Benchmarking, Analyzing, and Fine-tuning Large Language Models for Coding Tasks

Mohammad Abdollahi

A Thesis Submitted to the Faculty of Graduate Studies
In Partial Fulfillment of the Requirements
for the Degree of Master of Applied Science

Graduate Program in Electrical Engineering and Computer Science

York University
Toronto, Ontario

September 2026

© Mohammad Abdollahi, 2026

Abstract

Large Language Models (LLMs) generate, test, and repair code, yet improving them rests on three connected needs: good benchmarks, analysis of the limitations those benchmarks expose, and a way to turn those limitations into post-training signals. Study I, a systematic survey of 156 benchmark datasets spanning 32 coding tasks, finds that benchmarks score only final outputs and leave the model's intermediate reasoning unmeasured. Study II analyses the reasoning traces of four LLMs across 22,896 program-execution inferences, builds a nine-class error taxonomy, and shows that failures usually originate at one identifiable step whose correction often fixes the output. Study III applies this insight to reinforcement learning for repository-scale issue resolution, replacing the binary reward with a dense test-level reward and adaptively selecting and refining tests during training. The resulting 4B model resolves 28.2% of SWE-bench Verified instances against 27.2% for a reinforcement-learned baseline, while using 1.9% fewer tokens on solved instances.

Acknowledgements

First and foremost, I want to thank my parents. Although the distance between us has meant that we could not share these years in person, their love and belief in me have never felt far away. Every phone call, every word of encouragement across the miles, carried me through the difficult stretches of this degree. Whatever I have accomplished here rests on the foundation they gave me, and I hope this thesis stands as a small return on their sacrifices.

I am deeply grateful to my supervisors, Professor Hadi Hemmati and Professor Song Wang. Their guidance shaped not only this thesis but the way I think about research: how to ask a question worth answering, how to be honest about what the evidence does and does not show, and how to keep going when results refuse to cooperate. I thank them for their patience with my missteps, for the time they invested in reading and re-reading my work, and for the trust they placed in me from the very beginning.

My heartfelt thanks go to my partner, whose steady support and understanding sustained me through the long nights, the deadlines, and the moments of doubt. She celebrated every small success with me and made the setbacks lighter to carry, and this work would have been far harder, and far lonelier, without her.

Finally, I want to thank my friends and colleagues at York University. Their willingness to discuss half-formed ideas, review my drafts, question my assumptions, and lend a hand whenever I needed one improved this research immeasurably and made the journey a genuinely enjoyable one. I have been fortunate to learn alongside such generous people.

Contents

Page

Abstract **ii**

Acknowledgements **iii**

Contents **iv**

List of Tables **viii**

List of Figures **x**

1 Introduction **1**

1.1 Motivation 1

1.2 Research Questions 2

1.3 Contributions 3

1.4 Thesis Organization 4

2 Background and Related Work **5**

2.1 Large Language Models for Code 5

2.1.1 From General-Purpose Language Models to Code Models 5

2.1.2 Applications Across the Software Lifecycle 6

2.1.3 From Single Completions to Agents 6

2.2 Benchmarking Large Language Models for Code 6

2.2.1 Volume and Breadth of the Benchmark Ecosystem 6

2.2.2 How Performance Is Measured 7

2.2.3 Data Contamination 7

2.2.4 Existing Surveys and What They Miss 8

2.2.5 Documented Weaknesses 9

2.3 Reasoning in Large Language Models 9

2.3.1 Eliciting Reasoning Through Prompting 9

2.3.2 Reasoning-Specialised Models . . . 9
2.3.3 Evaluating Code Reasoning . . . 10
2.3.4 Faithfulness of Reasoning Traces . . . 10
2.3.5 Process-Level Supervision . . . 10
2.4 Post-Training Large Language Models . . . 11
2.4.1 Supervised Fine-Tuning . . . 11
2.4.2 Reinforcement Learning from Feedback . . . 11
2.4.3 Reinforcement Learning with Verifiable Rewards . . . 12
2.4.4 Policy Optimisation Algorithms . . . 12
2.4.5 Post-Training for Code . . . 13
2.5 Repository-Scale Issue Resolution . . . 13
2.5.1 From Classical Program Repair to LLM Agents . . . 13
2.5.2 The Task and Its Benchmarks . . . 14
2.5.3 Agent Scaffolds . . . 14
2.5.4 Training Environments and Verifiers . . . 14
2.5.5 Reinforcement Learning Recipes for Software-Engineering Agents . . . 15
2.5.6 Test Quality as an Optimisation Target . . . 15
2.5.7 Test Selection and Adaptive Difficulty . . . 16
2.6 Summary of Research Gaps . . . 16

3 Methodology **18**

3.1 Study I: Systematic Survey of the Benchmarking Landscape . . . 18
3.1.1 Why Survey the Benchmark Landscape? . . . 18
3.1.2 Survey Methodology . . . 19
3.2 Study II: White-Box Analysis of Reasoning on Code-Execution Inference . . . 21
3.2.1 Motivation and Research Questions . . . 21
3.2.2 What Counts as Reasoning in This Study . . . 23
3.2.3 Study Design . . . 24
3.3 Study III: Adaptive Test Selection for Reinforcement Learning on Issue Resolution . . . 27

3.3.1 Problem Formulation . . . 28
3.3.2 Method . . . 30
3.3.3 Experimental Setup . . . 33

4 Results **37**

4.1 Study I: The Benchmarking Landscape . . . 37
4.1.1 RQ1: What Does the Landscape Look Like? . . . 37
4.1.2 RQ2: How Are the Benchmarks Built, and How Good Is the Data? . . . 47
4.1.3 RQ3: How Is Performance Measured? . . . 51
4.1.4 RQ4: What Do the Benchmarks' Users Say Is Wrong with Them? . . . 54
4.1.5 RQ5: What Should Come Next? . . . 58
4.1.6 What Study I Establishes, and Where It Points . . . 60
4.2 Study II: Reasoning Errors on Code-Execution Inference . . . 60
4.2.1 RQ6: How Accurate Are Reasoning LLMs at Execution Inference? . . . 60
4.2.2 RQ7: Where in the Code Does Reasoning Break? . . . 61
4.2.3 RQ8: What Kinds of Failures Are These? A Nine-Class Taxonomy . . . 63
4.2.4 Further Analyses . . . 67
4.2.5 What Study II Establishes, and Where It Points . . . 70
4.3 Study III: Adaptive Test Selection for RL on Issue Resolution . . . 71
4.3.1 RQ11: Effectiveness of Test Refinement . . . 71
4.3.2 RQ9 and RQ10: Training Dynamics across Five Conditions . . . 72
4.3.3 RQ12: Held-Out Evaluation on SWE-bench Verified . . . 75
4.3.4 What Study III Establishes . . . 77

5 Discussion **78**

5.1 Study I: Threats to Validity . . . 78
5.2 Study II: Threats to Validity . . . 78
5.3 Study III: Limitations and Threats to Validity . . . 79

6 Conclusion and Future Work **81**

6.1 Summary of the Thesis 81
6.2 Implications 82
6.3 Future Work 82

Bibliography **84**

List of Tables

1 Comparison of contributions between our survey and the existing related surveys/reviews. . . 8

2 Overview of the datasets. **CC** denotes the Cyclomatic Complexity metric, and **HM** is the Halstead Metrics. . . . 24

3 States of the test-selection operator. Tests move between states at the end of each RL step, conditioned on how that step's rollouts performed on the instance. No state is visible to the agent; the partition governs validation and scoring only. . . . 31

4 Composition of the 800-instance training set, and its test-granularity correspondence with the evaluation benchmark. . . . 35

5 Distribution of code intelligence tasks across benchmark datasets, categorized by Software Development Lifecycle phases. . . . 39

6 Task scope of benchmark datasets, i.e., Single-Task vs. Multi-Task. . . . 40

7 Distribution of benchmark datasets by Code Context Granularity . . . 40

8 Classification of benchmarks based on Interaction Paradigm . . . 41

9 Distribution of benchmarks across various Application Domains . . . 42

10 Distribution of LLM benchmark datasets based on conference and journal venues . . . 43

11 Distribution of programming languages across benchmark datasets . . . 44

12 Distribution of LLMs in the subject studies . . . 46

13 Detailed distribution of benchmark sources . . . 48

14 Quality controls applied in data collection process . . . 49

15 How datasets ensured real-world representativeness . . . 49

16 Overview of Data Contamination Mitigation Approaches in Code Intelligence Datasets . . . 50

17 Detailed distribution of benchmark evaluation metrics . . . 52

18 Recommended Suite for Evaluating Code Intelligence Tasks . . . 54

19 Limitations of LLM benchmark datasets in code intelligence . . . 56

20 The number of unsuccessful inferences (i.e., program–input pairs) for the four reasoning models across different input types. Numbers in parentheses denote the count of unique programs within those unsuccessful cases. . . . 61

21 Distribution of reasoning failures for each LLM according to the identified failure patterns. . 66

22 Mann–Whitney U test comparing code complexity metrics between tasks where LLM reasoning failed and those where it succeeded across function-level and repository-level datasets. The last two columns show the average metric values for tasks with reasoning failures and successful reasoning, respectively. 67

23 Impact of tool-augmented reasoning on *Computation Errors* 69

24 Distribution of hidden reasoning errors in unfaithful traces (correct final output but flawed reasoning) mapped to our proposed taxonomy. 70

25 Held-out evaluation of 4B policies on SWE-bench Verified, against original (never-refined) test suites. Best value in each column in bold. 75

List of Figures

1 Big picture of the thesis. The three studies follow the benchmark/analyse/fine-tune pipeline, and each produces a finding (arrows) that motivates the next: Study I scopes the reasoning challenge for Study II, and Study II's localisable-fix finding yields the post-training signal used by Study III. 1

2 How the survey corpus was assembled: automated searches over four databases, three manual filtering stages, and a quality checklist, complemented by venue-based manual search. . . 20

3 A reasoning failure in the wild: (a) the program under analysis, its input, and the true output; (b) the model's step-by-step trace, with the first semantically wrong step highlighted; (c) the prompt that elicited the trace. The single faulty step – a misevaluation of Python's `split()` – determines the incorrect final answer. 22

4 The four-step design of Study II: benchmark and input preparation, reasoning-trace collection, error annotation against executed ground truth, and taxonomy construction. 24

5 Regular, edge-case, and invalid inputs for the running example of Figure 3. 25

6 Dual-level annotation of the failure from Figure 3: a single-statement inspection point plus a semantic failure-pattern label. 27

7 Chronological overview of LLM benchmark datasets by task. 38

8 Publication trend of LLM benchmark datasets (2020–2025). 39

9 Evolution of code context scope in LLM benchmark datasets (2021–2025) 41

10 Distribution of LLM benchmark dataset sizes (scaled using $\log_{10}$) 45

11 Distribution of citations for the papers introducing LLM benchmark datasets 45

12 Pipeline for extracting community-reported benchmark shortcomings: citation crawling over 14 representative benchmarks, LLM-assisted screening, and dual-author manual extraction and validation. 55

13 Community-reported limitations of LLM benchmark datasets, mapped to the representative datasets in which each was observed. 57

14 Success rates of the four reasoning LLMs on HumanEval+, LiveCodeBench, and REPOEXEC, broken out by input tier. 61

15 Distribution of inspection-point reasoning failure locations across all unsuccessful inferences for the evaluated LLMs on the regular and edge inputs of HumanEval+, LiveCodeBench, and REPOEXEC datasets. The chart categorizes the specific code locations (e.g., Basic Statement, For Statement) where the model's reasoning first diverged from the correct execution path. 62

16 The nine-class taxonomy of reasoning failures in LLM code inference, organised under four parent categories. 63

17 The distribution of error categories per LLM. Each bar represents an LLM, and each segment shows the contribution of a specific error category. 66

18 The tool-augmented reasoning pipeline: an LLM reasoning policy and a deterministic tool executor alternating over a centralised, append-only state. 68

19 Quality of the $n = 2{,}071$ tests that passed through refinement, before (blue) and after (orange) the operator: distributions of (a) test-suite mutation score, (b) line coverage, and (c) branch coverage. 72

20 Training dynamics of the five experimental conditions over 200 optimiser steps. Each panel shows average training reward (top) and training solve rate (bottom): (a) the SWE-Master baseline under the binary reward of Equation (3); (b) the dense reward with test selection over original suites; (c) test selection over suites refined offline before step one; (d) the full method with refinement waves fired by the reward-band rule – dashed lines mark the waves at steps 70 and 115; (e) the full method with waves fired by the reward-trend rule – the dashed line marks the wave at step 150. 73

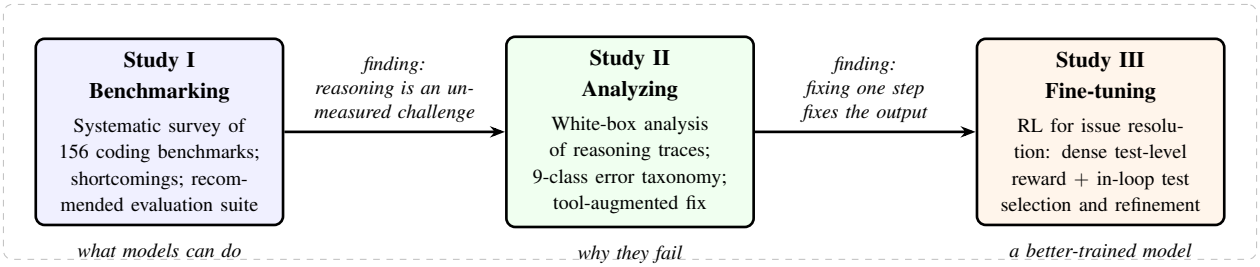


Figure 1: Big picture of the thesis. The three studies follow the benchmark/analyse/fine-tune pipeline, and each produces a finding (arrows) that motivates the next: Study I scopes the reasoning challenge for Study II, and Study II's localisable-fix finding yields the post-training signal used by Study III.

1 Introduction

1.1 Motivation

Large Language Models (LLMs) have become a standard tool for coding tasks. Code generation, completion, translation, testing, debugging, and program repair are increasingly delegated to models such as the GPT, Claude, Gemini, and DeepSeek families, and to code-specialised models built on top of them [1, 2, 3]. Tools such as GitHub Copilot and Cursor have carried these capabilities directly into everyday development workflows, so the question of how good these models really are – and how to make them better – is no longer academic. Making LLMs for code (LLMs4Code) genuinely better, however, is not a single problem but a pipeline of three connected activities:

1. **Benchmarking.** We can only improve what we can measure. Good benchmarks are needed to assess what current models can and cannot do, and to make comparisons across models and tasks meaningful and reproducible.
2. **Analysing limitations.** A benchmark score tells us *that* a model fails, not *why*. To make progress we must look inside the failures and characterise the limitations that the benchmarks expose.
3. **Post-training on high-quality data.** Once the limitations are understood, the most direct route to a better model is to turn those limitations into high-quality training signals and post-train (fine-tune) the model against them.

This thesis is organised along exactly these three lines, and the three studies that make it up were not chosen independently: each study produces a finding that motivates the next. Figure 1 shows the overall picture.

From benchmarking to a reasoning challenge (Study I → Study II). We began with a broad, systematic analysis of the benchmarking landscape for coding tasks, with particular attention to code generation, the most heavily benchmarked area [1]. Study I reviews 156 unique benchmark datasets from 142 studies covering 32 coding tasks. Beyond cataloguing how these benchmarks are built and where they fall short, the analysis surfaced a recurring theme: current benchmarks score final outputs but do not isolate the model's

intermediate *reasoning*, and step-by-step reasoning over program behaviour is one of the central challenges behind incorrect outputs. This is precisely why the second study focuses on reasoning.

From a reasoning analysis to a fixable signal (Study II → Study III). The second study opens the black box and analyses the reasoning traces that LLMs produce when inferring program behaviour [4]. It contributes a taxonomy of nine reasoning-error types and, most importantly for the thesis, an actionable finding: errors usually originate at a single identifiable step, and correcting that step is frequently sufficient to correct the final output – a tool-augmented pipeline that fixes one class of faulty steps repairs 69% of those failures. In other words, the limitations found by analysis can be converted into a concrete, localisable training signal. This is the bridge to the third study.

From a fixable signal to a harder task with our own method (Study III). The third study takes a deliberately harder, end-to-end task – program repair, in the form of repository-scale issue resolution – and builds a post-training method on top of the Study II finding. Rather than only inferring behaviour, the model must produce a patch that fixes a real GitHub issue, and it is trained with reinforcement learning against an execution-based verifier. Study II's finding translates here into a statement about *credit assignment*: standard practice collapses an entire agentic rollout into a single bit – did the patch fix every targeted test without breaking anything – which discards precisely the localised, per-step information that Study II showed to be decisive. Study III replaces that bit with a dense, test-level reward, and then goes further: instead of accepting the test suite as a fixed oracle, it lets the training loop *select* which tests define the reward and *repair* the tests that prove defective, with a container-validated gate ensuring repair reduces verifier error rather than introducing it.

Thesis Statement. LLMs for coding tasks can be systematically improved by (i) benchmarking them to expose where they fail, (ii) analysing their reasoning to characterise *why* they fail and to show that localised reasoning fixes propagate to correct outputs, and (iii) using that localisation principle to post-train a model on the harder task of repository-scale issue resolution, replacing the binary verifier reward with a dense, test-level signal whose defining test suite the training loop is allowed to select and repair – which measurably improves the tests it trains on, learns faster and converges higher in training than a binary-reward baseline, and resolves held-out issues at lower token cost.

1.2 Research Questions

The thesis is organised around twelve research questions, numbered consecutively across the three studies. Each study's methodology section restates the questions it addresses, and the results sections are labelled with the questions they answer.

Study I – Benchmarking. *RQ1 (Landscape).* When and where were LLM coding benchmarks published, which tasks and languages do they target, how large are they, and which models do they evaluate? *RQ2 (Construction and quality).* Where does their data come from, how are they organised, what quality controls

were applied, and how do they address real-world relevance and data contamination? *RQ3 (Evaluation).* Which metrics and evaluation pipelines are used, and how consistent are they across benchmarks? *RQ4 (Limitations).* What weaknesses have researchers who use these benchmarks reported about them? *RQ5 (Future directions).* Given those weaknesses, what should the next generation of benchmarks look like?

Study II – Analysing reasoning. *RQ6 (Performance).* How accurately do reasoning LLMs infer execution outcomes across regular, edge-case, and invalid inputs? *RQ7 (Failure location).* At which program statements does a reasoning trace first diverge from actual execution? *RQ8 (Failure pattern).* What recurring patterns do the failures follow, can they be organised into a taxonomy, and does correcting a single faulty step correct the final output?

Study III – Post-training for issue resolution. *RQ9 (Cost of the binary reward).* How much usable learning signal does the binary verifier reward discard in RL for issue resolution? *RQ10 (Dense reward and selection).* Does a dense, test-level reward with in-loop test selection improve sample efficiency over the binary baseline? *RQ11 (Safe refinement).* Can defective tests be repaired during training without injecting false negatives, and does refinement measurably improve test quality? *RQ12 (Held-out transfer).* Does the resulting policy transfer to a held-out benchmark scored on original, never-refined tests, and at what token cost?

1.3 Contributions

This thesis makes the following contributions:

1. A peer-reviewed systematic survey of 156 LLM-for-code benchmarks from 142 studies, with a detailed taxonomy of their construction, quality controls, evaluation pipelines, an evidence-backed catalogue of five recurring shortcomings, a recommended five-dimensional evaluation suite, and a unified framework for building the next generation of benchmarks (Study I, published at ACM TOSEM [1]).
2. The first white-box study of LLM reasoning on code-execution inference: a 477-program augmented benchmark with a three-tier input model, 22,896 collected inferences across four state-of-the-art reasoning LLMs, a nine-class error taxonomy grounded in statement-level annotation, a tool-augmented pipeline resolving 69% of the dominant error class, and reproducibility (80.2% error-reproduction rate) and faithfulness (9.7% unfaithful correct traces) analyses (Study II, published at ACM TOSEM [4]).
3. A reinforcement-learning method for repository-scale issue resolution that treats the reward-defining test suite as an object the training loop may adapt: a dense, test-level reward with a regression guard and a token penalty; an in-loop, policy-conditioned test-selection curriculum that adapts the acceptance criterion *within* an instance, over its verifier sub-goals, at the end of each RL step; and a gold-patch-validated test-refinement operator that repairs defective tests without injecting false negatives. Both operate entirely on the reward side of the loop, leaving the agent's prompt and observation space unchanged. Refinement raises the mean mutation score of the 2,071 tests it touches by 5.3 points under policy-independent measurement; trained at 4B scale on an 800-instance corpus spanning 706 repositories, the full method reaches a 15% training solve rate in under a third of the optimiser steps of the binary-reward baseline,

converges to a higher terminal training solve rate than that baseline, than selection alone, and than an offline-refined variant, and exceeds the baseline's held-out resolve rate on SWE-bench Verified (28.2% against 27.2%) at 1.9% lower median token cost (Study III).

1.4 Thesis Organization

The remainder of the thesis is organised as follows. Chapter 2 reviews the background and related work for the three lines of the thesis: LLMs for code, benchmarking, reasoning, post-training, and repository-scale issue resolution. Chapter 3 presents the methodology of the three studies – the survey protocol of Study I, the empirical design of Study II, and the training method and experimental apparatus of Study III. Chapter 4 reports their results in the same order. Chapter 5 discusses the threats to validity and limitations of each study, and Chapter 6 concludes the thesis and outlines future work.

2 Background and Related Work

This chapter establishes the technical background the three studies build on, and positions each of them against the prior work it engages. It proceeds in the order the thesis does. Section 2.1 introduces large language models for code: how they are built, what they are used for, and how they have grown into agentic systems. Section 2.2 reviews how such models are evaluated – the benchmark ecosystem, its metrics and methodologies, its exposure to contamination, and the surveys that have tried to map it – and closes on the gap that motivates Study I. Section 2.3 turns to reasoning: how it is elicited, how reasoning-specialised models differ, how code reasoning in particular is evaluated, and what is known about whether a model's stated reasoning reflects its actual computation, closing on the gap that motivates Study II. Section 2.4 covers post-training – supervised fine-tuning, reinforcement learning from feedback and from verifiable rewards, and the policy-optimisation algorithms involved, including the formulation of GRPO that Study III uses directly. Section 2.5 then narrows to repository-scale issue resolution: its benchmarks, agent scaffolds, training environments, RL recipes, and the emerging literature on test quality and adaptive difficulty, closing on the gap that motivates Study III. Section 2.6 collects the three gaps in one place.

2.1 Large Language Models for Code

2.1.1 From General-Purpose Language Models to Code Models

Contemporary language models are decoder-only Transformers [5] trained autoregressively to predict the next token over very large text corpora. Scale turned out to matter enormously: GPT-2 [6] showed that a single unsupervised model could perform many tasks it was never explicitly trained for, and GPT-3 [7] showed that at sufficient scale such a model could be steered by examples supplied in its prompt alone, without gradient updates. Because public code repositories were part of these corpora, general-purpose models acquired a surprising amount of programming ability as a side effect.

Deliberately code-specialised models followed. Codex [8] fine-tuned a GPT model on public code and demonstrated that it could solve a substantial fraction of hand-written programming problems, a result that also gave the field HumanEval, still its most-cited benchmark. Open code-centric foundation models arrived shortly after – CodeLlama [9], StarCoder [10], and DeepSeek-Coder [11] – trained on mixtures of source code and natural language so that they can move between the two. Three training ingredients distinguish code models from their general-purpose ancestors: pre-training corpora dominated by source code, often with repository structure preserved; *fill-in-the-middle* objectives that condition on both a prefix and a suffix, matching how developers actually edit; and instruction tuning on curated pairs of natural-language requests and code responses, which is what makes a raw completion engine usable as an assistant.

