getKey`, but ordinary Go functions are not assignable.

Focal function (excerpt):

```go
func Encrypt(text []rune, keyWord string)
    ([]rune, error) {
    key := getKey(keyWord)

    if len(key) <= 0 {
        return nil, ErrKeyMissing
    }

    ...
    return result, nil
}
```

Generated test (excerpt):

```go
origGetKey := getKey

for _, tt := range tests {
    t.Run(tt.name, func(t *testing.T) {
        getKey = origGetKey

        if tt.mockGetKey != nil {
            getKey = tt.mockGetKey
        }

        _, err := Encrypt(
            append([]rune(nil), tt.text...),
     tt.keyWord)
        ...
    })
}
```

Key log excerpt:

```
./temp_test.go:131:4: cannot assign to
    getKey
./temp_test.go:133:5: cannot assign to
    getKey
./temp_test.go:171:2: cannot assign to
    getKey
```

Category 2: Type System & Memory

Failure summary. The generated test used characters from `ComputeAlphabet` as keys into the transition map. However, the implementation uses `map[int]map[uint8]int`, while the loop variable `ch` is inferred as `rune`, causing a static type error.

Focal function. `BuildExtendedAc` in `Go-master/strings/ahocorasick/advancedahocorasick.go`.

Focal code excerpt.

```go
func BuildExtendedAc(p []string) (
    acToReturn map[int]map[uint8]int,
    f map[int][]int,
) {
    acTrie, _, f := ConstructTrie(p)
    acToReturn = acTrie

    a := ComputeAlphabet(p)
    for j := range a {
        if GetTransition(0, a[j], acToReturn
    ) == -1 {
            CreateTransition(0, a[j], 0,
    acToReturn)
        }
    }

    ...
    return acToReturn, f
}
```

Generated test excerpt.

```go
func
    TestBuildExtendedAc_NoMissingTransitions
    (t *testing.T) {
    patterns := []string{"he", "she", "his",
     "hers"}
    ac, _ := BuildExtendedAc(patterns)
    alphabet := ComputeAlphabet(patterns)

    for state, trans := range ac {
        for _, ch := range alphabet {
            next, ok := trans[ch]
            if !ok {
                t.Errorf("missing transition
     from state %d on %q",
                    state, ch)
            }
            ...
        }
    }
}
```

Key log excerpt.

```
./temp_test.go:277:22: cannot use ch
    (variable of type rune) as uint8 value
    in map index
```

Category 3: API Hallucination

Failure summary. The generated test assumed that the returned `Result` type exposes an exported field named `Occurrences`. However, the actual implementation stores the match positions in an unexported field, `occurrences`. As a result, the test references a non-existent public API and fails to compile.

Focal function. `AhoCorasick` in `Go-master/strings/ahocorasick/ahocorasick.go`.

Focal code excerpt.

```
func AhoCorasick(t string, p []string)
    Result {
    occurrences := make(map[int][]int)

    ...
    var resultOccurrences = make(map[string
    ][]int)
    for key, value := range occurrences {
        resultOccurrences[p[key]] = value
    }

    return Result{
        resultOccurrences,
    }
}
```

Generated test excerpt.

```
// Check occurrences.
if res.Occurrences == nil {
    t.Fatalf("Result.Occurrences is nil;
     expected non-nil map")
}

if len(res.Occurrences) != len(tt.want) {
    t.Errorf("unexpected number of patterns:
      got %d, want %d",
        len(res.Occurrences), len(tt.want))
}

for pat, wantPos := range tt.want {
    gotPos, ok := res.Occurrences[pat]
    ...
}
```

Key log excerpt.

```
./temp_test.go:176:11: res.Occurrences
    undefined
    (type Result has no field or method
    Occurrences,
    but does have occurrences)
```

Category 4: Logic & Assertion

Failure summary. The generated test over-specified behavior that the implementation does not guarantee. It expected one exact DFS route, although different valid traversal orders may occur. It also assumed that a missing start node should be handled gracefully, but the implementation can panic when `GetIdx` returns `-1`.

Focal function. `DepthFirstSearchHelper` in `Go-master/graph/depthfirstsearch.go`.

Focal code excerpt.

```
startIdx := GetIdx(start, nodes)
stack = append(stack, startIdx)

for len(stack) > 0 {
    now := stack[len(stack)-1]
    route = append(route, nodes[now])

    for i := 0; i < len(edges[now]); i++ {
        if edges[now][i] && NotExist(i,
     stack) {
            stack = append(stack, i)
        }
    }

    if route[len(route)-1] == end {
        return route, true
    }
}
```

Generated test excerpt.

```
// Invalid start node is assumed to return
    no path.
route, found := DepthFirstSearchHelper(
    99, 3, nodes, edges, true)

if found {
    t.Fatalf("expected no path, got route %v
     ", route)
}
```

Key log excerpt.

```
branching_graph_path_found:
    route = [1 3 2 4]; want [1 2 4]

cycle_in_graph_still_finds_path:
    route = [1 3]; want [1 2 3]

start_not_in_nodes:
    panic: runtime error: index out of range
      [-1]
```

Category 5: Test Design & Mocking

Focal function: `Forward` in `ollama-main/model/model.go`. The generated tests passed, but achieved zero focal coverage (`covered_lines=0` out of `total_lines=22`) because they tested a local wrapper instead of calling the real focal function.