2.1.2 Applications Across the Software Lifecycle

These models are now applied at nearly every stage of software development. The most visible application is code generation and completion, where a natural-language prompt or a partially written function is turned into working code [8, 12]; commercial assistants have carried this capability into mainstream editors. Beyond authoring new code, LLMs are used for summarisation, condensing unfamiliar functions into explanations that make a codebase navigable [2], and for translation between languages, which offers a starting point for migrating legacy systems [3]. In testing, they generate unit tests, synthesise assertions, and derive cases from requirements or existing implementations [13, 14]. In maintenance, they are applied to program repair [15, 16], vulnerability detection [17], and software-engineering question answering [18].

What unites these applications is that the model's output is *probabilistic* and must be checked. Unlike a compiler or a refactoring tool, an LLM offers no guarantee of correctness: its output can contain hallucinated APIs, security flaws, or semantic errors that are subtle precisely because the surrounding code looks idiomatic [4]. This is the fundamental reason the community has standardised on empirical benchmarking as its instrument for assessing capability, and it is why the quality of that instrument – the subject of Study I – matters so much.

2.1.3 From Single Completions to Agents

The most consequential recent shift is architectural rather than model-level. Rather than issuing one prompt and accepting one completion, *agentic* systems place the model in a loop with an environment: it can read files, search a repository, run tests, observe the resulting output, and revise its plan before acting again. SWE-agent [19] demonstrated that a purpose-designed agent–computer interface substantially improves an LLM's ability to resolve real repository issues; AutoCodeRover [20] combines program-structure-aware search with fault localisation; and Agentless [21] showed that a fixed localise–repair–validate pipeline can rival more elaborate agents, a useful reminder that scaffold complexity is not automatically an advantage.

Agentic operation changes the evaluation problem in two ways that matter for this thesis. First, a single task now produces a long trajectory of interdependent decisions rather than one output, so a verdict on the final artefact says very little about where a failure originated – the concern Study II takes up at the level of reasoning traces. Second, because trajectories are long and executed against real repositories, they are expensive; the cost of a wasted rollout becomes a first-order concern in training, which is precisely the problem Study III addresses.

2.2 Benchmarking Large Language Models for Code

2.2.1 Volume and Breadth of the Benchmark Ecosystem

The post-GPT-2 era has produced an extraordinary number of coding benchmarks. The foundational generation targeted function-level synthesis from natural language: HumanEval [8], MBPP [12], APPS [22],

and the multi-task CodeXGLUE suite [23]. Subsequent work extended this template along three axes. *Language coverage* grew with HumanEval-X, MBXP, and MultiPL-E [24], which port existing problems into many languages. *Context scope* grew from isolated functions to whole repositories with RepoBench [25], CrossCodeEval [26], REPOEXEC [27], and SWE-Bench [28]. And *execution awareness* grew with benchmarks that ask about runtime behaviour rather than source text – CRUXEval [29], REval [30], and LiveCodeBench [31], the last of which also continuously refreshes its problems to stay ahead of training cut-offs.

This growth has been uneven in ways that shape what the field can claim. Code generation is by a wide margin the most heavily benchmarked task, while comprehension, repair, and testing receive far less attention; Python dominates the language distribution; and benchmarks vary widely in how their data was sourced, how carefully it was validated, and how faithfully it reflects the software developers actually maintain. Compounding this, model capability has improved faster than the benchmarks meant to measure it, so datasets that discriminated sharply between models in 2021 are saturated today. Quantifying these imbalances systematically, rather than anecdotally, is the task of Study I.

2.2.2 How Performance Is Measured

Evaluation methodology divides into a few recognisable families. *Execution-based* evaluation runs the generated code against tests and is the dominant approach for any task where functional correctness is well defined; its canonical metric is *Pass@k*, the probability that at least one of k sampled solutions passes [8], alongside compilation rates, resolution rates for repair tasks, and runtime or memory measurements where efficiency matters. *Similarity-based* evaluation compares generated text against references using BLEU, CodeBLEU, exact match, edit distance, or embedding-based scores, and remains appropriate for tasks such as summarisation and translation where executability is not the criterion. *Human evaluation* applies structured rubrics to qualities automated metrics miss, at correspondingly higher cost. More recently, *LLM-as-a-judge* protocols use a strong model with a rubric as a scalable proxy for human assessment, with reported correlations to human judgement that are good but not perfect.

Each family has a characteristic blind spot, and one of them recurs throughout this thesis. Execution-based evaluation certifies that code passes the tests it was given – nothing more. If the tests are thin, incorrect, or incidental to the task, the certification is correspondingly weak, and the metric offers no way to detect that from the outside. This is a limitation of the instrument that Study I documents at the level of the benchmark ecosystem and that Study III confronts directly, since in reinforcement learning the same test suite that evaluates a model also *trains* it.

2.2.3 Data Contamination

Benchmarks are only meaningful if the model has not already seen their contents. Because both benchmark authors and model trainers draw on the same public sources, this assumption is fragile, and leakage has been documented repeatedly in this domain [32]. The community has developed four broad mitigations:

temporal isolation, which admits only data created after the models' training cut-offs; *corpus exclusion*, which algorithmically removes overlap using N-gram matching, MinHash deduplication, or exact matching; *novelty and perturbation*, which either hand-writes new problems or semantically perturbs existing ones so that memorisation does not transfer; and *private holdouts*, which withhold the evaluation data entirely and score submissions in a sandbox. Each buys a different kind of protection at a different cost, and, as Study I quantifies, a large fraction of published benchmarks adopt none of them.

2.2.4 Existing Surveys and What They Miss

Several surveys have attempted to map parts of this ecosystem. Study I compares them systematically along ten aspects – the time frame covered, data sources, data-quality considerations, LLM-specific focus, coverage of the current landscape, analysis of key benchmark characteristics, reported limitations, evaluation pipelines, future directions, and data contamination – and Table 1 presents that comparison.

Table 1: Comparison of contributions between our survey and the existing related surveys/reviews.

No	Studies	Time Frame of Survey	Data Source	Data Quality	LLM Focus	Current Landscape	Key Characteristics Analysis	Limitations	Evaluation	Future Directions	Data Contamination
1	Hu et al. [33]	2014-2025	✓	×	✓	✓	×	✓	✓	✓	×
2	Wang et al. [34]	2018-2024	✓	×	✓	✓	×	✓	×	✓	×
3	Koohestani et al. [35]	2014-2025	×	×	×	✓	×	✓	×	×	×
4	Zhu et al. [36]	2005-2025	✓	✓	×	✓	✓	✓	×	✓	×
5	Cao et al. [37]	2014-2024	✓	✓	✓	×	×	✓	×	✓	×
6	Chen et al. [38]	2021-2025	✓	×	✓	×	×	✓	×	×	✓
7	Zhou et al. [39]	2017-2025	✓	×	✓	×	×	✓	×	✓	✓
8	Zheng et al. [40]	2020-2024	×	×	✓	✓	×	×	✓	×	×
9	Chen et al. [41]	2021-2024	×	×	✓	×	×	×	✓	×	×
10	Zhang et al. [42]	2021-2024	×	×	✓	×	×	×	×	×	×
11	Jin et al. [43]	2021-2024	×	×	×	×	×	×	×	×	×
12	Our survey	2020-2025	✓	✓	✓	✓	✓	✓	✓	✓	✓

The most closely related studies are the surveys by Hu et al. [33] and Wang et al. [34]. Hu et al. provide a broad systematic review of 291 benchmarks for evaluating LLMs in software engineering, focusing on construction methods, evaluation metrics, and future challenges; Wang et al. categorise CodeLLM benchmarks by software-engineering task across the development life cycle. Both, however, largely enumerate datasets by task without jointly examining how the datasets are *constructed*, how their *quality* is controlled, how *representative* they are of real software, which *evaluation pipelines* they use, and how exposed they are to *data contamination*.

Other related efforts are narrower in scope. Koohestani et al. [35] address the operational challenge of benchmark discoverability with the BenchScout search tool; Zhu et al. [36] concentrate solely on benchmarks for software defect detection; Cao et al. [37] contribute HOW2BENCH, a 55-criterion checklist for benchmark creation; and Chen et al. [38] and Zhou et al. [39] focus specifically on data contamination, the latter introducing LessLeak-Bench to quantify leakage across 83 benchmarks. Broader surveys of Code LLMs [40, 42], of evaluation methods for code generation [41], and of LLM-based agents in software engineering [43] treat benchmarks only as a secondary concern.

2.2.5 Documented Weaknesses

Across the empirical literature, five weaknesses recur [1]: (1) *limited task complexity and realism* – many benchmarks use single-function, algorithmic puzzles, and Python dominates roughly 77% of datasets; (2) *weak data quality* – LLM-generated content with little human validation and a bias toward short, easy items; (3) *inadequate test coverage* – thin or incorrect canonical tests, a documented issue even in HumanEval; (4) *narrow evaluation* – *Pass@k* overwhelmingly dominant, with efficiency, readability, and safety under-represented; and (5) *data leakage* – public-repository sources and pre-cutoff data inflating scores [32]. A further, cross-cutting observation is that benchmarks score only *final outputs*: they cannot tell whether a correct answer was reached by sound reasoning, nor localise the step at which an incorrect one went wrong.

Gap (motivates Study I). The field lacks a unified, evidence-backed map that goes beyond enumeration to categorise benchmarks by task and lifecycle phase, examine construction and quality control jointly with evaluation and contamination, extract shortcomings as reported by the papers that actually use these benchmarks, and translate the result into actionable recommendations – including the observation that intermediate reasoning is unmeasured, which directly motivates Study II.

2.3 Reasoning in Large Language Models

2.3.1 Eliciting Reasoning Through Prompting

Reasoning was first approached as a prompting problem. Chain-of-Thought prompting [44] showed that asking a model to produce intermediate steps before its answer substantially improves performance on multi-step tasks, and Kojima et al. [45] showed that even the instruction "let's think step by step" suffices to trigger the effect. Refinements followed quickly: Self-Consistency [46] samples many reasoning paths and takes a majority vote over their answers, trading computation for robustness against a single bad path; Tree-of-Thought [47] generalises the linear chain into a search over branching partial solutions. A parallel line offloads the parts of reasoning that language models are worst at: ReAct [48] interleaves reasoning with environment actions, Toolformer [49] teaches models to call external tools, and PAL [50] and Program-of-Thought [51] express reasoning as executable programs so that an interpreter, rather than the model, performs the computation. The last of these is directly relevant to Study II, whose tool-augmented experiment applies the same principle to program-execution reasoning.

2.3.2 Reasoning-Specialised Models

More recently the capability has been internalised rather than prompted. Models such as OpenAI o4-mini [52], DeepSeek-R1 [53], Gemini 2.5 Flash [54], and Claude 4 Sonnet [55] are explicitly optimised for multi-step inference and emit extended reasoning traces by default. What distinguishes them is not only scale but training procedure: DeepSeek-R1, for instance, combines reasoning-focused supervised fine-tuning with reinforcement learning against outcome and process rewards, and the resulting models allocate substan-

tially more computation at inference time before committing to an answer. Because these models produce their traces as a matter of course, they make the reasoning process available for inspection in a way earlier models did not – an opportunity Study II exploits.

2.3.3 Evaluating Code Reasoning

Code reasoning – deducing a program's behaviour by mentally executing it over control flow, data structures, and state changes – has been evaluated along two lines. This thesis uses the term in that operational sense, as the model's externalised simulation of execution, rather than in the narrower sense of logical inference; Section 3.2.2 makes the distinction precise and shows where the two diverge in the taxonomy. *Output-focused* work measures whether the model predicts the right result: CRUXEval [29] tests input and output prediction on short functions; CodeMind [56] adds natural-language specifications; later work extends the setting to chain-of-thought quality [57], to nineteen programming languages [58], to type inference [59], and to correlating performance with static and dynamic program properties [60]. *Black-box* analyses probe reasoning indirectly, by comparing LLMs to abstract computational devices [61], by collecting human judgements of reasoning quality [62], or by measuring consistency across tasks of increasing difficulty, as in REval [30] and CoCoNUT [63].

Both lines share a limitation. They treat the trace as opaque: the reported quantity is whether the answer is right, or how consistent answers are across conditions, never *where* inside the trace the model's simulated program state first diverged from reality. Failures are therefore counted but not diagnosed.

2.3.4 Faithfulness of Reasoning Traces

A related question is whether a model's stated reasoning reflects the computation that actually produced its answer. Because traces are optimised to precede correct answers rather than to be internally valid, a correct answer can in principle rest on invalid reasoning, and an articulated justification may be post-hoc. This matters practically: if traces are used to build trust, to debug failures, or – as in process supervision – to train the model, then unfaithful traces mislead in proportion to how much weight is placed on them. Prior work has raised the concern, but for program execution the question is unusually tractable, because ground truth for every intermediate state can be obtained simply by running the program. Study II takes advantage of exactly this to audit faithfulness directly.

2.3.5 Process-Level Supervision

A final line argues that supervising the reasoning *process* produces better reasoners than supervising outcomes alone. Process-reward models score individual steps rather than final answers and have been shown to outperform outcome-only supervision on mathematical reasoning [64, 65]. Bootstrapped-rationale methods take a complementary route: STaR [66] iteratively trains a model on the rationales that led it to correct

answers, while distillation approaches transfer a stronger model's rationales to a smaller one [67, 68]. The common premise is that step-level signal is more informative than a terminal verdict – a premise this thesis returns to in Study III, where the same argument is made about test-level rather than rollout-level rewards. These techniques, however, were developed for mathematics, where a step is a self-contained inference; program execution has stronger and cheaper ground truth, but no comparable body of work exploits it.

Gap (motivates Study II). No prior work annotates code-reasoning traces at the granularity of the single statement where the trace first diverges from true execution, organises the resulting failures into a taxonomy separating mechanical simulation errors from systemic epistemic ones, audits whether correct answers rest on faithful traces, or tests whether deterministically repairing one faulty step propagates to a correct final output. Establishing that last point is what turns diagnosis into an actionable training signal, and bridges to Study III.

2.4 Post-Training Large Language Models

Pre-training produces a model that can continue text; post-training is what makes it useful, aligned, and specialised. This section covers the techniques Study III builds on.

2.4.1 Supervised Fine-Tuning

The simplest route is supervised fine-tuning (SFT): continue training on curated pairs of prompts and desired responses, whether human-written demonstrations, filtered model outputs, or rationales [66, 68]. SFT is stable, cheap, and effective at inducing a format or a style – including agentic trajectory formats. Its limitation is intrinsic to the objective: maximising the likelihood of reference text teaches the model to *imitate*, not to achieve. A patch that differs from the reference but repairs the bug earns no credit, while a patch that mimics the reference's surface form and fails earns full credit. Performance is therefore bounded by the quality and coverage of the demonstrations.

2.4.2 Reinforcement Learning from Feedback

Reinforcement learning removes that ceiling by optimising a reward over the model's own outputs. RLHF [69] trains a reward model on human preference comparisons and then optimises the policy against it, and RLAIF [70] substitutes model-generated preferences to reduce annotation cost. Both are well suited to objectives that are easy to judge but hard to specify, such as helpfulness or tone. Both also inherit a characteristic failure mode: the learned reward model is an imperfect proxy, and sufficiently aggressive optimisation finds inputs that score well under the proxy while violating the intent behind it.

2.4.3 Reinforcement Learning with Verifiable Rewards

In domains where correctness can be checked mechanically, the proxy can be replaced by the check itself. This is the premise of RL with verifiable rewards (RLVR), which underlies the training of modern reasoning models [53]: instead of a learned scorer, the reward comes from a deterministic verifier – a mathematics answer checker, or, for code, a test suite. In code specifically, RL against execution feedback has a longer history: CodeRL [71] uses unit-test outcomes as a critic signal, PPOCoder [72] combines execution feedback with syntactic and semantic matching, and RLTF [73] exploits fine-grained feedback from the interpreter, including error type and location. Historically these operated at single-function granularity on competitive-programming-style problems rather than on realistic engineering tasks.

RLVR is often described as though the verifier were the objective. It is not. The verifier is an *estimator* of an objective – correctness – that cannot be observed directly, and like any estimator it has error. The distinction matters because optimisers exploit estimator error: the phenomenon of *reward hacking*, in which a policy maximises the measured reward while defeating its intent, is a long-recognised failure mode of misspecified objectives [74], and in code RL it takes a concrete form – patches that satisfy a weak test suite without repairing the underlying defect. Recent audits find such exploitable suites in roughly a quarter of sampled instances in widely used training environments [75], and false negatives – correct patches rejected by over-specific tests – are reported to degrade training even faster than false positives [76]. For code, then, verifier error is measurable and consequential in both directions, and this observation is the conceptual foundation of Study III.

2.4.4 Policy Optimisation Algorithms

The optimiser used to turn a reward into a parameter update matters for what the reward can express. Proximal Policy Optimization (PPO) [77] is the classical choice – it stabilises policy-gradient training with a clipped surrogate objective and estimates advantages with a learned value function – and it powered the original RLHF pipelines [69]. DPO [78] eliminates both the reward model and the RL loop by reparameterising preference learning as a classification problem over preferred and dispreferred responses, which is efficient but requires paired preference data and does not naturally accommodate a scalar execution signal.

Group Relative Policy Optimisation (GRPO) [79] occupies a middle ground and is the algorithm Study III uses. Its central idea is to discard the learned value function and estimate the baseline empirically, from a group of rollouts sampled for the same prompt. For a prompt x, sample G trajectories $\{\tau_1, \ldots, \tau_G\} \sim \pi_{\theta_{\text{old}}}(\cdot \mid x)$ and compute scalar rewards $R_i = R(\tau_i)$. The advantage assigned to every token of trajectory i is the group-normalised reward,

$$A_i = \frac{R_i - \mu_G}{\sigma_G + \delta}, \qquad \mu_G = \frac{1}{G}\sum_{j=1}^{G} R_j, \qquad \sigma_G = \sqrt{\frac{1}{G}\sum_{j=1}^{G}(R_j - \mu_G)^2}, \tag{1}$$

with $\delta > 0$ for numerical stability. The policy is then updated by maximising a clipped surrogate objective with a KL penalty against a reference policy,

$$\mathcal{J}(\theta) = \mathbb{E}\left[\frac{1}{G}\sum_{i=1}^{G}\frac{1}{|\tau_i|}\sum_{k=1}^{|\tau_i|}\min\Big(\rho_{i,k}(\theta)A_i,\ \mathrm{clip}\big(\rho_{i,k}(\theta),\,1-\varepsilon_{\text{low}},\,1+\varepsilon_{\text{high}}\big)A_i\Big)\right] - \beta_{\text{KL}}\,\mathbb{D}_{\text{KL}}[\pi_\theta \,\|\, \pi_{\text{ref}}], \qquad (2)$$

where $\rho_{i,k}(\theta) = \pi_\theta(a_{i,k} \mid s_{i,k})/\pi_{\theta_{\text{old}}}(a_{i,k} \mid s_{i,k})$ is the token-level importance ratio. A common variant computes the baseline leave-one-out, $A_i = R_i - \frac{1}{G-1}\sum_{j\neq i} R_j$, which removes the self-contribution bias of the group mean.

Dispensing with the critic is what makes GRPO attractive for agentic tasks, where trajectories are long and value estimation is difficult – and, as Study III observes, it also has an unappreciated consequence for methods that modify the reward during training. It does, however, introduce a dependency that becomes central to this thesis: because the baseline is estimated *within* a group, the informativeness of an update depends on the rewards inside that group differing from one another. A reward that assigns the same value to every member of a group produces no gradient at all.

2.4.5 Post-Training for Code

Applying these techniques to code raises a practical question that general-purpose alignment does not face: where does the reward come from at scale? Human preferences are expensive and poorly suited to judging whether a patch is correct; learned reward models inherit the proxy problem; and execution-based rewards, while trustworthy, require a runnable environment per task – a repository at the right commit, with dependencies installed and a test runner that reports per-test outcomes. Building such environments in bulk is a research contribution in its own right, which is why the training corpora described in the next section are as central to this literature as the algorithms.

2.5 Repository-Scale Issue Resolution

2.5.1 From Classical Program Repair to LLM Agents

Automated program repair long predates LLMs. Generate-and-validate approaches such as GenProg [80] search a space of syntactic edits and accept a candidate when the test suite passes; semantics-based approaches such as Angelix [81] extract constraints from failing tests and synthesise patches that satisfy them. Two lessons from this classical literature carry directly into the LLM era. First, the test suite has always been the de facto correctness oracle, and its weakness has always been the field's central validity problem: the classical literature documented *plausible but incorrect* patches – edits that pass the suite while breaking unspecified behaviour – as the dominant failure mode of test-driven repair, the same phenomenon that reappears in RL training as reward hacking. Second, repair quality is bounded by fault localisation, which is

why modern agents invest so much of their trajectory in navigation and comprehension before editing. The step from search to learning was taken first in program synthesis: DeepCoder [82] trained a neural network to predict, from input–output examples, which primitives a target program uses, and used the prediction to prune an enumerative search – an early demonstration that a learned prior and an execution-based check compose, the same division of labour between generator and verifier that every LLM-era recipe below relies on. LLMs changed the patch generator – from syntactic search to learned generation – but they inherited the oracle problem unchanged, which is precisely the part of the pipeline Study III targets.

2.5.2 The Task and Its Benchmarks

Issue resolution is the hardest widely studied coding task: given a real GitHub issue and a snapshot of the repository at the commit where it was filed, produce a patch that makes the project's failing tests pass without breaking those that already pass. It subsumes fault localisation across many files, comprehension of unfamiliar code, and edit generation that respects existing behaviour. SWE-Bench [28] established the format; SWE-bench Verified [83] is a 500-instance subset filtered by human annotators to remove under-specified issues and broken tests, and is the field's standard held-out benchmark; SWE-Bench+ [84] re-examines the original for solution leakage and weak tests. The benchmark's structure – *fail-to-pass* tests that a correct patch must repair and *pass-to-pass* tests it must not break – is what makes automatic verification possible, and it is also what Study III scrutinises.

2.5.3 Agent Scaffolds

Because the task requires interaction, results depend heavily on the scaffold wrapped around the model. SWE-agent [19] contributed a purpose-built agent–computer interface with commands for navigating and editing files; AutoCodeRover [20] adds program-structure-aware search and spectrum-based fault localisation; Agentless [21] replaces the agent loop with a fixed three-phase pipeline. For the purposes of this thesis, the scaffold is held fixed: Study III changes only the reward, so that measured differences cannot be attributed to a better harness.

2.5.4 Training Environments and Verifiers

Training, as opposed to evaluation, needs environments in bulk. SWE-Gym [85] provides 2,438 instances across 11 repositories with executable environments and curated tests. R2E-Gym [86] contributes the verifier abstraction this line of work relies on – per-instance containers exposing a test-running entry point whose output parses into a canonical status map – together with procedurally curated environments and automatically generated reproduction tests. SWE-rebench [87] scales the idea to a continuously updated corpus of over twenty thousand instances, and additionally supplies per-instance quality annotations produced by applying an LLM to the human-annotation rubric behind SWE-bench Verified. What all of these share is that test suites are produced *once*, at environment-construction time, and then held fixed for the lifetime of the

environment.

2.5.5 Reinforcement Learning Recipes for Software-Engineering Agents

The dominant recipe pre-trains a code model, supervised fine-tunes it on patch-style agent traces, and applies GRPO or a close variant against an execution verifier. SWE-RL [88] demonstrated that RL over open software-evolution data sharpens issue-resolution reasoning. SWE-Master [89] and DeepSWE [90] are representative of current practice: both train Qwen-family policies against R2E-Gym-style containerised verifiers, with small group sizes and leave-one-out or group-mean baselines. Agent-RLVR [91] confronts the sparse-reward landscape of agentic RL from the agent side rather than the reward side: it injects guidance – strategic plans and error feedback – into unsuccessful rollouts to steer them toward successful trajectories, and reports 27.8% on SWE-bench Verified with a 72B policy. Its diagnosis of the problem matches this thesis's; its remedy leaves the binary verifier untouched. LeDex [92] likewise pairs supervised fine-tuning with RL, but sources its training data from execution-verified explanation-and-refinement trajectories, showing that execution-checked supervision improves both a model's self-debugging accuracy and the usefulness of the explanations it produces. SWE-RM [93] takes a different route again, observing that unit tests in real repositories are frequently over-specific or unrelated to the target issue, and responding by training a learned reward model to supplement or replace execution feedback.

Across all of these the reward is the same: a patch is credited if and only if it repairs every targeted test while breaking no regression test (formalised in Section 3.3.1). This single design choice carries two costs that Study III addresses. It compresses a rollout of fifty to a hundred and fifty agent actions into one bit, so that under the group-baselined estimator of Equation (1) a group whose members all agree contributes no gradient. And it treats the suite as the objective when the suite is only a noisy estimator of it – an audit of code RL environments finds that roughly a quarter of sampled instances have suites weak enough to accept a container-verified incorrect patch [75], with the resulting noise persistent per instance rather than averaging out across sampling.

2.5.6 Test Quality as an Optimisation Target

A recent line of work responds by treating test quality as something to be optimised rather than assumed. CURE [94] co-evolves a code generator and a unit-test generator with no ground-truth code, rewarding a test by a precision criterion: it should fail on all incorrect solutions and pass on all correct ones. UTRL [95] casts the same relationship adversarially, with a test generator maximising discrimination against a code generator maximising pass rate. MIST-RL [96] builds suites incrementally under a mutation-score reward, arguing that a suite's value lies in its discriminative power rather than its size or coverage. EvolveCoder [97] evolves test cases iteratively against the execution behaviour of candidate solutions, and shows that the harder, more discriminative tests this produces translate into larger downstream RL gains; Code-A1 [98] trains separate code and test models with opposing objectives – one maximising test passage, the other maximising bug detection – so that white-box test generation cannot collude with the generator it is meant

to check. All of these operate at function level, on self-contained problems, with a co-trained model as the tester. CodeContests-O [99] refines test cases iteratively under execution feedback and demonstrates downstream RL gains, but refinement happens once, during dataset construction.

2.5.7 Test Selection and Adaptive Difficulty

Classical software engineering has long studied which tests to run for a given change. Regression test selection and prioritisation [100, 101] identify informative subsets using coverage, dependency, or execution-history information, on the premise that a well-chosen subset preserves most of a suite's fault-detection power at a fraction of its cost. A complementary classical instrument is *mutation testing* [102], which measures a suite's discriminative power by checking how many seeded faults it detects – the policy-independent notion of test quality that Study III's discussion invokes when addressing the circularity of policy-conditioned selection. Both literatures are mature, but they have been applied to reducing the cost of continuous integration and assessing suite adequacy, not to shaping a learning signal.

Separately, machine learning has long known that the *order and difficulty* of training experience matters. Curriculum learning [103] presents easier examples before harder ones and demonstrates faster convergence and better optima; its standard unit of difficulty is the training *instance*. More recently, the RL literature has recognised that a static task distribution produces vanishing signal as the policy improves. RLVE [104] argues that environments should adapt their difficulty to the current policy, and implements this through procedural problem generation – an actuator unavailable for issue resolution, since one cannot synthesise a harder variant of a specific GitHub issue. DAPO [105] and GRESO [106] attack the same degeneracy at the prompt level: DAPO filters out degenerate groups after their rollouts have been paid for, while GRESO predicts and skips uninformative prompts in advance. Both treat per-instance difficulty as immutable and respond by discarding data rather than by making the instance informative again.

Gap (motivates Study III). No existing approach performs *in-loop, policy-conditioned* adaptation of the reward-defining test suite for repository-scale issue resolution. Offline refinement is stale by construction, because which tests carry signal depends on the current policy and because exploits of a weak suite are emergent – they cannot be audited before the policy discovers them. Learned reward models replace the execution signal rather than repairing it, forgoing the gold patch as an external ground truth. And prompt-level filtering removes degenerate instances from training instead of returning them to the informative regime. Study III occupies this gap.

2.6 Summary of Research Gaps

Three gaps emerge from this review, and they are connected rather than independent. First, the benchmarking literature lacks a unified, evidence-grounded map of how LLM-for-code benchmarks are constructed, controlled, evaluated, and criticised – and, more fundamentally, every benchmark in it scores final outputs only, leaving the reasoning that produced them unmeasured. Second, the code-reasoning literature evaluates

that reasoning as a black box, counting failures without localising them, and has not tested whether reasoning failures are repairable at the level of the individual step. Third, the post-training literature for issue resolution inherits a binary, fixed verifier that both wastes expensive rollouts and misestimates correctness, and no existing method adapts that verifier during training. The three studies of this thesis address these gaps in turn, and each supplies the finding that makes the next one possible.

3 Methodology

This chapter describes how the three studies of the thesis were conducted. The studies share one arc – benchmark, analyse, fine-tune – but they are methodologically distinct: Study I is a systematic literature survey, Study II is a controlled empirical study with human annotation, and Study III is a reinforcement-learning training method with its experimental apparatus. Each is presented in turn: its motivation and research questions, then its protocol. The results of all three are reported together in Chapter 4, and their threats to validity are discussed in Chapter 5.

3.1 Study I: Systematic Survey of the Benchmarking Landscape

Study I is the measurement pillar of the thesis: a systematic survey of the benchmark datasets used to evaluate LLMs on coding tasks between January 2020 and June 2025, published at ACM TOSEM [1]. This section motivates the survey, states its research questions, and describes its protocol; the findings are reported in Section 4.1.

3.1.1 Why Survey the Benchmark Landscape?

LLM-based coding tools behave very differently from the deterministic tools that software engineering is used to. Given a natural-language description or a code fragment, an LLM produces a *probabilistic* answer, and that answer can look convincing while hiding hallucinated APIs, security flaws, or subtle logic mistakes [107]. The only practical way to know whether a model is trustworthy – and whether one model is better than another – is to test it against benchmark datasets [37].

The trouble is that the benchmark ecosystem itself has grown explosively and unevenly. Datasets differ widely in which tasks they cover, where their data comes from, how carefully that data was checked, what they measure, and how much they resemble the software that developers actually write. On top of that, model capability has been improving faster than the benchmarks meant to measure it, so many datasets that were challenging in 2021 are saturated today. Without a systematic map of this landscape, it is hard to say what a reported score actually means, and harder still to design the next generation of evaluations.

Study I builds that map. It reviews 156 unique benchmark datasets, drawn from 142 primary studies and spanning 32 distinct coding tasks, and asks five questions of the corpus, RQ1 to RQ5 of the thesis:

- **RQ1 – Landscape.** When and where were these benchmarks published, which tasks and programming languages do they target, how large are they, and which models are they used to evaluate?
- **RQ2 – Construction and quality.** Where does the underlying data come from, how are the datasets organised, what quality controls were applied during construction, and how do the datasets justify their claim to real-world relevance – including what, if anything, they do about data contamination?

- **RQ3 – Evaluation.** Which metrics and evaluation pipelines are used, and how consistent are they across datasets?
- **RQ4 – Limitations.** What weaknesses have researchers who *use* these benchmarks actually reported about them?
- **RQ5 – The way forward.** Given those weaknesses, what should the next generation of benchmarks look like?

Two design decisions distinguish this survey from earlier ones (compared in detail in Chapter 2). First, it is deliberately restricted to the LLM era: only benchmarks created after early 2020, with the explicit purpose of evaluating LLMs, are included, so that the analysis reflects how the community evaluates generative models rather than mixing in pre-LLM legacy datasets. Second, the limitations reported in RQ4 are not our own judgements – they are extracted from the published papers of researchers who applied these benchmarks in practice, which grounds the criticism in community evidence. The list of surveyed papers and all extraction scripts are publicly available[1].

3.1.2 Survey Methodology

The survey follows the established systematic-review protocols of Keele et al. [108] and Petersen et al. [109]. Figure 2 summarises the pipeline; each stage is described in turn below.

Scope and Sources The corpus covers work published between January 2020 and June 2025. The starting point is not arbitrary: GPT-2 and GPT-3 arrived around 2020, and with them the first datasets built specifically to test generative models on code. The window since then captures both the surge in LLM adoption for coding tasks and the parallel maturation of evaluation methodology, which makes it the right period for studying how data-driven evaluation has evolved. Candidate papers were retrieved from four databases – the ACM Digital Library, IEEE Xplore, ScienceDirect, and Google Scholar.

Search Strategy The automated search combined three groups of keywords – benchmarking terms, LLM terms, and coding-task terms:

> (Benchmark Dataset **OR** Benchmarking) **AND** (LLM **OR** Large Language Model) **AND** (Code Intelligence **OR** Coding Task **OR** Code Generation **OR** Code Summarization **OR** Test Cases Generation **OR** Patch Generation **OR** Code Optimization **OR** Code Translation **OR** Program Repair **OR** Requirement Generation **OR** Software Development **OR** Software Engineering **OR** Code Review **OR** Code Analysis **OR** Code Completion **OR** Code Understanding)

[1] https://github.com/mohammad-abdollahi/survey_llm_benchmark_dataset

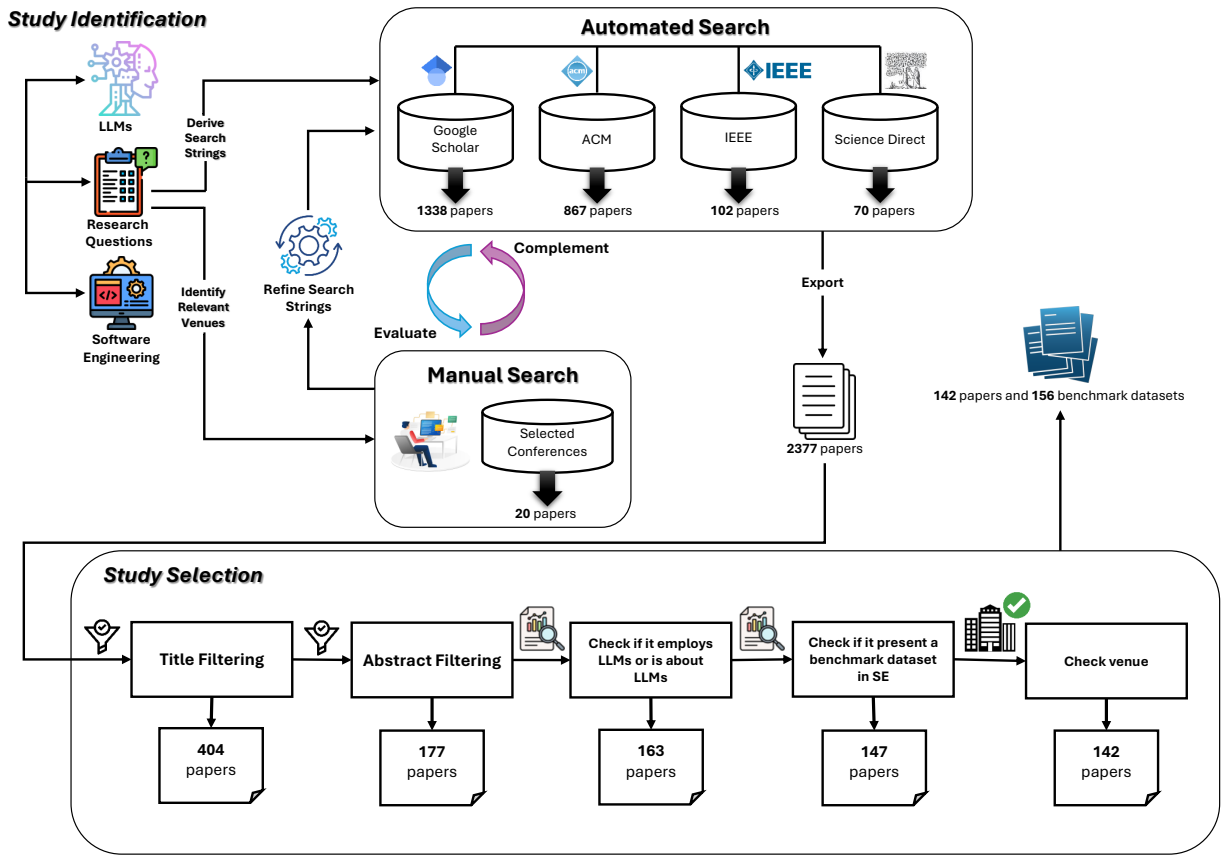


Figure 2: How the survey corpus was assembled: automated searches over four databases, three manual filtering stages, and a quality checklist, complemented by venue-based manual search.

Keyword search alone misses work that uses different terminology, so the initial results were expanded by snowballing in both directions: the reference lists of retrieved studies (backward) and the papers citing them (forward) were inspected, and any new terminology discovered along the way was folded back into the search string. This iterative loop continued until it stopped surfacing new candidates.

Filtering the Candidates The raw search returned 2,377 records, many of them irrelevant hits on incidental keyword matches, plus duplicates indexed by multiple databases. Filtering proceeded in three manual passes.

Pass 1 – titles. Each title was checked for genuine relevance to LLM benchmarking in code intelligence. Superficially matching but out-of-scope work was discarded here – a paper titled "Benchmarking Large Language Models for Automated Verilog RTL Code Generation", for example, contains every keyword yet belongs to hardware design. This pass kept 404 unique studies.

Pass 2 – abstracts. Each surviving abstract was decomposed into its four conventional components – context, objective, approach, and findings – and judged against the survey's objectives component by component. This reduced the pool to 177 studies.

Pass 3 – full text. Every remaining paper was read in full and scored against a 13-item quality checklist probing, among other things: whether the paper states a clear benchmark-construction objective; whether LLMs are involved in building or evaluating the dataset; whether the dataset generalises across models; whether its construction, validation, and quality-assurance steps are documented; whether it is realistic and representative; and whether its metrics, pipelines, and reported results are sound. Papers meeting at least 75% of the criteria were retained. The surviving corpus comprises 142 primary studies introducing 156 unique benchmark datasets.

To keep the latest developments in scope, recent arXiv preprints (late 2023 onward) were eligible, but only if they passed the same checklist – with particular weight on validation methodology, data integrity, and evaluation soundness. Excluded outright were workshop-only papers, surveys, books and tutorials, work outside software engineering, and studies of traditional NLP techniques without LLMs.

Reliability of the Selection All three passes were performed independently by two researchers, whose decisions agreed 95% of the time (Cohen's $\kappa = 0.89$). The residual disagreements went to a third researcher for arbitration. One deliberate exception to the inclusion rules deserves mention: five benchmarks – ConDefects [110], GitHub-Recent-Bugs [111], BinPool [112], ICVul [113], and VulZoo [114] – report no LLM experiments in their original papers, yet were kept in the corpus. The first two exist precisely to defeat data leakage, curating samples created after the training cut-offs of major models; the other three fill acute gaps in binary analysis and vulnerability detection, where trustworthy benchmarks are rare. Excluding them would have removed exactly the resources that address the validity threats this survey cares about.

3.2 Study II: White-Box Analysis of Reasoning on Code-Execution Inference

Study I ends on a pointed observation: every benchmark in the surveyed landscape grades the model's final answer, and none can see the reasoning that produced it. Study II acts on that observation. It puts four state-of-the-art *reasoning* LLMs to work on a task where reasoning is the whole game – predicting what a program will do when executed – and then, instead of stopping at accuracy numbers, reads the models' reasoning traces line by line to find out where, why, and how systematically they go wrong. The study was published at ACM TOSEM [4]; this section presents its design, and Section 4.2 its findings.

3.2.1 Motivation and Research Questions

Knowing what a program does at runtime – which branches fire, how variables evolve, what finally gets returned – underpins nearly everything we want LLMs to do with code: generating it correctly, debugging it, and reasoning about it automatically. The newest generation of LLMs seems purpose-built for this. Reasoning-specialised models emit an explicit chain of intermediate steps before answering [53], which invites an appealing assumption: that a model which "shows its work" is actually simulating the program's semantics as it goes [29].



Figure 3: A reasoning failure in the wild: (a) the program under analysis, its input, and the true output; (b) the model's step-by-step trace, with the first semantically wrong step highlighted; (c) the prompt that elicited the trace. The single faulty step – a misevaluation of Python's `split()` – determines the incorrect final answer.

Whether that assumption holds is an open question, because the existing evidence does not test it. Prior evaluations of code reasoning [56, 30, 63] score models on whether their predicted outputs match ground truth – valuable baselines, but fundamentally black-box measurements. When a model gets an execution question wrong, output-level scoring cannot say whether the breakdown happened in variable tracking, in control flow, in arithmetic, or somewhere else entirely. The structure of reasoning failure is invisible.

Figure 3 makes the problem concrete. Claude 4 Sonnet is asked to predict a small program's output and walks through the execution step by step – yet stumbles on the semantics of Python's `split()` on strings. Everything before the highlighted step is sound; everything after is built on the mistake. One faulty step, one wrong answer. Whether such failures are typical, where they concentrate, and what kinds exist – these are empirical questions no prior study had answered.

Two motivations drive the study. The first is scientific: reasoning LLMs deserve to be understood as execution reasoners in their own right [56, 30], not least because there are settings – reasoning over a partial diff, over code that will not run – where no external tool can substitute for the model's internal simulation. The second is constructive: any hybrid neuro-symbolic system in which an LLM orchestrates external tools depends on the model holding a sound semantic hypothesis about the code, and knowing exactly how that

hypothesis fails is a prerequisite for designing the hybrid well – a point the tool-augmented experiment of Section 4.2.4 demonstrates directly.

Concretely, Study II asks three questions, RQ6 to RQ8 of the thesis:

- **RQ6 (Performance):** How accurately do reasoning LLMs infer execution outcomes, and how does accuracy shift across ordinary, boundary, and invalid inputs?
- **RQ7 (Failure Location):** At which kinds of program statements does the reasoning first go wrong?
- **RQ8 (Failure Pattern):** What recurring semantic patterns do the failures follow, and can they be organised into a taxonomy?

Along the way, the study contributes: the first large-scale comparative analysis of four frontier reasoning LLMs on execution inference across input types and complexity levels; a dual-level annotation of every failed trace (the earliest diverging statement, plus a semantic pattern label); a nine-class error taxonomy under four parent categories; and the first application of tool-augmented reasoning to software-engineering execution tasks, which repairs 69% of the dominant error class. All data and code are publicly released.

3.2.2 What Counts as Reasoning in This Study

The word "reasoning" is used loosely in the literature, so it needs a working definition before it can be measured. In this study, reasoning refers to the explicit, step-by-step account of program execution that a reasoning-specialised model produces in its trace before stating an answer. That account has one job: to simulate the interpreter. At each step the trace must hold the current program state, follow control flow to select the next statement, evaluate that statement's effect on the state (an arithmetic result, an index, the return value of a library call), and carry the result forward, verifying it where the trace claims to. A trace is correct when this account matches the actual execution at every step, and a reasoning error is any point at which it departs from it.

This definition is deliberately broader than reasoning in the sense of logical inference, and the distinction matters for reading the taxonomy of Section 4.2.3. Misevaluating an arithmetic expression is not a failure of logic; it is a failure of computation. But in execution inference the computation is part of the reasoning process: the trace has undertaken to compute that value, and a wrong value at one step corrupts every step that depends on it, exactly as a wrong inference would. The taxonomy therefore covers the full range of ways a trace can depart from execution, and its parent categories separate them by kind. Categories A to C (program semantics and state simulation, execution path and flow tracking, input/output disconnects) are mechanical failures of simulation: the model attempts the right step and gets it wrong. Category D (hallucination and lack of verification) contains reasoning failures in the narrow sense: the model does not attempt the step, fabricates a result, or accepts a claim it never checked. Where this thesis says "reasoning error" without qualification, it means an error in the reasoning trace in the broad sense; where it means the narrow sense, it says so.

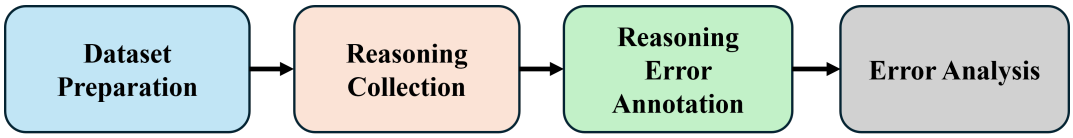


Figure 4: The four-step design of Study II: benchmark and input preparation, reasoning-trace collection, error annotation against executed ground truth, and taxonomy construction.

3.2.3 Study Design

The study proceeds in four steps (Figure 4): assemble a benchmark of programs and inputs (Section 3.2.3); collect reasoning traces from four LLMs for every program–input pair (Section 3.2.3); execute the programs to obtain ground truth and annotate every failed trace (Section 3.2.3); and distil the annotations into an error taxonomy (Section 3.2.3).

Programs and Inputs

Assembling the program corpus. The corpus draws on three established benchmarks chosen to span a difficulty gradient. From HumanEval+ [115] we take all 164 tasks – well-understood, comparatively simple functions. From LiveCodeBench [31] we keep only the 263 problems rated medium or hard, discarding the easy tier whose one-line manipulations would tell us little about multi-step reasoning. And to reach beyond isolated functions, we add the 50 most complex problems of REPOEXEC [27], identified by ranking the dataset on Halstead and Cyclomatic complexity – realistic, multi-file code that stresses execution reasoning at repository scale. Each benchmark's reference solutions serve as the programs under analysis. Table 2 summarises the corpus: by every complexity measure [116, 117], REPOEXEC programs are the heaviest, LiveCodeBench the hardest at function level, and HumanEval+ the gentlest.

Table 2: Overview of the datasets. **CC** denotes the Cyclomatic Complexity metric, and **HM** is the Halstead Metrics.

Dataset	#Task	Avg. #LOC	Avg. CC	Avg. HM
HumanEval+	164	8	3.1	2.2
LiveCodeBench	263	12	5	3
REPOEXEC	50	152	8.6	16

Designing the inputs. A program is only half of an execution question; the input is the other half, and input choice controls which reasoning skills get exercised. Each program is therefore paired with twelve inputs across three deliberately different tiers:

- **Regular inputs** (five per program) come from the benchmarks' official test cases and represent well-formed, in-spec usage. Selection follows Input Space Partitioning [118]: the input domain is divided into disjoint, jointly exhaustive blocks over the relevant parameters, and representatives are drawn

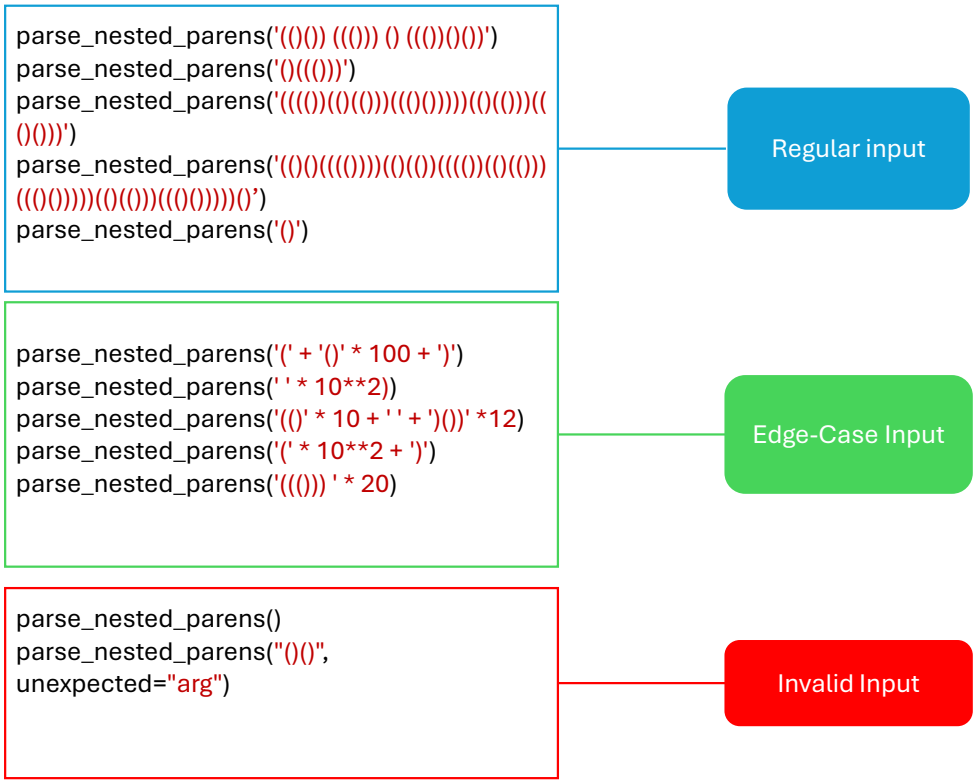


Figure 5: Regular, edge-case, and invalid inputs for the running example of Figure 3.

from every block, so the five inputs collectively cover the valid space rather than clustering in one corner of it.