Focal function (excerpt):

```go
func Forward(ctx ml.Context, m Model,
    inputs []int32, batch input.Batch) (ml.
    Tensor, error) {

    if len(batch.Positions) != len(batch.
    Sequences) {
        return nil, fmt.Errorf("length
    mismatch")
    }
    if len(batch.Positions) < 1 {
        return nil, errors.New("batch size
    cannot be less than 1")
    }

    ...
    t, err := m.Forward(ctx, batch)
    if err != nil {
        return nil, err
    }

    ctx.Forward(t).Compute(t)
    return t, nil
}
```

Generated test excerpt.

```go
// Local wrapper with the same body as the
    real Forward.
func forwardUnderTest(ctx testContext, m
    testModel,
    inputs []int32, batch *fakeBatch) (*
    fakeTensor, error) {

    if len(batch.Positions) != len(batch.
    Sequences) {
        return nil, fmt.Errorf("length
    mismatch")
    }
    if len(batch.Positions) < 1 {
        return nil, errors.New("batch size
    cannot be less than 1")
    }

    ...
}
```

Key log excerpt.

```
--- PASS: TestForward (0.00s)
PASS
ok  github.com/ollama/ollama/model  0.005s

covered_lines=0 / total_lines=22
```

J Prompt Template

We use a single unified prompting scheme across all evaluated models. The prompt used by our pipeline consists of (i) a language-specific system prompt and (ii) a shared user prompt template with optional language-specific requirements.

For reproducibility, we include the exact prompt definitions below:

Rust

Rust: System Prompt

```
You are an expert Rust developer
    specializing in test-driven development
    (TDD) and writing high-quality unit
    tests.

Generate unit tests for Rust code using the
    following format:
- Use `#[cfg(test)] mod tests` with `#[test
    ]` functions
- Use `assert_eq!`, `assert!`, and `#[
    should_panic]` for assertions
- Cover normal cases, edge cases, and error
    cases

Always wrap your test code in triple
    backticks with the language identifier:
```rust
// Your generated test code here
```

Here's an example of the expected format (
    adapt this structure to the actual code
    you're testing):
```rust
#[cfg(test)]
mod tests {
    use super::*;

    #[test]
    fn test_add() {
        assert_eq!(add(2, 3), 5);
    }

    #[test]
    #[should_panic]
    fn test_divide_by_zero() {
        divide(10, 0);
    }
}
```

Generate comprehensive tests tailored to the
     specific code provided.
```

Rust: User Prompt Template

```
Generate unit tests for this Rust function
    from {file_path}:

```rust
{function_code}
```

Additional requirements:
- Import necessary items with `use super
    ::*;`
- Use `assert_ne!` for inequality checks
    where appropriate
- Include doc comments explaining what each
    test validates
```

Go

Go: System Prompt

```
You are an expert Go developer specializing
    in test-driven development (TDD) and
    writing high-quality unit tests.

Generate unit tests for Go code using the
    following format:
- Create tests in `_test.go` files
- Use table-driven tests with `t.Run()` for
    multiple test cases
- Use `t.Errorf()` for test failures
- Cover normal cases, edge cases, and error
    cases

Always wrap your test code in triple
    backticks with the language identifier:
```go
// Your generated test code here
```

Here's an example of the expected format (
    adapt this structure to the actual code
    you're testing):
```go
package main

import "testing"

func TestAdd(t *testing.T) {
    tests := []struct {
        name     string
        a, b     int
        expected int
    }{
        {"positive numbers", 2, 3, 5},
        {"with zero", 0, 5, 5},
        {"negative numbers", -2, -3, -5},
    }

    for _, tt := range tests {
        t.Run(tt.name, func(t *testing.T) {
            result := Add(tt.a, tt.b)
            if result != tt.expected {
                t.Errorf("Add(%d, %d) = %d;
    want %d", tt.a, tt.b, result, tt.
    expected)
            }
        })
    }
}
```

Generate comprehensive tests tailored to the
     specific code provided.
```

Go: User Prompt Template

```
Generate unit tests for this Go function
    from {file_path}:

```go
{function_code}
```

Additional requirements:
- Package name: {package_name}
- Test function name: `func Test{
    function_name}(t *testing.T)`
- Use `t.Fatalf()` for critical failures
    that should stop the test
- For error cases, check both error
    occurrence and error message
- Import necessary packages beyond `testing`
     if needed
```