- **Edge-case inputs** (five per program) sit at the boundaries of validity: parameters pushed to their minima and maxima, zeros, empty strings, single-element collections, extreme integers. They are legal inputs – but the kind that break sloppy reasoning.
- **Invalid inputs** (two per program) deliberately break the contract: wrong types, missing or surplus arguments, out-of-range values, unexpected keywords, division by zero, forbidden `None`s, malformed structures. Every one was run against the reference implementation to confirm it genuinely raises an exception. The correct behaviour for the model is not to predict an output at all, but to recognise the input as invalid.

Why three tiers rather than one well-designed set? Because different input classes provoke qualitatively different failures. Regular inputs check whether the model can simulate ordinary execution at all; edge cases stress branch logic and state tracking until latent control-flow defects and input misreads surface; invalid inputs probe something else again – whether the model notices that the contract is broken instead of confidently hallucinating an answer. Any single tier would yield a partial, biased picture of reasoning ability. Figure 5 shows all three tiers instantiated for the running example.

The arithmetic of the design: 477 programs $\times$ 12 inputs $=$ 5,724 inference tasks per model, and 22,896

collected inferences over the four models.

Collecting the Reasoning Traces

Which models, and why these. The study evaluates DeepSeek-R1 [53], OpenAI o4-mini [52], Gemini 2.5 Flash [54], and Claude 4 Sonnet [55] – the reasoning-specialised flagships of four leading providers at the time of the study, each with strong published results on mathematics, code, and multi-step inference, and together spanning meaningfully different architectures and training recipes. The contrast with earlier work matters: prior evaluations [30] largely tested small or general-purpose models (CodeLlama 7B–34B, StarCoder2, Mistral, GPT-3.5) that were never optimised for extended reasoning. Study II deliberately restricts itself to models that were. And because these models reason in multiple steps natively, no Chain-of-Thought prompting was layered on top – consistent with standard practice for this model class [30, 63].

Eliciting the traces. For every program–input pair, each model is prompted to reason its way to the execution result explicitly, not merely to state an answer. The prompt further directs the model to attend to control flow – branching, loops, execution order – and to data flow – variable dependencies and state transitions – anchoring the trace in program semantics so that it can later be compared, step by step, against real execution. For invalid inputs, the expected behaviour is detection, not output prediction. The exact prompts ship with the replication package.

Annotating the Failures Running every program on every input yields ground truth; comparing it against the models' predictions isolates the failures. Because a trace's final answer is the product of its intermediate steps, a wrong answer implies something went wrong *inside* the trace – and finding that something is the annotation task. Three authors independently labelled every failed trace with two coordinates that capture different granularities:

- **Where it first went wrong – the inspection point.** Writing $\sigma_0, \sigma_1, \ldots, \sigma_T$ for the true execution states of program P on input x, and $\hat{\sigma}_0, \hat{\sigma}_1, \ldots$ for the states the trace implicitly describes, the inspection point is the first statement s_k whose described effect on $\hat{\sigma}_{k-1}$ departs from its true effect on σ_{k-1}. It pins the failure to one static code location – an assignment, a branch condition – and marks the cheapest place for a human (or a repair mechanism) to start looking.
- **How it went wrong – the pattern label.** Location alone does not explain a failure, so each trace also receives a semantic label for its dynamic failure pattern – "Iteration Window Miscalculation", say, or "Hallucinated Runtime Error". Unlike the inspection point, the evidence for a pattern label has no fixed extent: some patterns are visible in one step, others only across a stretch of the trace.

The two coordinates are complementary, not redundant: every failure has a first point of divergence *and* a semantic character, and recording both is what lets the study answer RQ7 (where) and RQ8 (how) from

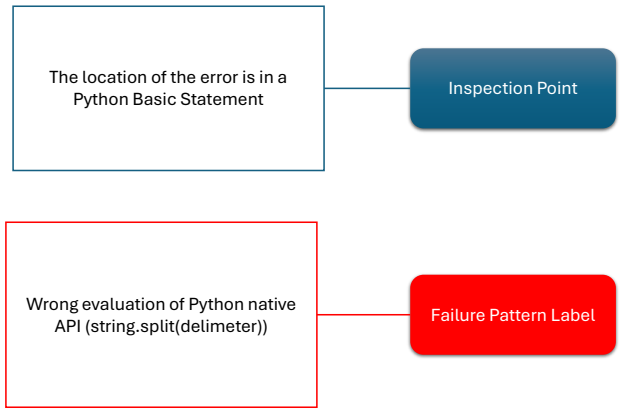


Figure 6: Dual-level annotation of the failure from Figure 3: a single-statement inspection point plus a semantic failure-pattern label.

the same annotation. Independent labels were reconciled in review sessions with three additional authors, disagreements resolved by consensus. Pre-consensus reliability was strong – 85.06% raw agreement, Fleiss' $\kappa = 0.875$. For the running example, Figure 6 shows the outcome: inspection point at a basic method-call statement, pattern label reflecting the botched evaluation of `split`.

From Annotations to a Taxonomy The final step organises the pattern labels into a taxonomy. The method was open card sorting: every failed trace became a card; annotators grouped cards by shared failure characteristics into candidate categories; and all six authors then debated, merged, and split the candidates until a stable structure emerged. Consensus-driven refinement served two purposes – damping individual bias, and forcing each category to earn its place by covering a genuinely recurring phenomenon. The resulting taxonomy (presented under RQ8) doubles as the coding scheme for the quantitative analyses that follow.

3.3 Study III: Adaptive Test Selection for Reinforcement Learning on Issue Resolution

The first two studies were diagnostic. Study I established how the community measures LLMs on coding tasks and where those measurements fail; Study II opened the reasoning traces behind the failures and showed that they are localisable – a trace usually goes wrong at one identifiable step, and repairing that step usually repairs the outcome. Study III is the constructive counterpart: it takes that insight to the hardest end-to-end coding task we have, repository-scale issue resolution, and uses it to change how a model is trained.

The connection is direct. Study II's finding is, in training terms, a statement about *credit assignment*: when the thing that determines success is a single localisable sub-step, a training signal that scores only the final outcome throws away almost everything the rollout revealed. Reinforcement learning for issue resolution has exactly this shape, and exactly this defect. Every published recipe rewards a patch with one bit – did it fix all the targeted tests without breaking anything – and a rollout of a hundred agent actions collapses into

that bit. Study III replaces the bit with a graded, test-level signal, and then goes one step further: rather than accepting the test suite as a fixed oracle, it lets the training loop *choose* which tests define the reward and *repair* the tests that are defective.

Study III addresses the last four research questions of the thesis. *RQ9* asks how much usable learning signal the binary verifier reward discards; the analysis of Section 3.3.1 and the baseline's training curve answer it. *RQ10* asks whether a dense, test-level reward with in-loop test selection improves sample efficiency over that baseline, and *RQ11* asks whether defective tests can be repaired during training without injecting false negatives, and whether refinement measurably improves test quality; the five-condition comparison of Section 4.3 answers both. *RQ12* asks whether the resulting policy transfers to a held-out benchmark scored on original tests, and at what token cost, which the held-out evaluation answers.

3.3.1 Problem Formulation

Issue Resolution as a Reinforcement Learning Task The task is issue resolution: an agent is placed in a snapshot of a real software repository containing a defect, is given a natural-language description of that defect, and must produce a patch that repairs it. A training instance is a tuple $x_i = (\mathcal{R}_i, p_i, \mathcal{F}_i, \mathcal{P}_i)$, where $\mathcal{R}_i$ is a repository snapshot at a base commit, p_i is the issue text, $\mathcal{F}_i$ is the set of *fail-to-pass* tests that the defect currently causes to fail, and $\mathcal{P}_i$ is the set of *pass-to-pass* tests that already pass and must continue to do so.

A rollout τ is a sequence of agent actions $a_1, \ldots, a_T$ – file reads, edits, shell invocations, and a terminating submission – executed against $\mathcal{R}_i$ inside an isolated container. The rollout induces a patch $\pi(\tau)$, and an execution-based verifier returns a pass map $V_\tau : \mathcal{F}_i \cup \mathcal{P}_i \to \{0,1\}$ obtained by running the repository's test suite against the patched snapshot.

Essentially every published recipe in this setting rewards the conjunction of the two test conditions:

$$R_{\text{bin}}(\tau) \;=\; \prod_{t \in \mathcal{F}_i} V_\tau(t) \;\cdot\; \prod_{t \in \mathcal{P}_i} V_\tau(t) \;\in\; \{0,1\}. \tag{3}$$

A patch is rewarded if and only if it repairs every targeted test while breaking no regression test. The consequence is stark: a patch that resolves $|\mathcal{F}_i| - 1$ of $|\mathcal{F}_i|$ tests receives precisely the reward of a patch that resolves none.

Two Pathologies of the Binary Verifier Reward Equation (3) is responsible for two distinct failure modes, and the two mechanisms proposed in this study are answers to them.

Sparse and degenerate credit assignment. A rollout of 50–150 agent steps collapses to a single bit. Under group-baselined policy-gradient estimators such as GRPO, a group of rollouts whose members all receive the same verdict contributes exactly zero gradient – an effect analysed formally in Section 3.3.1. In issue resolution, agreement within a group is the common case rather than the exception: early in training almost

every rollout fails, and later a growing fraction of instances are uniformly solved. Both regimes consume full agentic rollouts – by a wide margin the most expensive operation in the pipeline – and return no learning signal whatsoever.

Imperfection of the verifier itself. The test suite is not the objective; it is a noisy estimator of the objective. It admits *false positives* – incorrect patches that a too-weak suite accepts – and *false negatives* – correct patches that the suite rejects because individual tests are over-specific or incidental to the issue. Both directions occur at substantial rates in precisely this domain: an audit of code reinforcement-learning environments reports that 28.5% of a 49-task SWE-bench Verified sample has test suites weak enough that a container-verified *incorrect* patch passes, with a comparable 25.0% figure on an R2E-Gym sample [75]. The resulting reward noise is not the benign kind. It is *persistent and per-instance* rather than random: the same wrong patch is accepted every time the instance is sampled, so the error never averages out over training.

The Proposal, and What It Is Not Standard practice fixes the verifier suite at dataset-construction time and holds it constant for the duration of training. This study instead treats the reward-defining suite as an object the training loop may adapt, and defines a policy-conditioned, epoch-indexed reward

$$R_t(\tau_i) \;=\; f(\pi(\tau_i),\, \mathcal{S}_t(\mathcal{T}_i \cup \mathcal{G}_t(i)))\,, \tag{4}$$

where $\mathcal{S}_t$ is a **selection** operator over the available tests and $\mathcal{G}_t$ is a **refinement** operator that produces new ones, both recomputed at training time t conditioned on the current policy π_{θ_t}. Two things change during training: *which* tests count, and *what* tests exist. The instance set, the agent scaffold, and the optimisation algorithm all remain fixed.

One framing point deserves emphasis, because the rest of the argument depends on it. This is **not reward shaping**. It is **verifier-error reduction against a fixed latent objective**. The true objective is the unknown correctness oracle $R^\star$ – whether a patch genuinely resolves the issue without regression – and the test suite is merely an estimator $\widehat{R}$ of it. The two operators reduce $\|\widehat{R}_t - R^\star\|$ over the course of training. The objective never moves; only the estimator of it improves. Read this way, the standard objection to time-varying rewards – that the optimum becomes ill-defined – simply does not apply.

Advantage Collapse under Binary Rewards The GRPO estimator of Section 2.4.4 makes the existence of a gradient contingent on within-group reward variance, and the binary reward interacts with that dependence in the worst possible way. Consider a group in which the reward is binary, and let $\hat{p}$ denote the empirical fraction of successful rollouts. The variance of the group reward is $\hat{p}(1-\hat{p})$, maximal at $\hat{p} = 1/2$ and vanishing at both extremes. When all G rollouts agree – $\hat{p} \in \{0,1\}$ – every advantage in Equation (1) of Chapter 2 is identically zero, the surrogate gradient vanishes, and the group contributes nothing to the update despite having consumed G complete agentic rollouts.

In issue resolution this is not a corner case but the dominant regime, at both ends of training. An implementation detail of the training frameworks in common use compounds it: the trainer applies a variance filter that excludes groups satisfying $(\mathbf{R} = 0).\text{all}()$ or $(\mathbf{R} = 1).\text{all}()$ from the update. Keyed on exact equality to the two extreme values, this predicate is correct for binary rewards, but it has the consequence that a binary recipe can *never* obtain gradient on the hardest instances, where all rollouts fail, nor on the easiest, where all succeed. In our own baseline runs this filter triggers on approximately 40% of optimiser steps.

Any reward that is continuous rather than binary escapes the failure mode structurally. Group members differ in reward even when they agree in verdict, so $\sigma_G > 0$, advantages are non-zero, and no group is discarded. This observation motivates the form of reward adopted in Section 3.3.2.

3.3.2 Method

Both operators are implemented as modifications to the reward and to the test set over which it is computed, leaving the optimiser of Section 2.4.4 and the agent scaffold untouched.

One point of scope should be fixed before the details, because it determines where in the pipeline the two operators sit. **Neither operator is visible to the agent, which observes only its reward.** The tests are never rendered into the prompt and play no part in the rollout: the agent explores the repository and produces a patch exactly as it does under the binary recipe, and nothing about the acceptance criterion is disclosed to it. The tests enter only afterwards, during *validation* – the phase in which completed rollouts are executed against the verifier to produce their rewards – and both selection and refinement run at the *end of each RL step*, once that step's rollouts have been scored, updating the state that the next step's validation will use. The unit of adaptation is therefore the optimiser step, not the agent step, and the entire mechanism lives on the reward side of the loop.

This has a useful consequence for interpreting the results. Because the agent's observation space is unchanged, any difference between the proposed method and the binary baseline in Section 4.3 is attributable to the reward signal alone, and not to the policy having been handed extra information about what it is being graded on.

Test Selection The selection operator maintains, for each instance, a partition of $\mathcal{F}_i$ into three states, updated at the end of every RL step in which that instance was sampled (Table 3). We write $\mathcal{A}_i^{(t)} \subseteq \mathcal{F}_i$ for the *active* set at step t: the tests executed during validation and scored by the dense reward term.

The active set is bounded below by m_{init} and above by $m_{\max}$. Each active test carries a stall counter, and when a test remains unsolved for k consecutive RL steps the scheduler promotes additional tests from the pool – an operation we call *stall-expansion*.

Three properties of this construction are worth making explicit. The cooldown-and-re-verify cycle closes a *transient-pass* loophole, in which a test passes once as an incidental side effect of an unrelated edit and is then banked permanently. Stall-expansion is the mechanism by which the curriculum responds to the

Table 3: States of the test-selection operator. Tests move between states at the end of each RL step, conditioned on how that step's rollouts performed on the instance. No state is visible to the agent; the partition governs validation and scoring only.

State	Role in validation	Transition at end of RL step
Active	Executed during validation and scored by the dense reward term.	On being passed, moves to cooldown. On remaining unsolved for k consecutive steps, triggers stall-expansion; when a refinement wave fires, stalled active tests are submitted for refinement.
Pool	Held back: neither executed nor scored.	Promoted to active by stall-expansion.
Cooldown	Suspended from scoring for a fixed number of steps, then re-executed.	Returned to the pool if still passing; returned to active if it has regressed.

policy rather than following a fixed schedule: a stalled instance has its acceptance criterion broadened until it discriminates again. And the bounds m_{init} and $m_{\max}$ keep the criterion in a useful range on instances with large $|\mathcal{F}_i|$ – the lower bound stops the reward from degenerating into a single-test verdict, while the upper bound stops repeated stall-expansion from collapsing back into the full-suite conjunction of Equation (3), and caps the validation cost per rollout.

The contrast with conventional curriculum learning is worth stating plainly: conventional curricula order *instances* by difficulty, admitting or deferring whole problems, whereas this one adapts the acceptance criterion *within* an instance, over the verifier sub-goals that instance provides. The task never changes; only the strictness of the verdict against which it is scored does.

Test Refinement Refinement is applied in *waves*, on the same end-of-RL-step schedule as selection. A trigger rule, evaluated on the training reward at the end of each RL step, decides when a wave fires; two such rules are compared in Section 3.3.3. When a wave fires, every active fail-to-pass test that has remained unsolved for a threshold number of consecutive steps is submitted to a refiner, which either rewrites it as a single cleaner test or splits it into several smaller tests, each asserting one facet of the intended behaviour. The refiner receives the issue text, the gold patch, the original test, and a selection of the policy's own failed trajectories on this instance, and it operates as an agent with read and execution access to a checkout of the buggy repository, so that it can run candidate tests and iterate.

Diversifying the evidence. Trajectories drawn from a single rollout group are frequently near-duplicates, which would give the refiner a distorted view of how the policy fails. A diversity filter is therefore applied before the refiner prompt is constructed. Each trajectory is normalised to a canonical action sequence by regular-expression removal of volatile tokens – addresses, line numbers, temporary paths, timestamps, durations – and pairwise similarity is computed on the normalised strings. Trajectories are then selected greedily, admitting a candidate only if its maximum similarity to the already-selected set falls below a threshold, so that the refiner observes materially distinct failure modes rather than repetitions of one.

The validation gate. Every candidate test is validated in clean containers, independently of the refiner's own observations, against two conditions: it must **fail on the unpatched repository** with a genuine assertion failure, and it must **pass once the gold patch is applied**. The first condition excludes tests that pass trivially; insisting on a genuine assertion failure rather than merely a non-zero exit status additionally excludes candidates that "fail" only because they do not import or collect – a failure mode a naive gate admits. The second condition excludes tests that assert behaviour the intended fix does not produce, which would inject false negatives directly into the reward. Candidates surviving both conditions replace the stalled test in the active set and in $\mathcal{F}_i$, and take effect from the next RL step's validation; candidates failing either are discarded and the original test is retained.

This gate is the component that makes refinement safe rather than dangerous. Without it, a generative refiner injects false negatives faster than it removes false positives – and since false negatives are reported to degrade performance monotonically faster than false positives, a gap that additional compute does not close [76], an ungated refiner would make training strictly worse than the baseline.

The Reward Function Let $\mathcal{F}_\tau^{\text{solved}} \subseteq \mathcal{F}_i$ denote the fail-to-pass tests passing at termination, let $\mathcal{A}_i^{(t)}$ be the active set of Section 3.3.2 – the tests actually executed during that step's validation – and let $\mathcal{P}_\tau^{\text{broken}} \subseteq \mathcal{P}_i$ denote the regression tests broken by the patch. The terminal reward is

$$R_{\text{ATS}}(\tau) = \underbrace{w_s \frac{|\mathcal{F}_\tau^{\text{solved}} \cap \mathcal{A}_i^{(t)}|}{|\mathcal{A}_i^{(t)}|}}_{\text{partial credit}} - \underbrace{w_r \frac{|\mathcal{P}_\tau^{\text{broken}}|}{|\mathcal{P}_i|}}_{\text{regression guard}} - \underbrace{\lambda \min\left(\frac{n_{\text{tok}}(\tau)}{T_{\text{ref}}}, c\right)}_{\text{token penalty}}. \tag{5}$$

The first term answers the credit-assignment pathology of Section 3.3.1 directly; the other two restore properties that Equation (3) enforced implicitly or not at all.

Partial credit replaces the product of Equation (3) with a normalised count over the tests validated at that step, rendering the reward continuous on $[0, w_s]$ and thereby restoring non-zero group variance in exactly the regime analysed in Section 3.3.1. This term is where the selection operator acts: $\mathcal{A}_i^{(t)}$ is what the scheduler of Section 3.3.2 widens or narrows, so selection is a direct actuator on the strictness of the criterion and hence on the per-instance pass rate.

Two safeguards keep this from becoming a route to easy rewards. First, the reported *solve rate* – the quantity compared against the baseline in Section 4.3.2 – is never the reward: it is computed against the complete, unmodified fail-to-pass set of the instance, so a policy that satisfies a narrowed active set without resolving the issue gains reward but not credit in the metric. Second, stall-expansion tightens the criterion precisely when the policy stops making progress, so the active set ratchets toward, not away from, the full conjunction. Held-out evaluation (Section 4.3.3) closes the argument by scoring exclusively against original, never-refined suites.

The regression guard was absent from our initial formulation. We report this defect because its consequence is instructive. Without the guard, the objective rewards resolving $\mathcal{F}_i$ irrespective of collateral damage, and

the policy exploits exactly that: in an ablation in which the regression weight is set to zero while pass-to-pass tests continue to be executed and logged, patches that satisfy the targeted tests at the cost of breaking regression tests are scored identically to clean fixes, and the policy is under no pressure to avoid them. Reinstating the term restores the second half of the specification that Equation (3) enforced by construction.

The token penalty makes trajectory length an explicit objective rather than a passive cost – the efficiency dimension that Study I's recommended evaluation suite (Section 4.1.3) argued should sit alongside correctness. The clipping constant c is not a tuning parameter but a safety measure: an unbounded penalty is an outlier generator in an agentic loop, as we discovered when a single trajectory emitting roughly 3×10^8 characters of shell output produced a reward of -495 and a corresponding gradient-norm spike. Capping the ratio bounds the term's contribution while preserving its ordering over the range that matters.

Why GRPO Tolerates a Moving Reward A methodological point in favour of this algorithm family deserves note. Modifying the reward during training ordinarily violates stationarity, and the standard casualty is the learned value function: a critic fitted under R_{t-1} produces systematically biased advantages under R_t, inducing a transient after every change. GRPO has no learned critic. Its baseline is recomputed from scratch at every step under whichever reward is currently in force, so there is no stale value estimate to invalidate, and the non-stationarity cost that would afflict actor-critic methods is structurally absent. Two residual effects remain and should be acknowledged rather than dismissed: the KL term anchors to a reference policy trained under earlier rewards, and any reuse of stored experience carries reward labels computed under an earlier suite.

We claim no policy-invariance guarantee. The classical result of Ng et al. [119] and its extension to time-varying potentials [120] apply to potential-based shaping, which adds a telescoping term to the reward; the present method modifies the terminal reward itself, so neither result is applicable. The appropriate theoretical framing is the one adopted in Section 3.3.1: the verifier is a stochastic reward channel with per-instance asymmetric false-positive and false-negative rates, and the two operators reduce those rates at the source. This is complementary to, rather than competitive with, methods that debias the gradient given estimated noise rates.

3.3.3 Experimental Setup

Training Sources Training instances are drawn from two public corpora of executable issue-resolution environments.

SWE-rebench [87] is a large-scale, continuously updated collection of real GitHub issues with verified execution environments, comprising 21,336 instances, of which 6,542 carry a publicly resolvable container image. Beyond the standard fields it carries per-instance automated annotations, obtained by applying an LLM to the human-annotation rubric used to construct SWE-bench Verified: a difficulty score, an issue-specification-quality score, and a test-validity score, each on a four-point scale, together with the number of files modified by the gold patch. These annotations feed directly into the instance-selection procedure below.

SWE-Gym [85] provides 2,438 instances across 11 repositories with the same schema but without quality annotations. Its value in the mixture is distributional rather than volumetric: it supplies a second, independently constructed environment-generation pipeline, guarding against the possibility that any measured effect is an artefact of a single corpus's construction methodology.

Evaluation Benchmark Evaluation is performed on SWE-bench Verified [83], the 500-instance human-validated subset of SWE-bench [28] spanning 12 repositories. Because contamination between training and evaluation would invalidate every downstream claim, those 12 repositories are excluded from the training pool *at the repository level* rather than at the instance level.

Evaluation runs exclusively against the original, never-refined test suites. This is a structural requirement, not a stylistic preference. The most damaging available objection to a method that modifies its own reward is that the measured gains are artefacts of easier tests. Freezing the evaluation suite – and reporting training reward and evaluation reward as separate quantities that are never conflated – is what makes the comparison meaningful.

Constructing the Training Set From the union of the two corpora we construct an 800-instance training set, admitting instances subject to two families of constraints.

The first family is required for correctness. Every repository appearing in SWE-bench Verified is excluded. Each instance must have at least one fail-to-pass test and at least one pass-to-pass test – the latter because the regression term of Equation (5) is otherwise undefined. Each must carry a non-empty gold patch, since the refinement operator's validation gate applies it. And its container image must be resolvable and locally cached at build time, which eliminates rollout failures caused by registry rate limits.

The second family targets instance quality and exploits the SWE-rebench annotations. We retain instances whose issue-specification and test-validity scores both indicate a well-specified issue and a valid suite; impose a controlled difficulty mixture of 25% easy, 50% medium, and 25% hard; and require that the gold patch touch at most three files and that the issue statement fall between 200 and 10,000 characters, so that it fits the prompt budget. Repository diversity is enforced by admitting a single instance per repository from SWE-rebench and capping SWE-Gym at ten instances per repository for a quota of 100 instances.

Table 4 summarises the result. The correspondence in the last two rows matters more than it might appear: because the training and evaluation distributions agree closely in test granularity – a mean of 3.08 fail-to-pass tests per instance against 3.03, with identical medians of one – the partial-credit reward is operating on comparable structure in both settings, which supports the external validity of the transfer from training distribution to evaluation.

Training Configuration and Experimental Conditions All runs train a 4B-parameter policy, initialised from the SWE-Master supervised checkpoint, for 200 optimiser steps on a cluster of 8 NVIDIA H100 GPUs. Each optimiser step samples a batch of 16 instances and draws a rollout group of $G = 4$ trajectories per in-

Table 4: Composition of the 800-instance training set, and its test-granularity correspondence with the evaluation benchmark.

Property	**Training set**	**SWE-bench Verified**
Instances	800	500
Distinct repositories	706	12
from SWE-rebench	700	—
from SWE-Gym	100	—
Fail-to-pass tests (total)	2,463	—
Pass-to-pass tests (total)	82,807	—
Mean fail-to-pass tests/instance	3.08	3.03
Median fail-to-pass tests/instance	1	1

stance, so one step consumes 64 full agentic rollouts and the 200-step run totals 12,800 rollouts per condition. The implementation is built directly on the SWE-Master codebase [89], and every change is confined to the reward and to the scheduler that maintains each instance's validated test set; the optimiser, the agent scaffold and its prompts, and the verifier execution paths are untouched. The optimiser is GRPO as formulated in Section 2.4.4, with the leave-one-out baseline variant, in which the mean of Equation (1) is computed over the remaining $G-1$ group members, removing the self-contribution bias of the group mean; nothing in the preceding analysis depends on that choice.

To separate the contributions of the dense reward, the selection curriculum, and the two possible placements of refinement – offline before training, or in-loop during it – the evaluation compares five conditions, all trained from the same checkpoint, on the same instances, for the same number of steps:

1. **SWE-Master RL** – the unmodified SWE-Master recipe under the binary reward of Equation (3): the published state of practice, reproducible from our codebase by a single configuration flag.

2. **RL with Test Selection** – the dense reward of Equation (5) with the selection scheduler of Section 3.3.2, operating over the *original* test suites; the refinement operator is disabled.

3. **RL with Predefined Tests** – identical to the previous condition, except that training starts from suites refined *offline*: every test that the refiner improves is replaced before step one, and the suites are then held fixed. This condition isolates what refinement contributes as a dataset-preparation step, in the spirit of offline test-improvement pipelines such as CodeContests-O [99].

4. **RL with Dynamic Test Refinement (reward-band trigger)** – the full method: test selection plus the in-loop refinement operator of Section 3.3.2, with refinement waves fired by a *reward-band* rule. A wave fires when the average training reward has settled into a narrow band around 0.25 – that is, when $|R-0.25|$ stays small across consecutive steps – the signature of a policy that is collecting partial credit on a stable set of tests without resolving them.

5. **RL with Dynamic Test Refinement (reward-trend trigger)** – identical, except that waves are fired by a *reward-trend* rule: the average training reward is monitored over a sliding window of 20 RL steps, and a

wave fires when the reward has been decreasing or flat across that window. Where the band rule keys on the reward's *level*, the trend rule keys on its *derivative*, and so can fire at any reward level once progress stalls.

Two contrasts in this design carry the argument. The first, between the predefined-tests condition and the two dynamic conditions, is the study's central ablation: all three consume the same refiner, but the predefined condition receives its output statically at step zero while the dynamic conditions receive it adaptively, at moments chosen by the state of the current policy. Any separation between them measures the value of *in-loop, policy-conditioned* timing as such, over and above the value of better tests. The second, between the two dynamic conditions, isolates the trigger rule: the same operator, the same reward, and the same selection scheduler, differing only in *when* the training loop decides that the verifier needs repair.

4 Results

This chapter reports the findings of the three studies whose methodologies were described in Chapter 3. Section 4.1 answers Study I's research questions (RQ1 to RQ5) about the benchmarking landscape. Section 4.2 answers Study II's questions (RQ6 to RQ8) about reasoning failures and presents its four follow-up analyses. Section 4.3 answers Study III's questions (RQ9 to RQ12) through its measurements of refinement effectiveness, its five-condition training dynamics, and its held-out evaluation. Each section closes with the finding that connects it to the next.

4.1 Study I: The Benchmarking Landscape

4.1.1 RQ1: What Does the Landscape Look Like?

The first research question profiles the corpus along eight dimensions: time, task, context scope, interaction style, application domain, venue, language, size, and the models being evaluated.

Growth over Time Figure 7 lays out all 156 datasets chronologically by task, and Figure 8 counts them per year. The trajectory is striking: nothing in 2020, five datasets in 2021, twelve in 2022 – and then an inflection. Twenty-four benchmarks appeared in 2023, seventy-five in 2024, and forty in just the first half of 2025; the last eighteen months of the window account for more than half the corpus. The timing is no coincidence. As increasingly capable models became broadly available from 2023 onward, fair and reproducible comparison became a community-wide need, and benchmark construction turned from a niche activity into a central research output of the LLM-for-code field.

Which Tasks Are Benchmarked? Table 5 organises the datasets by coding task and by phase of the software development lifecycle (SDLC), from Requirements & Design through Implementation, Comprehension, Testing, and Maintenance to Security. The distribution is heavily lopsided: code generation alone accounts for roughly 52.8% of all task instances, with code translation a distant second at 7%, followed by completion, test-case generation, and vulnerability detection. At the other extreme, tasks such as formal verification, mobile-agent evaluation, and code optimisation are represented by only a handful of datasets each.

Two structural trends cut across the task distribution. The first is rising ambition: where the early benchmarks posed isolated single-function puzzles [121, 122], newer ones demand class-level generation, repository-wide reasoning, multi-turn debugging, or full agentic workflows – Codev-Bench [123] and CoderEval [124] are representative of the move toward project-scale context. The second is task bundling: 26 of the 156 datasets evaluate more than one task (Table 6), from StackEval's [125] mix of generation, debugging, review, and question answering to CodeXGLUE's [126] fourteen sub-datasets across ten tasks. Multi-task designs make it possible to probe whether a model's strength on one task transfers to its neighbours – something

Figure 7: Chronological overview of LLM benchmark datasets by task.

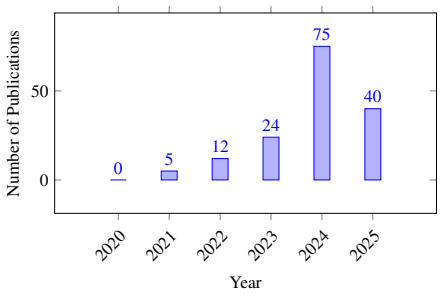


Figure 8: Publication trend of LLM benchmark datasets (2020–2025).

single-task benchmarks cannot ask.

Table 5: Distribution of code intelligence tasks across benchmark datasets, categorized by Software Development Lifecycle phases.

SDLC Phase	Coding Task	#Studies	#Datasets	References
Requirements & Design	API Generation	3	3	[127, 128, 129]
	Log Statement Generation	1	1	[130]
	Software Requirement Elicitation	2	2	[131, 132]
	Software Specification Generation	1	1	[133]
Implementation	Code Generation	75	86	[134, 135, 136, 137, 138, 126, 139, 140, 141, 142, 143, 144, 145, 146, 147, 148, 149, 150, 151, 152, 153, 154, 155, 156, 157, 122, 158, 159, 160, 161, 162, 163, 164, 165, 166, 167, 168, 169, 170, 171, 172, 173, 124, 174, 175, 176, 177, 178, 179, 180, 181, 182, 183, 184, 185, 186, 187, 188, 189, 190, 191, 192, 193, 121, 194, 195, 196, 197, 198, 199, 200, 201, 202, 203, 125]
	Code Translation	10	13	[204, 176, 126, 185, 205, 149, 206, 207, 195, 208]
	Code Completion	9	9	[123, 126, 135, 209, 187, 210, 179, 158, 211]
	General Software Development	2	2	[212, 196]
	Mobile Agent Evaluation	1	1	[213]
	Code Editing	3	3	[214, 215, 216]
	Code Optimization	1	1	[176]
	Code Summarization	3	3	[126, 185, 176]
	Code Refinement	1	1	[126]
	Clone Detection	1	1	[126]
	Code Search	1	1	[126]
	Code Retrieval	1	1	[211]
	API Recommendation	1	1	[135]
Comprehension	Code Understanding	5	5	[217, 218, 219, 158, 220]
	SE Question Answering	2	2	[221, 125]
	Formal Software Verification	1	1	[222]
	Code Reasoning	3	3	[219, 223, 224]
	Code Execution	2	2	[219, 192]
Testing	Test Case Generation	9	9	[182, 135, 225, 226, 227, 228, 229, 230, 231]
	Test Case Repair	3	3	[232, 233, 234]
Maintenance	Automated Program Repair	8	8	[235, 182, 110, 236, 237, 195, 238, 239, 240]
	Debugging	4	5	[241, 176, 111, 125]
	Issue Resolving	6	6	[242, 243, 244, 245, 246, 247]
	Bug and Defect Detection	5	5	[182, 126, 248, 241, 238]
	Fault Localization	8	8	[110, 241, 111, 179, 238, 239, 249, 250]
	Code Review	3	3	[251, 176, 125]
	Commit Message Generation	1	1	[179]
Security	Vulnerability Detection	9	9	[252, 253, 254, 112, 113, 255, 256, 114, 257]
Unique Total	32	142	156	-

Table 6: Task scope of benchmark datasets, i.e., Single-Task vs. Multi-Task.

Coding Task	# Datasets	# Studies	References
Single Task	130	118	[208, 127, 184, 167, 171, 163, 194, 198, 151, 199, 217, 218, 221, 124, 123, 225, 209, 186, 222, 235, 215, 161, 212, 188, 205, 190, 162, 197, 191, 189, 210, 203, 204, 139, 121, 236, 157, 168, 143, 177, 164, 136, 122, 148, 183, 174, 213, 155, 200, 147, 216, 141, 178, 165, 201, 150, 237, 145, 206, 214, 128, 146, 129, 207, 175, 220, 242, 243, 244, 245, 232, 159, 227, 228, 229, 240, 169, 249, 130, 112, 252, 202, 254, 156, 160, 144, 140, 224, 153, 173, 131, 113, 253, 250, 256, 180, 181, 251, 246, 133, 248, 154, 193, 231, 223, 138, 137, 247, 255, 142, 233, 234, 257, 132, 170, 134]
Multi-Task	26	24	[182, 176, 126, 135, 110, 185, 219, 241, 187, 172, 111, 149, 152, 192, 125, 179, 158, 238, 196, 239, 211, 226, 230, 195]
Unique Total	156	142	-

Table 7: Distribution of benchmark datasets by Code Context Granularity

Granularity	# Datasets	# Studies	References
Function Level	100	86	[249, 166, 130, 127, 184, 167, 204, 171, 163, 194, 252, 151, 156, 199, 217, 218, 176, 126, 202, 135, 110, 140, 185, 219, 222, 235, 241, 188, 215, 205, 187, 190, 224, 153, 197, 191, 189, 172, 208, 131, 139, 121, 236, 149, 157, 168, 143, 113, 169, 164, 136, 192, 125, 256, 205, 181, 251, 122, 148, 122, 168, 148, 158, 238, 183, 134, 171, 155, 200, 148, 205, 147, 147, 205, 141, 178, 248, 141, 139, 165, 154, 201, 237, 128, 146, 129, 175, 255, 220, 125, 125, 232, 159, 234, 229, 171, 240, 132, 114, 195]
Class Level	6	6	[198, 225, 180, 223, 150, 228]
Multi File	11	11	[112, 160, 186, 162, 174, 213, 216, 133, 233, 230, 257]
Repo Level	39	39	[144, 221, 124, 182, 123, 209, 254, 212, 161, 210, 173, 203, 111, 177, 253, 250, 152, 179, 246, 196, 193, 231, 239, 211, 138, 145, 137, 206, 214, 207, 142, 242, 243, 244, 245, 247, 226, 227, 170]
Unique Total	156	142	-

How Much Context Must a Model Handle? A second lens on difficulty is *context scope*: how much surrounding code a model must digest before it can solve the task. We distinguish four levels, and Table 7 gives the distribution across them:

- **Single-function**: a self-contained function specified by a docstring or signature – pure algorithmic logic, no external dependencies (100 datasets);
- **Class-level**: object-oriented tasks in which generated code must respect a class's internal state and interact with its other members (6 datasets);
- **Multi-file**: tasks with explicit cross-file dependencies, requiring correct imports and use of externally defined APIs or data structures (11 datasets);
- **Repository-level**: navigation of an entire codebase with implicit dependencies – the setting closest to real development, and the one that typically requires retrieval mechanisms or autonomous agents (39 datasets).

Figure 9 tracks these categories over time and shows a clear regime change. Until 2023, benchmarks were almost exclusively single-function. In 2024, repository-level datasets jumped from one to twenty-five, riding the wave of stronger long-context models and agent scaffolds, and by 2025 repository-scale evaluation had settled in as a standard companion to function-level testing rather than a novelty.

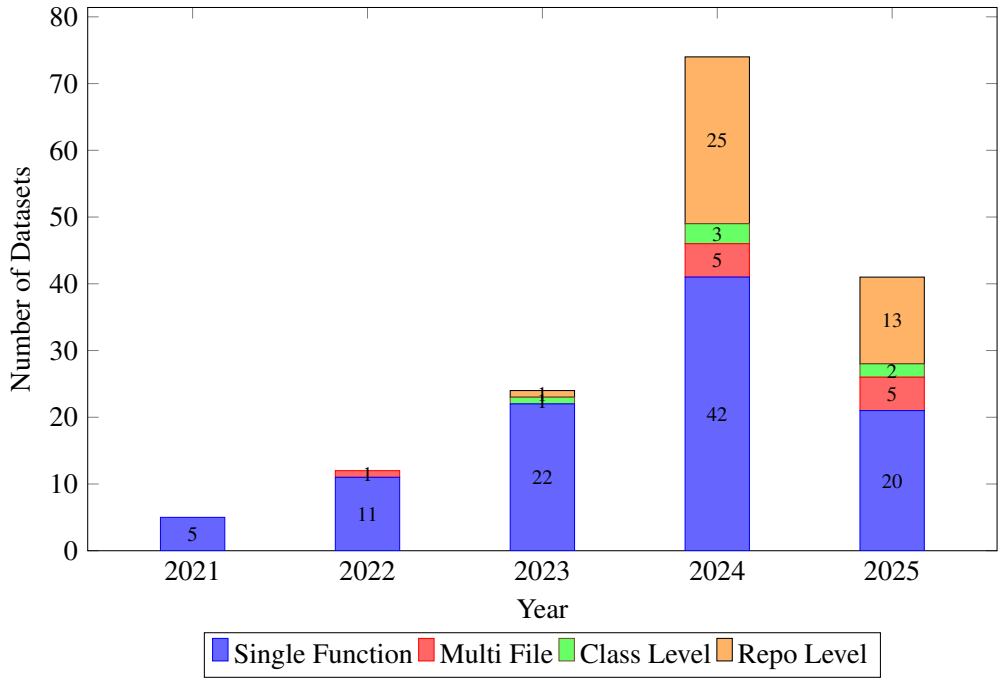


Figure 9: Evolution of code context scope in LLM benchmark datasets (2021–2025)

Table 8: Classification of benchmarks based on Interaction Paradigm

Paradigm	# Datasets	# Studies	References
Single-Turn	128	114	[171, 163, 112, 194, 252, 198, 151, 156, 199, 160, 217, 218, 124, 182, 176, 123, 126, 202, 135, 110, 225, 185, 209, 219, 254, 222, 235, 241, 188, 215, 161, 205, 187, 190, 162, 224, 153, 197, 191, 189, 172, 210, 173, 203, 208, 111, 131, 121, 236, 149, 157, 168, 113, 177, 253, 169, 136, 180, 152, 192, 125, 256, 179, 251, 122, 148, 158, 238, 183, 134, 155, 147, 216, 141, 178, 248, 133, 139, 165, 154, 239, 201, 223, 150, 237, 211, 138, 145, 137, 146, 129, 207, 175, 255, 142, 220, 233, 232, 159, 234, 227, 228, 229, 230, 240, 257, 132, 114, 195, 249, 166, 130, 184, 167, 204]
Multi-Turn	28	28	[127, 144, 221, 140, 186, 212, 139, 143, 164, 250, 181, 174, 213, 200, 246, 196, 193, 231, 206, 214, 128, 242, 243, 244, 245, 247, 226, 170]
Unique Total	156	142	-

One Shot or a Conversation? Benchmarks also differ in how much interaction they allow. *Single-turn* designs treat the model as a pure function: one prompt in, one answer out, no second chances. *Multi-turn or agentic* designs instead let the model act in an environment – run tests, read files, react to errors – and converge on a solution over several exchanges. As Table 8 shows, the single-turn paradigm still dominates (128 datasets against 28), even though iterative refinement is much closer to how developers, and increasingly coding agents, actually work.

Table 9: Distribution of benchmarks across various Application Domains

Domain	# Datasets	# Studies	References
Algorithm	78	70	[249, 184, 204, 198, 151, 156, 199, 217, 218, 182, 176, 202, 110, 140, 219, 222, 235, 241, 205, 224, 153, 197, 191, 172, 210, 173, 208, 111, 121, 236, 149, 157, 168, 143, 177, 169, 164, 136, 192, 181, 251, 122, 148, 158, 238, 183, 134, 155, 200, 147, 216, 178, 248, 133, 165, 196, 201, 223, 150, 137, 206, 175, 220, 232, 159, 234, 228, 229, 240, 195]
Data Science	19	15	[166, 167, 171, 194, 186, 188, 190, 162, 139, 180, 152, 174, 141, 154, 138]
Web Development	13	13	[127, 189, 193, 214, 128, 129, 207, 142, 243, 247, 233, 257, 170]
Security	9	9	[112, 252, 254, 113, 253, 256, 146, 255, 114]
Composite and Niche Domains	36	34	[163, 160, 144, 221, 124, 123, 126, 135, 225, 185, 209, 215, 212, 161, 187, 131, 250, 125, 179, 246, 231, 239, 237, 211, 145, 242, 244, 245, 226, 227, 230, 132, 213, 130, 203]
Unique Total	156	142	-

Application Domains Writing a sorting routine, wiring up a REST endpoint, and patching a SQL-injection flaw exercise very different skills, so we also classified every dataset by application domain (Table 9). Half of the corpus – 78 of 156 datasets – sits in the **Algorithm** domain of self-contained, largely language-agnostic problems about data structures and algorithmic logic [121, 122]. The second-largest group (36 datasets) comprises **composite and niche** benchmarks, from full-stack simulations such as FullStack Bench [203] to verticals like mobile development [213]. The remaining clusters are smaller but consequential: **Data Science** (19 datasets) tests mastery of the Pandas/Scikit-learn stack [141, 190, 188]; **Web Development** (14) spans UI generation through RESTful back-ends [170]; and **Security** (9) grounds its tasks in the CWE taxonomy of vulnerabilities [256, 255]. The skew toward algorithmic puzzles is one of the recurring criticisms revisited in RQ4.

Where Are They Published? Publication venues (Table 10) reveal how interdisciplinary this area has become. Conferences and symposia account for 89 datasets (57%), arXiv preprints for 57 (≈36.5%), and journals for the remaining 10 (6.5%). Machine-learning venues – NeurIPS, ICLR, ICML – host a large share of the corpus [242, 125, 148, 184], software-engineering venues (ICSE, FSE, ASE, TOSEM) contribute the practice-oriented benchmarks [135, 110, 143, 185], and NLP venues (ACL, EMNLP, NAACL) supply those that blend code with natural language [167, 205, 213]. The heavy preprint share – more than 26 arXiv releases in 2025 alone – reflects a community that values speed and open distribution over formal review, a trade-off revisited in the threats discussion.

Programming Languages Python is everywhere: 120 of the 156 datasets (77%) include it, alone or alongside other languages, with Java and C++ the most common companions (Table 11). A genuine multilingual

Table 10: Distribution of LLM benchmark datasets based on conference and journal venues

Conference Venues	# Datasets	Reference	Journal Venues	# Datasets	Reference
ICLR	14	[158, 230, 192, 218, 243, 140, 211, 163, 200, 148, 152, 242]	IEEE TSE	5	[147, 147, 145, 234, 132]
NeurIPS	14	[168, 183, 184, 126, 226, 189, 194, 164, 197, 209, 125]	IEEE Access	1	[239]
EMNLP	12	[178, 205, 186, 171, 127, 201]	TOSEM	1	[143]
ACL	8	[167, 204, 176, 235, 161, 212, 213, 195]	Science	1	[151]
ICSE	8	[185, 236, 215, 124, 198, 223, 139]	EAAI	1	[199]
ASE	5	[135, 208, 177, 146, 114]	TMLR	1	[222]
COLM	3	[216, 191, 172]	arXiv	57	[188, 210, 123, 166, 220, 214, 121, 122, 162, 174, 155, 150, 159, 203, 187, 179, 257, 170] [196, 207, 206, 245, 244, 238, 237, 221, 227, 228, 225, 169, 130, 112, 252, 202, 254, 142] [156, 160, 144, 153, 173, 131, 253, 250, 256, 181, 251, 246, 133, 248, 193, 231, 138, 255]
ICML	4	[190, 219, 224, 247]			
NAACL	6	[241, 180, 154, 137, 229, 134]			
ISSTA	2	[240, 182]			
IJCAI	2	[141, 141]			
AAAI	2	[129, 249]			
FSE	1	[110]			
ICST	1	[111]			
KDD	1	[149]			
HPDC	1	[165]			
AITEST	1	[175]			
ISSRE	1	[232]			
LREC-COLING	1	[157]			
MSR	1	[113]			
PACMPL	1	[233]			
Overall	**89**		**Overall**	**67**	

movement has emerged – several recent benchmarks span ten or more languages [148, 158, 147] – yet low-resource and domain-specific languages such as Haskell, Rust, and COBOL remain nearly invisible, leaving a blind spot around the ecosystems that power much critical infrastructure.