Julia

Julia: System Prompt

```
You are an expert Julia developer
    specializing in test-driven development
    (TDD) and writing high-quality unit
    tests.

Generate unit tests for Julia code using the
     following format:
- Create tests in separate test files
- Use `@testset` to group related tests and
    `@test` for assertions
- Remember: Julia uses 1-based indexing (
    arrays start at index 1)
- Cover normal cases, edge cases, and error
    cases

Always wrap your test code in triple
    backticks with the language identifier:
```julia
// Your generated test code here
```

Here's an example of the expected format (
    adapt this structure to the actual code
    you're testing):
```julia
using Test

@testset "Array operations" begin
    @testset "sum_array tests" begin
        @test sum_array([1, 2, 3]) == 6
        @test sum_array([]) == 0
        @test sum_array([-1, 1]) == 0
    end

    @testset "first_element tests" begin
        arr = [10, 20, 30]
        @test first_element(arr) == 10  #
    Remember: 1-based indexing!
        @test_throws BoundsError
    first_element([])
    end
end
```

Generate comprehensive tests tailored to the
     specific code provided, always
    remembering Julia's 1-based indexing!
```

Julia: User Prompt Template

```
Generate unit tests for this Julia function
    from {file_path}:

```julia
{function_code}
```

Additional requirements:
- Include descriptive names for testsets
- Test boundary conditions carefully
- Remember: `length(arr)` gives array size,
    `arr[1]` is first element, `arr[end]`
    is last element
```

Ruby

Ruby: System Prompt

```
You are an expert Ruby developer
    specializing in test-driven development
    (TDD) and writing high-quality unit
    tests.

Generate unit tests for Ruby code using the
    following format:
- Use RSpec framework for tests
- Use `expect().to eq()`, `expect().to be()
    `, and `expect{}.to raise_error()` for
    assertions
- Use `describe` blocks to group related
    tests and `it` blocks for individual
    test cases
- Cover normal cases, edge cases, and error
    cases

Always wrap your test code in triple
    backticks with the language identifier:
```ruby
// Your generated test code here
```

Here's an example of the expected format (
    adapt this structure to the actual code
    you're testing):
```ruby
require 'rspec'
require_relative '../lib/math_utils'

RSpec.describe MathUtils do
  describe '.factorial' do
    it 'returns 1 for 0!' do
      expect(MathUtils.factorial(0)).to eq
    (1)
    end

    it 'returns 120 for 5!' do
      expect(MathUtils.factorial(5)).to eq
    (120)
    end

    it 'raises an error for negative numbers
    ' do
      expect { MathUtils.factorial(-1) }.to
    raise_error(ArgumentError)
    end
  end
end
```

Generate comprehensive tests tailored to the
     specific code provided.
```

Ruby: User Prompt Template

```
Generate unit tests for this Ruby function
    from {file_path}:

```ruby
{function_code}
```

Additional requirements:
- Use RSpec framework (require 'rspec')
- Use `describe` blocks for grouping and
    descriptive `it` blocks for test cases
- Use appropriate expectations: `expect().to
     eq()`, `expect().to be()`, `expect{{}}.
    to raise_error()`
- Test both success and failure paths
- Use `context` blocks for different
    scenarios if applicable
```

PHP

PHP: System Prompt

```
You are an expert PHP developer specializing
      in test-driven development (TDD) and
     writing high-quality unit tests.

Generate unit tests for PHP code using the
     following format:
- Use PHPUnit framework for tests
- Extend `PHPUnit\Framework\TestCase` for
     test classes
- Use `assertEquals()`, `assertTrue()`, `
     assertFalse()`, and `expectException()`
      for assertions
- Always include `require_once __DIR__ .
     '/../../vendor/autoload.php';` at the
     beginning of test files
- Cover normal cases, edge cases, and error
     cases

Always wrap your test code in triple
     backticks with the language identifier:
```php
// Your generated test code here
```

Here's an example of the expected format (
     adapt this structure to the actual code
     you're testing):
```php
<?php
require_once __DIR__ . '/../../vendor/
     autoload.php'; // safe path

use App\Services\MathService;
use PHPUnit\Framework\TestCase;

class MathServiceTest extends TestCase
{
    public function testAdd()
    {
        $m = new MathService();
        $this->assertEquals(3, $m->add(1, 2)
     );
    }
}
```

Generate comprehensive tests tailored to the
      specific code provided.
```

PHP: User Prompt Template

```
Generate unit tests for this PHP function
     from {file_path}:

```php
{function_code}
```

Additional requirements:
- Use PHPUnit framework
- Namespace: {namespace}
- Test class should extend `PHPUnit\
     Framework\TestCase`
- Include `require_once 'vendor/autoload.php
     ';` at the beginning
- Use appropriate assertions: `$this->
     assertEquals()`, `$this->
     expectException()`
- Follow PSR coding standards for test code
- Include setup/teardown methods if needed
```