Language choice, it turns out, tracks task type rather than fashion. Generative, algorithm-flavoured tasks and data-science agents live almost entirely in Python; maintenance-oriented tasks – program repair, fault localisation, vulnerability detection – lean on the compiled languages (C, C++, Java) that dominate industrial legacy code; and web-development benchmarks are the main force pulling JavaScript into the corpus.

How Big Are the Datasets? Dataset sizes span seven orders of magnitude, from 14 samples to 189 million, so Figure 10 plots the distribution on a $\log_{10}$ scale. The mode is modest: 60 datasets fall in the $\approx$138–955 sample range that contains classics like HumanEval (164 problems) [121] and Defects4J-NL2Fix (283 bugs) [215], and another 46 fall between roughly 955 and 6,600 samples – the sweet spot where statistical comparison is meaningful but full evaluation remains affordable, home to MBPP [122] and SWE-Bench [242]. The tail, however, is extremely heavy: the mean (about 1.6 million samples) sits four orders of magnitude above the median ($\approx$759). This spread is not a defect – small datasets support careful manual analysis and human study, while the giants enable training and evaluation at scale – and several benchmarks explicitly ship multiple subsets at different sizes and difficulty levels [195, 152, 242].

Which Models Get Evaluated? More than a hundred distinct LLMs appear across the corpus (Table 12), but the distribution is top-heavy. OpenAI's GPT family features in 85% of the studies [258, 259], with GPT-3.5-Turbo and GPT-4 the de facto baselines [127, 197, 218, 163]. Code-specialised models – CodeLlama [260], StarCoder [261], CodeGen [200], Codex [121], CodeT5 [262], CodeQwen [263] – appear in 58% of benchmarks, and the LLaMA family [264, 265] in 46%. The corpus also exposes a persistent tension between the closed frontier models (GPT-4, Claude 3 Sonnet), which set the accuracy bar but resist reproduction, and the open models (CodeLlama, Mistral [266], LLaMA 3.1) that the community prefers when

Table 11: Distribution of programming languages across benchmark datasets

Programming Language	#Studies	#Datasets	References
Python	106	120	[166, 127, 184, 167, 171, 163, 198, 151, 199, 217, 218, 221, 124, 176, 123, 126, 135, 110, 209, 219, 186, 235, 241, 161, 212, 188, 205, 187, 190, 162, 197, 191, 189, 172, 210, 203, 204, 139, 121, 149, 157, 168, 143, 164, 136, 152, 192, 125, 179, 122, 148, 158, 238, 183, 174, 155, 200, 147, 141, 178, 196, 239, 201, 223, 150, 211, 145, 206, 214, 128, 146, 207, 242, 245, 226, 159, 227, 229, 230, 169, 195, 208, 249, 130, 202, 156, 160, 144, 140, 224, 153, 173, 180, 181, 251, 248, 154, 193, 231, 223, 138, 137, 233, 229, 257, 132, 134]
Java	51	59	[204, 218, 221, 124, 182, 176, 126, 135, 110, 209, 235, 241, 215, 212, 205, 197, 203, 111, 236, 149, 157, 177, 164, 125, 148, 158, 238, 147, 216, 237, 211, 206, 146, 129, 175, 220, 244, 232, 228, 195, 130, 254, 160, 256, 251, 246, 133, 234, 257]
Ada	2	5	[176, 205]
Assembly	2	4	[176, 125]
Bash	3	5	[203, 125, 146]
BBC Basic	1	4	[205]
C	13	19	[218, 176, 212, 205, 197, 164, 125, 158, 238, 147, 165, 206, 195, 112, 252, 113, 253, 250, 256, 246, 133, 255, 257]
C#	18	26	[208, 218, 176, 126, 209, 205, 203, 157, 164, 125, 148, 158, 238, 147, 206, 146, 195, 251]
C++	29	37	[208, 218, 176, 225, 235, 241, 212, 205, 197, 203, 149, 125, 148, 158, 147, 165, 238, 206, 146, 240, 195, 130, 112, 160, 113, 253, 256, 251, 246, 255, 257]
Cobol	2	5	[176, 205]
CSS	2	2	[218, 170]
D	5	9	[176, 205, 203, 147, 195]
Dafny	1	1	[222]
Dart	4	6	[176, 164, 125, 158]
Delphi	3	8	[176, 205, 125]
Elisp	1	1	[158]
Elixir	4	9	[205, 164, 125, 158]
Erlang	3	6	[205, 164, 158]
F#	4	7	[176, 205, 158, 238]
Fortran	3	6	[176, 205, 158]
Go	20	28	[151, 221, 176, 126, 205, 203, 149, 157, 164, 125, 148, 158, 238, 147, 146, 195, 160, 251, 246, 257]
Groovy	2	2	[176, 158]
Haskell	5	8	[176, 125, 158, 147, 195]
HTML	7	7	[218, 203, 158, 238, 146, 257, 170]
J	1	4	[205]
Javascript	23	27	[208, 218, 221, 176, 126, 212, 203, 149, 157, 164, 125, 158, 238, 148, 147, 243, 195, 251, 246, 247, 257, 170]
Julia	5	9	[176, 205, 158, 238, 147]
Kotlin	8	13	[176, 157, 164, 125, 158, 148, 147, 195]
Lua	5	9	[151, 176, 205, 158, 147]
Matlab	3	6	[218, 176, 205]
MySQL	1	1	[164]
Objective-C	2	4	[176, 125]
OCAML	2	5	[205, 195]
Pascal	4	7	[205, 158, 238, 195]
Perl	8	16	[176, 205, 157, 125, 158, 148, 147, 195]
PHP	16	24	[151, 218, 176, 205, 203, 157, 164, 125, 158, 238, 148, 147, 146, 195, 251, 257]
R	8	14	[218, 176, 205, 203, 125, 158, 238, 147, 154]
Racket	2	2	[164, 158]
Ruby	15	20	[151, 218, 176, 126, 203, 157, 164, 125, 158, 238, 148, 147, 195, 251, 257]
Rust	12	18	[151, 176, 205, 203, 164, 125, 158, 238, 147, 207, 195, 246]
SAS	1	1	[176]
Scala	13	20	[151, 176, 205, 203, 157, 164, 125, 158, 238, 148, 147, 195]
Shell	2	4	[125, 158]
SQL	4	6	[194, 218, 203, 125]
Swift	9	17	[176, 205, 157, 164, 125, 158, 238, 148, 147]
TypeScript	15	20	[151, 218, 221, 209, 203, 157, 164, 125, 158, 148, 147, 251, 246, 247, 257]
Visual Basic	3	6	[176, 205, 158]
Solidity	3	3	[256, 142, 257]

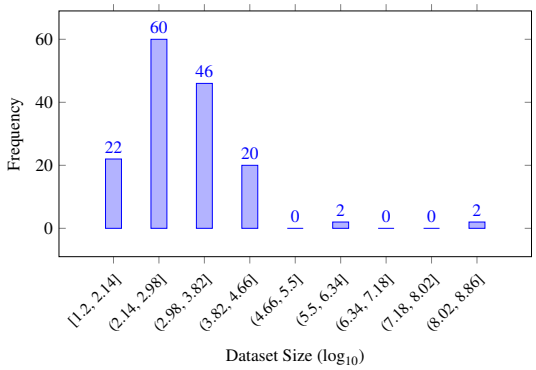


Figure 10: Distribution of LLM benchmark dataset sizes (scaled using $\log_{10}$)

transparency and repeatability matter.

Adoption: A Few Standards and a Long Tail Citation counts, used as a proxy for adoption, follow a power law (Figure 11). At the head sit a handful of de facto standards: HumanEval [121] with over 7,100 citations, MBPP and MathQA-Python [122] near 2,600 each, and CodeContests [151] and MTPB [200] above 1,500 – early releases that defined the field's evaluation protocols and have been grandfathered into every comparison since. A middle tier of domain leaders (roughly 200–1,500 citations) shows how quickly a benchmark that fills a real gap can rise: SWE-Bench [242] became the standard for agentic issue resolution within two years, LiveCodeBench [192] passed 800 citations on the strength of its contamination resistance, and DS-1000 [190] owns the data-science niche. The long tail – most of the corpus, under 100 citations each – is not low-quality so much as specialised: benchmarks for vulnerability detection [255], mobile development [213], or individual languages serve narrower audiences by design.

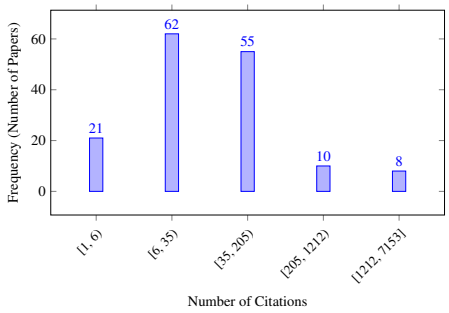


Figure 11: Distribution of citations for the papers introducing LLM benchmark datasets

RQ1 in brief. Benchmark construction has accelerated sharply since 2023, with the last eighteen months of the survey window contributing over half the corpus. The landscape is dominated by code generation, by Python (77% of datasets), by single-turn and single-function designs, and by evaluations centred on the GPT model family; dataset sizes range over seven orders of magnitude, and adoption follows a power law in which a few early benchmarks serve as universal standards.

Table 12: Distribution of LLMs in the subject studies

Category	#Studies	Models	References
OpenAI GPT-family	117	o1-preview, o1-mini, GPT-4, GPT-4o, GPT-4o-mini, GPT-3, GPT-3.5-Turbo, GPT-4-Turbo, Text-DaVinci-003, Code-DaVinci-002, Code-DaVinci-Edit-001, Codex-DaVinci-002, Codex-DaVinci-001	[166, 127, 184, 163, 194, 198, 199, 217, 218, 221, 124, 182, 176, 123, 126, 135, 209, 219, 186, 222, 235, 241, 215, 161, 212, 188, 205, 187, 190, 197, 191, 189, 172, 203, 208, 139, 157, 168, 143, 177, 164, 136, 152, 192, 125, 179, 122, 158, 238, 174, 213, 155, 200, 216, 141, 165, 196, 239, 201, 223, 150, 237, 145, 206, 214, 128, 146, 129, 207, 175, 220, 242, 243, 244, 245, 226, 232, 159, 227, 228, 229, 230, 240, 169, 249, 130, 252, 202, 254, 156, 160, 144, 140, 224, 173, 131, 253, 250, 256, 180, 181, 251, 246, 133, 248, 154, 193, 231, 138, 137, 247, 255, 142, 257, 132, 170, 134]
Meta LLaMA-family	80	LLaMA 3, LLaMA 3.1, LLaMA 2, LLaMA, CodeLLaMA, TinyLLaMA	[249, 130, 252, 202, 156, 160, 224, 153, 173, 256, 181, 251, 133, 154, 193, 231, 138, 137, 247, 142, 257, 132, 170, 166, 163, 217, 218, 182, 176, 123, 135, 225, 219, 222, 235, 241, 161, 212, 188, 187, 197, 191, 189, 172, 210, 203, 168, 143, 136, 152, 192, 125, 179, 158, 238, 183, 174, 213, 155, 216, 165, 201, 223, 150, 237, 206, 214, 128, 146, 129, 207, 220, 159, 227, 228, 229, 230, 240, 169, 195]
Code-centric Models	63	StarCoder, StarCoder2, CodeGen, InCoder, CodeT5, WizardCoder, CodeGemma, DSCoder, OpenCoder, CodeGeeX, CodeGPT, CodeBERT, Magicoder-S-DS, PLBART, Magicoder, StarCoderBase, GraniteCoder, SantaCoder, CodeT5+, PolyCoder, TransCoder, DSCoder V2, CodeGen2, CodeClippy, GraphCodeBERT, CodeParrot, PanguCoder,OctoCoder, StableCode, PyCodeGPT, PalmCoder, StarChat	[130, 202, 224, 173, 180, 181, 137, 142, 234, 204, 167, 171, 163, 194, 198, 217, 124, 182, 176, 123, 126, 135, 185, 209, 219, 241, 161, 205, 190, 197, 191, 189, 172, 210, 203, 208, 236, 149, 157, 168, 177, 164, 192, 238, 183, 155, 200, 147, 216, 141, 178, 165, 196, 201, 223, 150, 211, 220, 159, 229, 240, 169, 195]
Anthropic Claude-family	46	Claude 3.5 Sonnet, Claude 3 Opus, Claude 3 Haiku, Claude 3 Sonnet, Claude 2, Claude 1, Claude 3.5 Haiku, Claude Instant	[249, 202, 156, 160, 144, 224, 173, 181, 246, 133, 154, 231, 137, 247, 255, 257, 170, 134, 163, 194, 217, 218, 182, 123, 186, 222, 188, 197, 172, 203, 136, 152, 192, 125, 238, 174, 196, 150, 237, 206, 214, 207, 242, 243, 226, 240]
Mistral-family	43	Mistral, Mixtral, CodeStral	[252, 202, 160, 224, 173, 131, 180, 154, 137, 255, 257, 170, 163, 217, 218, 221, 123, 225, 219, 186, 235, 188, 197, 172, 210, 168, 136, 152, 192, 125, 179, 158, 238, 155, 216, 201, 223, 150, 237, 206, 226, 227, 230]
Alibaba Qwen-family	37	CodeQwen 1.5, Qwen 2.5, Qwen 2, Qwen 2.5 Coder, Qwen 1.5, QwenCoder 2.5, Qwen1	[252, 156, 160, 224, 173, 256, 181, 246, 133, 231, 138, 137, 255, 142, 170, 163, 217, 218, 123, 186, 241, 187, 191, 189, 172, 210, 203, 143, 136, 152, 158, 238, 183, 201, 150, 237, 214]
DeepSeek-family	37	DeepSeek, DeepSeek V2, DeepSeek V2.5, DeepSeek-Coder V2, DeepSeekCoder 2.5, DeepSeek R1	[233, 130, 252, 202, 156, 160, 144, 224, 153, 173, 256, 181, 246, 133, 154, 231, 138, 137, 247, 255, 142, 170, 163, 218, 123, 186, 241, 187, 172, 203, 143, 155, 237, 206, 214, 207, 244]
Google Gemini-family	25	Gemini Flash, Gemini Pro	[252, 202, 156, 160, 224, 173, 181, 248, 154, 193, 231, 137, 257, 170, 217, 221, 172, 152, 192, 125, 237, 214, 229]
Microsoft Phi-family	18	Phi-2, Phi-3, Phi-1.5, Phi-4, Phi-4 Mini, Phi-3 Mini	[202, 156, 160, 224, 153, 173, 256, 246, 193, 138, 137, 218, 219, 187, 172, 152, 155, 201]
others	53	Vicuna, GPT-Neo, Gemma, Gemma 2, GPT-J, Yi-Coder, Yi, ChatGLM, Phind, PaLM 2, Yi 1.5, Phind V2, OpenCodeInterpreter-DS, Lynx, Alpaca, QwQ, AlphaCode, PaLM, RoBERTa, BART, StableLM, Doubao-1.5-pro, WizardLM, Zephyr, Tulu-2, Vicuna 1.5, SwellLLaMA, Bard, GLM-4	[131, 246, 133, 257, 170, 127, 184, 167, 171, 163, 194, 198, 151, 217, 218, 123, 126, 185, 219, 190, 162, 197, 189, 172, 210, 203, 139, 121, 149, 168, 164, 152, 192, 125, 122, 158, 238, 155, 200, 147, 141, 178, 201, 223, 211, 128, 146, 129, 242, 229, 230, 240, 169]
Unique Total	**137**	-	-

4.1.2 RQ2: How Are the Benchmarks Built, and How Good Is the Data?

Scores are only as trustworthy as the datasets behind them. The second research question therefore looks inside the benchmarks: their raw material, their organisation, the quality assurance applied during construction, their claims to realism, and their defences against data contamination.

Raw Material: Four Sourcing Strategies Table 13 breaks down where benchmark data originates; many datasets combine several sources. *GitHub* is the single biggest supplier – 66 datasets (46%) mine its repositories, issues, pull requests, and version histories to recreate authentic engineering contexts, as DevEval [161], SWE-Bench [242], RepoEval [145], and ExecRepoBench [210] all do. *Manual authorship* comes second: 37 datasets (26%), including HumanEval [121], MBPP [122], LBPP [136], and API-Bank [127], were written from scratch, trading realism for clean task design and control. *Competitive-programming platforms* supply 35 datasets (24.6%) – LeetCode is the main source for 15 of them [169, 164, 176], with Codeforces and AtCoder behind roughly five more [192, 159, 195]. Finally, 32 datasets (22.5%) are *derivatives*, extending or repackaging earlier benchmarks: HumanEval+ [168] hardens HumanEval's tests, SWE-Bench+ [245] de-contaminates SWE-Bench, CodeJudge-Eval [217] re-purposes existing problems for judge evaluation, and NL2Fix [215] adapts the pre-LLM Defects4J [267]. Educational sites (GeeksForGeeks, Kaggle, Rosetta Code, W3Schools), StackOverflow, and the NVD round out the picture. Collectively, the trend line points toward repository-scale, real-world sourcing – but the curated and derivative datasets keep their place for targeted, controlled probing.

Anatomy of a Benchmark Beneath the variety, a few structural blueprints recur. The most common is the *problem-centric* layout – a task description handed to the model, plus unit tests or a canonical solution to judge the response – shared by HumanEval, MBPP, MathQA-Python, and APPS [121, 122, 184], preserved by their hardened successors HumanEval+ and LBPP [168, 136], and ported to other languages by HumanEval-X and MBXP [148]. *Translation* benchmarks instead pair equivalent programs across languages, sometimes with tests to confirm the translation still runs and means the same thing [205, 206, 147]; G-TransEval [208] additionally stratifies its pairs by translation type and difficulty. *Debugging* datasets couple buggy snippets with explanations, golden patches, and tests, often labelled by bug type or debugging stage [235, 237]. *Multi-task* suites such as CodeXGLUE and Long Code Arena [126, 179] adopt a modular organisation with per-task metadata and harnesses. *Data-science* benchmarks reproduce the notebook environment itself – contextual Jupyter cells, library-heavy pipelines, even Kaggle-style medal scoring [167, 162, 190]. *Repository-level* datasets (RepoEval, ComplexCodeEval, RepoBench) ship whole repository snapshots with issue metadata and test suites so that cross-file dependencies can be exercised. And a few designs are unique to their niche: SWE-Bench Multimodal [243] embeds screenshots in issue reports, while Mobile-Bench and EffiBench [213, 197] provide the domain-specific environments their tasks demand.

Table 13: Detailed distribution of benchmark sources

Source	Detailed Sources	# Studies	References
GitHub	-	66	[112, 176, 126, 135, 222, 189, 152, 216, 226, 227, 228, 230, 127, 217, 221, 124, 123, 110, 225, 235, 161, 212, 162, 197, 172, 210, 111, 213, 165, 239, 211, 214, 146, 129, 207, 242, 243, 244, 245, 159, 167, 209, 215, 187, 203, 143, 179, 238, 166, 186, 130, 254, 173, 113, 253, 256, 180, 246, 231, 138, 137, 142, 233, 234, 257, 170]
Manually Created	-	37	[252, 251, 198, 126, 141, 239, 244, 127, 184, 171, 163, 194, 185, 212, 188, 203, 139, 121, 236, 149, 157, 177, 136, 152, 122, 158, 155, 200, 178, 196, 201, 145, 214, 128, 146, 232, 240]
Exisiting Datasets	HumanEval	11	[218, 143, 148, 147, 191, 168, 202, 156, 224, 181, 134]
	MBPP	7	[148, 147, 191, 168, 202, 144, 181]
	Defects4J	4	[182, 244, 161, 237]
	APPS	4	[217, 159, 202, 181]
	ClassEval	2	[150, 223]
	Code-Contest	2	[159, 202]
	CodeNet	2	[218, 249]
	CodeSearchNet	2	[126, 181]
	PrimeVul	2	[253, 255]
	MBPP+	2	[150, 153]
	Others	16	[160, 160, 144, 224, 181, 193, 159, 237, 150, 152, 241, 148, 171, 218, 206, 222]
Coding Websites	LeetCode	15	[218, 126, 235, 192, 159, 229, 197, 152, 183, 176, 201, 169, 164, 156, 153]
	AtCoder	5	[184, 151, 110, 152, 192]
	CodeForces	5	[192, 159, 184, 151, 195]
	CodeChef	3	[151, 199, 159]
	CodeWars	2	[159, 184]
	Kattis	2	[184, 159]
	Aizu	1	[151]
	HackerEarth	1	[151]
	HackerRank	1	[159]
StackOverflow	-	5	[125, 190, 178, 129, 175]
GeekForGeeks	-	4	[201, 208, 159, 218]
Kaggle	-	5	[152, 174, 167, 194, 186]
National Vulnerability Database	-	3	[112, 113, 255]
W3School	-	2	[218, 175]
CVE	-	2	[255, 256]
Common Crawl	-	1	[218]
Frama-C-Problems	-	1	[220]
Jira	-	1	[161]
JML	-	1	[220]
Dive into Deep Learning Platform	-	1	[205]
Rosetta Code	-	1	[205]
Socrata	-	1	[152]
SpecGen	-	1	[220]
SVN	-	1	[161]
Upwork	-	1	[247]
Unique Total	-	142	-

Quality Control – or the Lack of It Perhaps the most sobering finding of RQ2: 67 datasets – 43% of the corpus – describe *no* quality-control procedure at all for their data collection (Table 14). Where quality assurance is reported, it takes four forms:

- **Manual and expert review** (32%), from author review of every sample [136, 232] to spot checks of subsets [205, 159] and, occasionally, dedicated external validators [128];
- **Multi-annotator verification** (9%), where reviewers other than the creators cross-check entries – FullStackBench [203] has annotators exchange and refine each other's work, and MCEval [158] double-annotates every sample with senior adjudication of conflicts;
- **Automated filtering** (8%), indispensable at scale – CodeRepoQA [221] distilled roughly 636,000 raw GitHub issues into 585,687 usable QA pairs via length, duplication, link, and bot filters, while SecVulEval [255] strips non-C files, multi-commit fixes, and pure refactorings;
- **Hybrid pipelines** (8%) that interleave human and automated (often LLM-assisted) checking – BigCodeBench's [163] curation/dry-run/cross-check chain, RepoTransBench's [206] iterate-until-it-compiles loop, and CrossCodeEval's [209] LLM pre-filtering before human annotation.

Table 14: Quality controls applied in data collection process

Quality Control	# Datasets	# Studies	References
Manual inspection / expert review	50	38	[171, 182, 205, 162, 189, 136, 125, 122, 148, 213, 200, 147, 141, 178, 239, 201, 145, 128, 146, 129, 207, 243, 244, 245, 232, 159, 228, 240, 160, 156, 193, 252, 253, 134, 254, 224, 248, 257]
Automated filtering / execution-based checks	15	15	[218, 221, 135, 219, 210, 111, 202, 231, 153, 173, 137, 113, 170, 255, 138]
Multi-annotator / peer verification	12	12	[127, 167, 194, 203, 158, 238, 220, 251, 144, 246, 250, 247]
Hybrid	12	12	[166, 163, 209, 235, 190, 157, 179, 155, 206, 195, 181, 154]
No Quality Control	67	65	[184, 204, 198, 151, 199, 217, 124, 176, 123, 126, 110, 225, 185, 186, 222, 241, 188, 215, 212, 161, 187, 197, 191, 172, 208, 139, 121, 236, 149, 168, 143, 177, 164, 152, 192, 183, 174, 216, 165, 196, 223, 150, 237, 211, 214, 175, 242, 226, 227, 229, 230, 169, 140, 142, 234, 130, 249, 133, 180, 233, 112, 131, 132, 256, 114]

Comparing across communities, software-engineering and AI venues agree on one thing – human review is the backbone of ground truth – but differ in what they add to it: SE venues favour execution-based automated checks that make correctness reproducible, while AI venues lean toward hybrid and multi-annotator workflows better suited to the scale and subjectivity of LLM-generated content.

Table 15: How datasets ensured real-world representativeness

	# Datasets	# Studies	References
Sourcing from Real-World Data Sources	65	63	[166, 204, 171, 194, 151, 199, 217, 218, 221, 124, 182, 176, 126, 135, 110, 225, 222, 188, 215, 212, 161, 187, 197, 189, 111, 208, 164, 152, 192, 183, 216, 223, 211, 145, 206, 214, 146, 207, 175, 242, 243, 244, 245, 226, 159, 169, 195, 156, 142, 231, 234, 130, 251, 173, 249, 154, 254, 112, 113, 246, 247, 256, 114]
Emphasis on Realistic Context and Dependencies	15	15	[172, 157, 136, 165, 140, 202, 153, 181, 137, 233, 131, 224, 248, 228, 209, 210]
Problem Design to Mirror Developer Workflows	25	25	[184, 230, 167, 186, 190, 162, 191, 203, 143, 179, 200, 196, 201, 150, 128, 232, 227, 240, 160, 144, 193, 180, 134, 170, 138]
Ensuring Realness through Community and Industrial Validation	13	9	[127, 123, 139, 125, 174, 213, 141, 178, 129]
Coverage and Structural Properties	18	15	[198, 185, 235, 241, 205, 177, 155, 239, 237, 252, 253, 133, 250, 255, 257]

Claims to Realism How do benchmark authors argue that their datasets reflect real software practice? Table 15 groups the strategies. The most common (42%) is to *source from reality directly* – recent non-forked repositories in RepoBench [211], active projects and contest platforms in CodeScope [176], and human-authored contest problems with strong tests in XCodeEval, TACO, and the LeetCode Contest Benchmark [195, 159, 169]. About 16% instead *design tasks around developer workflows*: APPS [184] poses problems in free-form natural language the way coding sites do, and TestGenEval [230] centres on maintaining test suites in living repositories. A further 10% emphasise *realistic dependency structure* – cross-file completion in ExecRepoBench and CrossCodeEval [210, 209], functions drawn from large projects in TestBench [228]. Others rely on *community or industrial validation* (upvoted StackOverflow answers in StackEval [125], screened Kaggle competitions in MLE-Bench [174], telemetry from a production completion tool in Codev-Bench [123]) or on *structural guarantees* (high-coverage test suites in ClassEval [198], deliberately non-trivial bugs in PyBugHive [239], breadth from basic syntax to neural-network code in CodeTransOcean [205]).

The Contamination Problem When benchmark content leaks into a model's training data, evaluation measures memory, not ability. Four defence strategies appear in the corpus (Table 16). *Temporal isolation* keeps evaluation data strictly newer than the models' training cut-offs – the approach of LiveCodeBench's rolling updates [192] and RepoEval's fresh repositories [145]. *Corpus exclusion* scrubs overlap algorithmically, via N-gram analysis, MinHash deduplication, or exact matching [163, 130, 210]. *Novelty and perturbation* make memorisation useless by hand-writing new problems [121] or semantically perturbing existing ones [190]. *Private holdouts* withhold the data entirely, as SWE-Lancer [247] does with its sandboxed private split.

The headline number, though, is on the other side of the ledger: **86 of 156 datasets take no contamination measure whatsoever**. For older and widely mirrored benchmarks in particular, reported scores should be read with that caveat firmly in mind. Provenance largely determines the defence chosen: hand-written datasets decontaminate proactively by construction, contest-sourced ones lean on publication timestamps for temporal isolation, and GitHub-derived or derivative datasets – the very material most likely to sit in pre-training corpora – most often rely on reactive de-duplication or do nothing at all.

Table 16: Overview of Data Contamination Mitigation Approaches in Code Intelligence Datasets

Mitigation Methodology	# Datasets	# Studies	References
Temporal Isolation & Knowledge Cutoffs	20	20	[151, 199, 135, 110, 209, 235, 189, 111, 169, 164, 152, 192, 211, 145, 242, 245, 233, 234, 229, 195]
Pre-training Corpus Exclusion	18	17	[130, 184, 167, 163, 218, 185, 222, 162, 210, 148, 238, 141, 133, 137, 159, 230, 114]
Perturbation and Novelty Construction	29	27	[198, 124, 176, 190, 208, 131, 121, 236, 113, 177, 136, 180, 256, 179, 122, 174, 155, 147, 150, 214, 255, 142, 220, 123, 224, 153, 183]
Environmental Isolation and Private Holdouts	3	3	[194, 247, 240]
No Mitigation Measures	86	75	[127, 249, 166, 204, 171, 112, 252, 156, 160, 144, 217, 221, 182, 126, 202, 140, 225, 219, 254, 186, 241, 188, 215, 212, 161, 205, 187, 197, 191, 172, 173, 203, 139, 149, 157, 168, 143, 253, 250, 125, 181, 251, 148, 158, 134, 213, 200, 246, 216, 178, 248, 165, 154, 196, 193, 231, 239, 201, 223, 237, 138, 206, 128, 146, 129, 207, 175, 243, 244, 226, 232, 227, 228, 257, 132, 170]

RQ2 in brief. GitHub mining and manual authorship are the two dominant construction routes, and the natural-language-task-plus-unit-tests blueprint is the most common structure. Quality assurance is uneven – manual review leads among datasets that have any, but 43% report none – and while roughly half the corpus grounds its realism in practical sources and workflows, more than half of all datasets take no precaution against data contamination.

4.1.3 RQ3: How Is Performance Measured?

The Metric Inventory Metrics follow tasks, and the corpus contains a remarkably long inventory of them (Table 17). One metric towers over the rest: **Pass@k** – the probability that at least one of k sampled solutions passes the unit tests – appears in 51 studies, with its special case Pass@1 in 13 more [167, 121, 198, 192, 189]. Around it cluster the other correctness measures: *accuracy* (usually a synonym for pass rate, though RepoEval and StackEval use it in the classification sense), *pass/success rate* [212, 149, 122, 183, 200], the *average test-case pass ratio* [184, 199], and rubric-based *acceptance rate* [125].

Beyond correctness, specialised metrics proliferate wherever a benchmark cares about something Pass@k cannot see. Data-science suites score pipeline artefacts (MLE-Bench's Kaggle-style *medal rate* [174]; DA-Code's *ML/chart/table match scores* [186]). Efficiency-aware benchmarks weigh runtime and memory (*Efficiency Score* [183], *Valid Efficiency Score* for SQL [194], *Differential Performance Score* [191], *Efficiency@k*/*Speedup@k* [165, 150], raw *Runtime* and *Memory Usage* [197, 196]). Maintainability-minded ones inspect the code itself (*Readability* and *Maintainability Index* [150], *LOC* [175, 166], *Cyclomatic/Cognitive Complexity* [175]). Repository-scale generation brings dependency- and structure-aware measures (*DEP* [198], *Library Coverage* and *Reuse Correctness* [145], *Compilation@k*/*Completion@k* [177], *Coverage@k* [182], *Edit Similarity* [135, 209, 210, 211], *Execution Rate* [196], F1 and mAP [179, 145]), and interaction-oriented designs even rate communication quality (*Good Question Rate*, *Communication Rate* [143]).

Text-similarity metrics form their own family, used both for generation and for the translation/summarisation/QA cluster: BLEU [204, 205, 208, 185, 176, 221], CodeBLEU [206, 178], METEOR, ROUGE-L [185], ChrF [179], BERTScore [176], Exact Match and Data-Flow Match [205, 204], complemented by build-and-run checks such as *Build@k* [206] and *Debugging Success Rate@k* [207].

Task-specific families complete the inventory. Test generation and repair use coverage and mutation measures – line/branch/change coverage [228, 229, 226], *Cov@k* [230], *Mutation Score* [229], *Fail-to-Pass Rate* and *TDD Score* [227], *Patch Applicability* [226], *Win Rate* [225], *DiffBLEU* and *Syntax Pass Rate* [232]. Issue resolution reports *% Resolved* and *Average Cost* [242, 243, 244, 245]; program repair adds *AST Match@1*, *Plausible@1* [237], correct-fix counts [236], and *Count@k* [182]; bug repair contributes *Top-5*, *Avg-5*, and the *Relative Patch Size Ratio* [240]; code reasoning relies on accuracy and the *Incremental Consistency score* [223] alongside Pass@k in CRUXEval [219]; and formal verification and mobile agents use success rate [222] and workflow *CheckPoints* [213], respectively.

Table 17: Detailed distribution of benchmark evaluation metrics

No	Source	# Studies	References
1	% Resolved	6	[242, 243, 244, 245, 246, 247]
2	Acceptance Rate	1	[125]
3	Accuracy	33	[234, 130, 249, 252, 253, 112, 113, 248, 255, 132, 127, 204, 217, 218, 182, 176, 126, 241, 208, 139, 149, 152, 125, 238, 216, 223, 211, 145, 146, 220, 232, 195, 194, 150]
4	Api Misuse Rate	1	[129]
5	Ast Match@1	1	[237]
6	Avg Generated Solution Complexity	1	[175]
7	Avg Generated Reference Complexity	1	[175]
7	Avg Cost	5	[196, 237, 243, 244, 245]
8	Avg Speedup	1	[216]
9	Avg Step	2	[186, 213]
10	Avg-5	1	[240]
11	Bertscore	3	[176, 179, 134]
12	Beyond(Efficiency Score)	1	[183]
13	Bleu	13	[234, 132, 167, 221, 176, 126, 135, 185, 205, 208, 149, 179, 178]
14	Build Repair Rate	1	[179]
15	Build@k	1	[206]
16	Code Coverage	1	[226]
17	Checkpoint	1	[213]
18	Chrf	2	[179, 178]
19	Code Lines	1	[166]
20	Code Tokens	1	[166]
21	Codebleu	9	[234, 126, 135, 205, 208, 149, 178, 211, 232]
22	Communication Rate	1	[143]
23	Compilation Rate	3	[193, 129, 228]
24	Compilation@k	2	[182, 177, 142]
25	Completion Rate	1	[186]
26	Completion@k	1	[177]
27	Correctness	6	[228, 229, 229, 193, 137, 233]
28	Cov@k	6	[193, 233, 182, 228, 229, 230]
29	Data Wrangling Score	1	[186]
30	Debugging Success Rate@k	3	[176, 205, 207]
31	DiffBLEU	2	[216, 232]
32	Differential Performance Score	1	[191]
33	Eda Score	1	[186]
34	Edit Distance	1	[152]
35	Edit Similarity	6	[221, 126, 135, 209, 210, 211]
36	Efficiency Score	3	[194, 197, 150]
37	Efficiency@k	2	[165, 202]
38	Exact Match	7	[234, 126, 185, 209, 205, 179, 211]
39	Executability Rate	1	[196]
40	F1 Score	20	[217, 176, 126, 135, 209, 179, 223, 145, 195, 251, 253, 112, 113, 131, 248, 255, 180, 254, 132, 256]
41	Failtopass	1	[227]
42	Good Question Rate	1	[143]
43	Incremental Consistency Score	1	[223]
44	LLM as a Judge	1	[225]
45	Machine Learning Score	1	[186]
46	Maintainability Index	1	[150]
47	Map	1	[179, 126]
48	Medal Rate(like Kaggle)	1	[174]
49	Memory Usage	1	[197]
50	Meteor	3	[132, 176, 178]
51	MRR	4	[140, 180, 250, 126]
51	Mutation Score	1	[229]
52	Number Of Correct Fixes	1	[236]
53	Number Of Errors	1	[196]
54	Pass Rate	9	[173, 154, 235, 188, 143, 183, 200, 175, 220, 213]
55	Pass@1	13	[199, 123, 190, 172, 210, 203, 152, 192, 158, 238, 207, 175, 169]
56	Pass@k	51	[140, 156, 142, 231, 144, 153, 181, 133, 170, 224, 138, 184, 167, 171, 163, 198, 151, 124, 182, 176, 219, 215, 161, 212, 187, 162, 197, 191, 189, 121, 149, 157, 168, 177, 136, 164, 122, 148, 155, 174, 200, 147, 141, 178, 165, 201, 214, 129, 159, 230, 195]
58	Patch Applicability	1	[226]
59	Plausible@1	1	[237]
60	Precision	15	[251, 173, 252, 253, 254, 112, 113, 131, 248, 255, 132, 176, 179, 145, 220]
61	Readability	1	[150]
62	Recall	17	[140, 251, 173, 252, 253, 254, 112, 113, 131, 248, 255, 132, 176, 135, 179, 145, 220]
63	Recall@k	2	[189, 250]
64	Rouge	4	[132, 176, 179, 178]
65	Rouge-l	3	[127, 221, 185]
66	RPSR	1	[240]
67	Runtime	3	[197, 196, 244]
68	Speedup@k	1	[165]
69	Success Rate	7	[166, 204, 222, 239, 206, 128, 226]
70	TDDScore	1	[227]
71	Token Usage	1	[196]
72	Top-5	1	[240]
73	Win Rate	1	[225]
-	**Unique Total**	**142**	-

Six Evaluation Methodologies Zooming out from individual metrics, the corpus employs six recognisable evaluation methodologies:

1. *Execution-based* evaluation – run the generated code against tests – is the workhorse, present in over 90 benchmarks [184, 122, 121]. It subsumes unit-test pass rates, compilation checks [182], performance profiling [150, 197], and end-to-end task-resolution measurement [242, 207], and it offers the strongest guarantee that "correct" means "actually works".

2. *Similarity-based* evaluation (17 benchmarks) compares output text to references with BLEU-family, edit-distance, or embedding metrics – appropriate where executability is not the point, as in translation or summarisation.

3. *Human-feedback* evaluation recruits annotators with structured rubrics, peer review, and majority voting [196]; costly, but the gold standard for qualities automated metrics miss [125].

4. *LLM-as-a-judge* evaluation replaces the human with a model given context and a rubric – scalable, cheap, and empirically well-correlated with human judgement [217, 125].

5. *Multiple-choice / QA* formats score accuracy, F1, or exact match on classification-style tasks, common in program understanding and SE question answering [221, 218].

6. *Ad hoc* designs – formal verification pipelines, specification matching, bespoke reasoning harnesses – defy the other categories and are tied to their benchmark's unique goals [223, 222].

A Critical Reading, and a Recommended Suite Standing back from the inventory, the imbalance is unmistakable: the community optimises overwhelmingly for *test-passing*. That focus carries two costs. First, a high Pass@k can conceal bad software – code that clears its tests while being convoluted, unmaintainable, or insecure earns full marks. Second, the qualities that distinguish good engineering from merely working code – efficiency, readability, safety – are measured almost exclusively in niche datasets, leaving mainstream leaderboards blind to them.

As a corrective, Study I distils the metric inventory into a compact recommended suite for future benchmarks (Table 18), covering five dimensions: *correctness* (Pass@k), *robustness* (mutation score, fail-to-pass rate), *efficiency* (normalised efficiency index, runtime/memory), *readability* (maintainability index, cognitive complexity), and *safety* (vulnerability-detection precision/recall/F1). This suite reappears in Study III (Section 4.3), whose evaluation reports correctness alongside token cost.

RQ3 in brief. Metric choice follows task, but Pass@k and accuracy dominate overall, and execution-based evaluation – running generated code against tests – is by far the most common methodology. The near-exclusive focus on functional correctness leaves efficiency, readability, and safety largely unmeasured, which motivates the five-dimensional evaluation suite proposed here.

Table 18: Recommended Suite for Evaluating Code Intelligence Tasks

Dimension	Definition	Recommended Metrics
Correctness	The functional adherence of the code to the problem description.	*Pass@k*
Robustness	The ability of the code to handle edge cases, changes, or stress.	Mutation Score, Fail-to-Pass Rate.
Efficiency	The computational and resource performance of the generated solution.	Normalized Efficiency Index, Runtime/Memory Usage.
Readability	The ease with which human developers can understand and maintain the code.	Maintainability Index, Cognitive Complexity.
Safety	The detection of security flaws and vulnerabilities.	Vulnerability Detection Rate (e.i. Precision, Recall, F1-score)

4.1.4 RQ4: What Do the Benchmarks' Users Say Is Wrong with Them?

An Evidence-Based Approach to Criticism It would be easy – and unconvincing – to criticise benchmarks from the armchair. RQ4 instead asks the people best placed to know: the researchers who have run experiments on these datasets and written up what they found lacking. The limitations reported below are all drawn from published papers that cite and use the benchmarks in question.

Choosing the benchmarks to examine. The original plan – take the three most-cited benchmarks in each SDLC phase – collapsed on contact with the data: implementation-phase benchmarks, and code-generation ones above all, so dominate the literature that most other phases lack any dataset with a substantial critical literature. The selection criteria were therefore revised to favour datasets with (i) strong community adoption, (ii) high citation counts, and (iii) a citing literature rich enough to actually discuss their weaknesses, topped up with influential niche code-generation benchmarks to capture emerging evaluation styles. The result is a panel of 14 benchmarks: HumanEval [121], MBPP [122], CodeContests [151], SWE-Bench [242], DS-1000 [190], LiveCodeBench [192], API-Bank [127], CrossCodeEval [209], CRUXEval [219], ARCADE [167], DebugBench [235], SWE-Bench Multimodal, TESTEVAL [229], and SWT-Bench [226].

Harvesting the reported limitations. For each of the 14 benchmarks, the crawling infrastructure from Section 3.1.2 gathered every citing publication. Because the citing literature runs to thousands of papers, an LLM performed a first screening pass, flagging papers that appear to discuss benchmark shortcomings. Everything downstream of that flag was manual: one author read the flagged papers and extracted and categorised the reported limitations, and a second author independently verified the extraction – a guard against both screening misses and hallucinated content. Figure 12 depicts the pipeline.

Five Recurring Shortcomings The extracted reports converge on five categories. Table 19 maps each to the affected datasets and the papers reporting it, and Figure 13 visualises the mapping.

- **Tasks too simple, too far from practice.** The most frequently voiced complaint. Competitive-

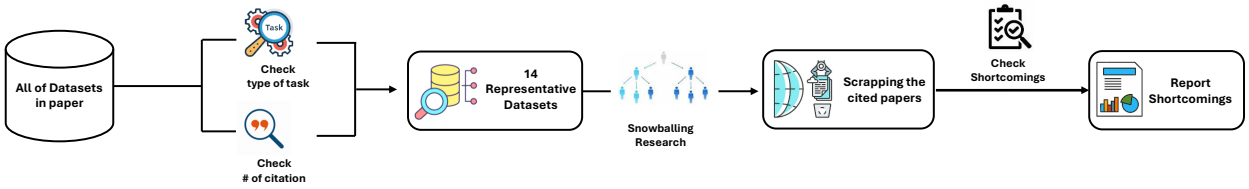


Figure 12: Pipeline for extracting community-reported benchmark shortcomings: citation crawling over 14 representative benchmarks, LLM-assisted screening, and dual-author manual extraction and validation.

programming and toy algorithmic tasks dominate, bearing little resemblance to day-to-day engineering [196, 201]; data-science benchmarks stick to basics and skip visualisation and data understanding [268, 269]; single-function and single-file framings erase the dependency structure of real projects [270]; test-generation benchmarks ask for isolated tests instead of coherent suites [230]; and Python-only scopes cap external validity [244].

- **Questionable data quality.** Some datasets lean on LLM-generated content without adequate human vetting [271, 272]; difficulty distributions skew toward short, easy items, blunting the benchmark's power to separate strong models from weak ones [273, 274]; and task descriptions are frequently terse, ambiguous, or simply wrong, so that failures measure prompt quality rather than model ability [168, 275].

- **Tests that miss too much.** Reported problems include thin coverage that lets buggy solutions pass [168, 276, 277], canonical solutions that are themselves incorrect – documented in HumanEval [168] – and the near-total absence of stress tests for runtime behaviour [202, 276].

- **Underspecified evaluation.** Several benchmarks lack a systematic evaluation process [278], evaluate on reduced subsets [227], or produce performance numbers that shift with the execution environment and hardware [279]; few pipelines even attempt to reproduce the interdependencies of real development.

- **Exposure to leakage.** Public-repository sourcing and pre-cutoff data invite overlap with training corpora [280, 245, 275], inflating scores in ways that are invisible from the leaderboard.

Why These Shortcomings Matter Each category distorts evaluation in its own way. *Simplistic tasks* let models coast on shallow pattern matching – their reasoning is never actually exercised – so scores overstate real capability [282], and the Python/single-file monoculture [149] says nothing about how a model copes with multi-language ecosystems and large structured codebases [209]. *Poor data quality* compounds this: easy-skewed task pools flatter every model, and ambiguous specifications convert prompt misreadings into apparent model failures, feeding hallucination and overconfidence [275]. *Weak tests* mean plausible-but-broken code gets certified as correct – with obvious downstream risk [293, 292] – and without stress testing, nothing is known about scalability or fault tolerance [276]. *Underspecified evaluation* pushes each research

Table 19: Limitations of LLM benchmark datasets in code intelligence

Category	Dataset	# Studies	References
Limited Task Complexity and Real-World Relevance	TestEval	10	[230]
	ARCADE		[268, 269, 281]
	HumanEval		[196, 201, 282, 149]
	MBPP		[196]
	CruxEval		[277]
	DS-1000		[188, 276, 283]
	DebugBench		[241, 284]
	SWTBench		[285]
	SWEBench		[244]
	CrossCodeEval		[270]
Data Quality	API-Bank	9	[272, 271]
	CruxEval		[286]
	LiveCodeBench		[273]
	CodeContests		[274]
	SWT-Bench		[287]
	ARCADE		[276]
	HumanEval		[168]
	MBPP		[275, 288]
	DS-1000		[289]
Evaluation Limitations	SWEBench Multimodal	8	[278]
	LiveCodeBench		[279]
	CruxEval		[220]
	ApiBank		[290, 291]
	MBPP		[35]
	HumanEval		[168]
	CodeContests		[159]
	SWTBench		[227]
Inadequate Test Coverage	LiveCodeBench	7	[292]
	HumanEval		[168, 282]
	MBPP		[277]
	CodeContests		[293, 294]
	DS-1000		[276]
	SWEBench		[245]
	TestEval		[202]
Data Leakage Risks	MBPP	4	[136]
	HumanEval		[275]
	SWEBench		[245]
	LiveCodeBench		[295, 280]
Unique Total	-	**14**	-

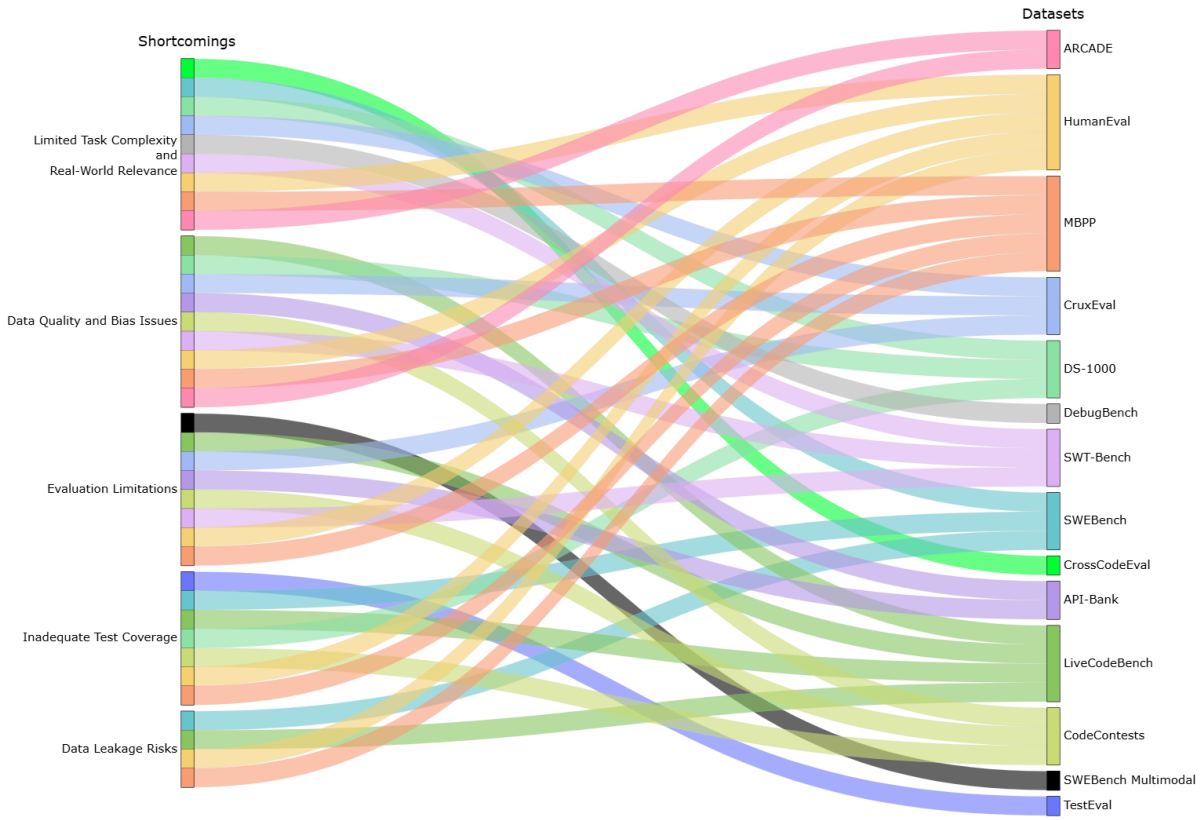


Figure 13: Community-reported limitations of LLM benchmark datasets, mapped to the representative datasets in which each was observed.

group toward its own ad hoc setup, destroying comparability across papers [278]. And *leakage* turns evaluation into a memory test [39] – a risk most acute for exactly the benchmarks everyone uses, since ubiquity is what gets a dataset into training corpora [136].

One further observation cuts across all five categories, and it is the one this thesis acts on: every benchmark in the corpus scores the model's *final output*. None can tell whether a correct answer was produced by sound reasoning or a lucky guess, and none can say at which point an incorrect answer went off the rails. The reasoning that produces the code is entirely unmeasured.

RQ4 in brief. The community's own reports converge on five recurring weaknesses – unrealistic task complexity, questionable data quality, inadequate tests, underspecified evaluation, and leakage exposure – which together inflate scores, hide failure modes, and undermine comparability. Cutting across all five: benchmarks judge only final outputs, leaving the model's intermediate reasoning invisible.

4.1.5 RQ5: What Should Come Next?

The final research question turns the diagnosis into prescriptions. Six directions emerge, each with its practical obstacles and plausible mitigations, followed by a framework that ties them together.

Bring in Other Modalities Software artefacts are not just text: interfaces are designed visually, architectures are drawn as diagrams, and behaviour is captured in execution traces. Early steps like SWE-Bench Multimodal [243] aside, benchmarks that pair code with UI mockups, architecture diagrams, or runtime traces barely exist, despite front-end work being precisely the setting where visuals and code evolve together [296, 297]. What is missing most is a standard way to score whether a code change and its corresponding interface change agree. The obstacles are real – aligning heterogeneous artefacts is expensive, and "does this look right?" is subjective – but synthetic generation from structured UI specifications, automated rendering checks, and crowdsourced annotation from front-end communities all offer ways to scale.

Specialise by Domain General-purpose benchmarks cannot answer domain-specific questions: whether a model can design a sound API, synthesise a CI/CD pipeline, or translate a design mockup into working front-end code. SWE-Bench [242] proves the value of grounding tasks in real issues but is domain-agnostic; DS-1000 [190] proves the value of specialisation but covers only data science. Back-end, DevOps, and front-end verticals all await comparable treatment. The catch is that specialisation needs scarce domain expertise and chases fast-moving tools – a benchmark pinned to one CI/CD system can be stale within months. Mining official documentation and migration guides can substitute for some expert annotation, and versioning benchmarks against specific tool releases (with per-version leaderboards) keeps them honest as tools evolve.

Scale up to Industrial Reality Even the best current repository benchmarks capture a slice of industrial maintenance, not the whole. The next generation should confront models with multi-repository microservice dependencies, legacy migrations riddled with deprecated APIs, and cross-version compatibility puzzles – and it should finally cover the low-resource languages (Go, Rust) that run critical infrastructure. Translation-based dataset generation in the style of MultiPL-T [298] helps bootstrap coverage, though translated tasks inevitably miss the idioms and runtime behaviours that make each language distinctive. The cost side is daunting – realistic multi-container environments are expensive and flaky – but pre-built lightweight containers with pinned dependencies, plus tiered *Lite*/*Full* evaluation configurations, can keep such benchmarks within reach of academic budgets.

Measure Reasoning, Not Just Answers If RQ4's cross-cutting finding is that reasoning goes unmeasured, then the most important new benchmarks are ones that measure it. Existing multi-file and cross-context designs [242, 211] gesture in this direction but provide no standard protocol for assessing whether a model can plan, track dependencies, and make sound decisions under ambiguity. Reasoning-oriented benchmarks

should demand:

- *multi-step problem solving*, where intermediate steps must be built up and integrated rather than pattern-matched;
- *cross-context comprehension*, spanning files, functions, and data sources;
- *strategic decision-making* under evolving, uncertain, or conflicting requirements;
- *agentic operation*, adapting to a dynamic environment without step-by-step supervision.

The difficulties are the familiar ones – reasoning quality is subjective, annotating intermediate steps is slow, and models can fake the surface form of reasoning without its substance. The most promising antidote is to anchor evaluation in *execution*: require the model to demonstrate its understanding against a compiler or debugger, for instance by predicting what a program will actually do – exactly the design adopted in Study II (Section 3.2). Distilling annotations from stronger reasoning models can further cut the labelling cost.

Respect the Training Cut-off Contamination-aware collection should become routine rather than exceptional: verify that every sample postdates the training cut-offs of the models under test [299]; treat GitHub- and LeetCode-sourced material with suspicion, since both feed pre-training corpora [300]; and build benchmarks that *refresh* – continuously ingesting new, post-cutoff data so they stay valid as new models ship. Since cut-offs move weekly, only automation makes this sustainable: a crawler that harvests trending repositories, filters by commit date, and pushes candidates through a standard test-generation pipeline can maintain a rolling evaluation set with minimal manual effort.

Harden the Tests Themselves Finally, the test infrastructure deserves the same rigour as the data: task inputs and metadata must not smuggle in the answer; coverage should deliberately span easy, typical, and edge-case scenarios so that generalisation, not memorised patterns, is what gets measured; and integrity should be monitored continuously, since leaks emerge over a benchmark's lifetime, not only at release. High-coverage suites are expensive to author at scale, but a pragmatic division of labour helps: let an LLM draft candidate tests, validate them automatically against reference implementations, and spend human attention only on the ambiguous or failing cases.

Putting It Together: A Benchmark Lifecycle These prescriptions cohere into a four-phase lifecycle for building and maintaining benchmarks, extending recent construction guidelines [37]: (1) *dynamic sourcing* – traceable provenance, versioning against specific library releases, and automated cutoff-aware filtering; (2) *construction and verification* – deduplication, denoising, and (for multi-modal tasks) cross-modal consistency checks such as confirming that generated code renders the intended interface; (3) *reproducible evaluation* – shipping complete containerised environments rather than bare snippets, and scoring the path to the solution (tool use, intermediate steps) alongside the solution itself; and (4) *sustainable maintenance* – continuous, automated contamination auditing and test-case refresh in place of one-shot releases.

RQ5 in brief. The next generation of benchmarks should embrace multi-modality, specialise into underserved technical domains, scale to industrial complexity and low-resource languages, target reasoning explicitly, enforce cutoff-aware and continuously refreshed data collection, and invest in test integrity – ideally organised as a single benchmark lifecycle from dynamic sourcing through sustainable maintenance.

4.1.6 What Study I Establishes, and Where It Points

Study I mapped the LLM-for-code benchmarking landscape: 156 datasets, 142 papers, 32 tasks. The map shows a field growing at remarkable speed but along narrow rails – toward Python, toward code generation, toward single-turn algorithmic tasks scored by Pass@k – with quality assurance and contamination defences lagging well behind the pace of dataset creation. Listening to the researchers who use these benchmarks surfaced five recurring shortcomings, which the study translated into concrete design prescriptions, a benchmark lifecycle, and a five-dimensional evaluation suite whose correctness-plus-efficiency frame Study III adopts.

For the thesis, the decisive output is the cross-cutting observation of RQ4: *every* current benchmark judges the final answer and ignores the reasoning that produced it. A model can be right for the wrong reasons or wrong at a single unlucky step, and no leaderboard can tell the difference – even though flawed step-by-step reasoning over program behaviour is repeatedly implicated in incorrect code. Section 4.1.5 argued that the remedy is execution-anchored, reasoning-aware evaluation. Study II, whose results follow in Section 4.2, builds precisely that: it puts state-of-the-art reasoning LLMs to work inferring program behaviour, opens their reasoning traces, and asks where, why, and how systematically they fail.

4.2 Study II: Reasoning Errors on Code-Execution Inference

4.2.1 RQ6: How Accurate Are Reasoning LLMs at Execution Inference?

Figure 14 gives the headline picture. On regular inputs, these models are genuinely strong: GPT-o4 (o4-mini) tops 97% on HumanEval+ and 98.1% on LiveCodeBench, with DeepSeek-R1 and Gemini close behind and Claude 4 trailing modestly. The ranking is not stable across conditions, though. On edge inputs, Gemini takes the lead on HumanEval+ (90.6%) while GPT-o4 leads on LiveCodeBench (91.9%); on invalid inputs, GPT-o4 dominates HumanEval+ (98.2%) but DeepSeek-R1 is best on LiveCodeBench (92.8%). The weakest configuration in the whole grid is Claude 4 on REPOEXEC edge cases, at 63.6% – a reminder that repository-scale context and boundary conditions together form the hardest setting.

Table 20 counts the failures underneath these rates, per model, dataset, and tier. Three regularities stand out. First, LiveCodeBench extracts more failures than HumanEval+ from every model – difficulty carries straight through from the program corpus to the error counts. Second, and most consequential for this thesis: *edge inputs roughly double the failure rate relative to regular inputs, for every model*. Competence on typical

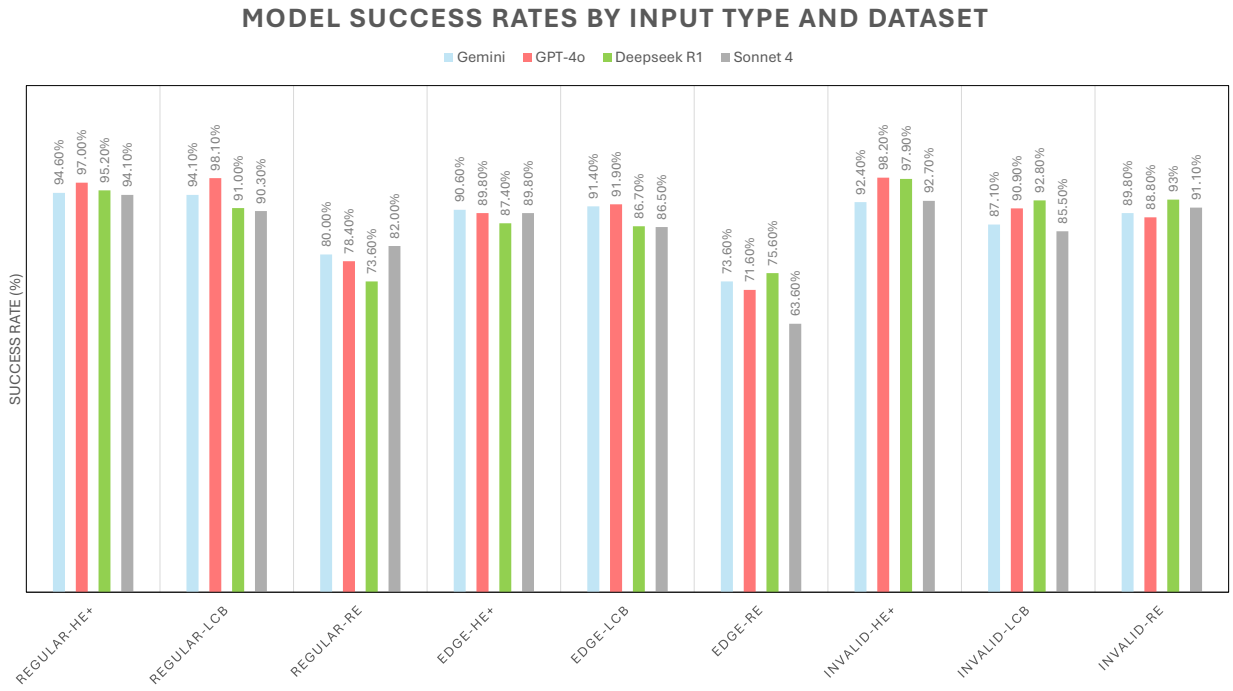


Figure 14: Success rates of the four reasoning LLMs on HumanEval+, LiveCodeBench, and REPOEXEC, broken out by input tier.

executions does not transfer to the boundaries of the input space – plausibly because boundary behaviour is both intrinsically harder to simulate and rarer in training data. Third, the models split into two behavioural profiles on invalid inputs: Gemini and Claude 4 fail at similar rates whether inputs are valid or not, while GPT-o4 and DeepSeek-R1 are markedly better at flagging invalid inputs – suggesting their training gave them stronger input-validation instincts.

Table 20: The number of unsuccessful inferences (i.e., program–input pairs) for the four reasoning models across different input types. Numbers in parentheses denote the count of unique programs within those unsuccessful cases.

Models	regular inputs			edge inputs			invalid inputs		
	HumanEval+	LiveCodeBench	REPOEXEC	HumanEval+	LiveCodeBench	REPOEXEC	HumanEval+	LiveCodeBench	REPOEXEC
Gemini	44 (26)	77 (45)	55 (20)	77 (56)	113 (84)	71 (23)	25 (23)	68 (64)	13 (7)
GPT-o4	25 (15)	25 (20)	51 (15)	84 (56)	107 (89)	66 (21)	6 (6)	48 (44)	18 (12)
Deepseek R1	39 (21)	118 (62)	66 (22)	103 (65)	175 (125)	61 (19)	7 (7)	38 (36)	10 (8)
Claude 4	48 (26)	127 (74)	45 (14)	84 (52)	178 (126)	91 (29)	24 (23)	76 (71)	15 (10)

RQ6 in brief. Across the three datasets and three input tiers, success rates span 63.6% to 98.17%. All four models handle regular and invalid inputs better than edge cases, where failures roughly double – boundary conditions are the soft spot of current execution reasoning.

4.2.2 RQ7: Where in the Code Does Reasoning Break?

Because every failed trace carries a single-statement inspection point, the failures can be mapped onto the kinds of statements at which they begin. Figure 15 shows this distribution for regular and edge inputs (invalid

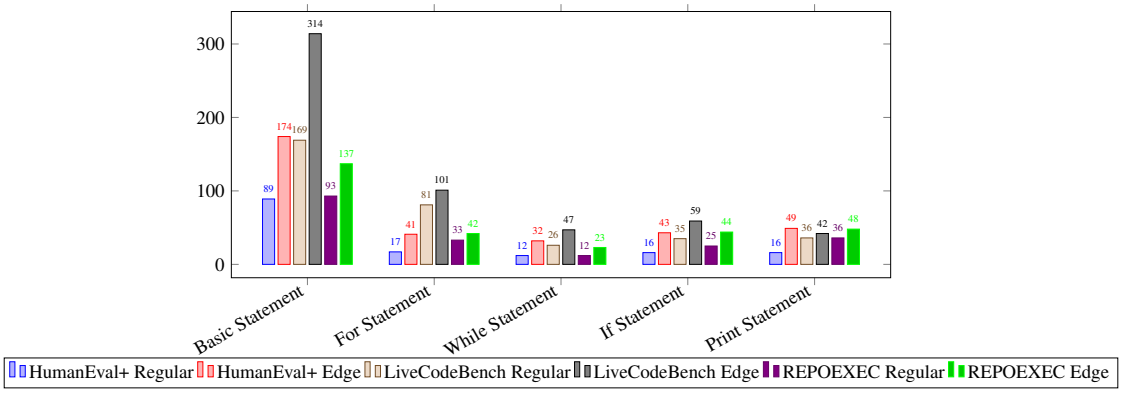


Figure 15: Distribution of inspection-point reasoning failure locations across all unsuccessful inferences for the evaluated LLMs on the regular and edge inputs of HumanEval+, LiveCodeBench, and REPOEXEC datasets. The chart categorizes the specific code locations (e.g., Basic Statement, For Statement) where the model's reasoning first diverged from the correct execution path.

inputs have no meaningful failure location and are excluded).

The most arresting result is where the failures are *not*: they do not concentrate in the exotic corners of the language. *Basic statements* – plain assignments, arithmetic updates, value substitutions – are the leading failure site everywhere, accounting for more than half of all errors: 89 regular / 174 edge failures in HumanEval+, 169 / 314 in LiveCodeBench, 93 / 137 in REPOEXEC. The mundane machinery of execution, not its complexity, is what current reasoning models most often get wrong – and the effect amplifies as code context grows.

Control flow is the clear second. Failures at *for* statements jump sharply from HumanEval+ to LiveCodeBench in both input tiers, tracking that benchmark's heavier use of iteration and index management, and loops remain a substantial failure source in REPOEXEC. *If* statements follow the same pattern – failures roughly triple between HumanEval+ and LiveCodeBench regular inputs, with REPOEXEC contributing 25 (regular) and 44 (edge) more – and *while* loops grow more moderately. Branch- and loop-heavy logic reliably stresses these models.

The oddest entry in the profile is the humble *print statement*. Producing output is not algorithmically hard, yet print statements generate a persistent slice of failures – 16 and 36 under regular inputs on HumanEval+ and LiveCodeBench, swelling to 49 and 42 under edge inputs, with REPOEXEC tracking the same curve (36 regular, 48 edge). Even when a model has simulated the computation correctly, formatting and faithfully reporting the result remains a failure mode of its own – especially under edge conditions.

RQ7 in brief. Reasoning failures begin most often at ordinary statements – assignments and arithmetic – which account for over half of all errors, with loops and conditionals the second-largest source and scaling with task difficulty. Surprisingly, even print statements contribute a persistent share of failures, exposing weaknesses in output reporting under edge conditions.

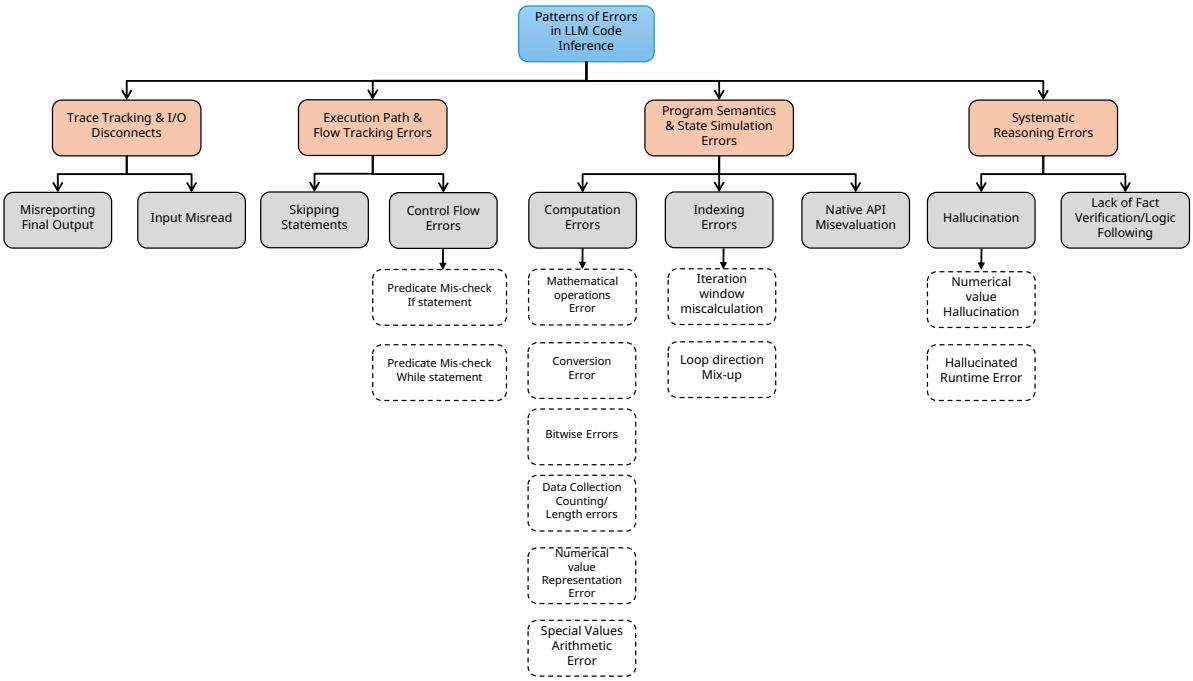


Figure 16: The nine-class taxonomy of reasoning failures in LLM code inference, organised under four parent categories.

4.2.3 RQ8: What Kinds of Failures Are These? A Nine-Class Taxonomy

Card-sorting the pattern labels (Section 3.2.3) produced a two-tier taxonomy: nine error classes under four parent categories, several refined further into subtypes. Figure 16 shows the structure; the classes are described below, each anchored by a real failure from the corpus. Following the definition of Section 3.2.2, the parent categories separate mechanical simulation failures (A to C) from reasoning failures in the narrow sense (D). The former dominate the counts; the latter are less frequent but, as the tool-augmentation experiment of Section 4.2.4 shows, are the ones a deterministic tool cannot repair.

A. Program Semantics and State Simulation Errors – local, mechanical mistakes in simulating an operation, a construct, or a library call.

1. Computation Errors – arithmetic and numeric simulation gone wrong, in six flavours:

- *Mathematical operations*: plain arithmetic slips – Gemini, on *HumanEval-130*, evaluated $2+3+1+(4/2)$ as 9 rather than 8 at iteration $i=3$, and the error contaminated every later odd-index term.
- *Conversions*: type-change mistakes – Claude, on *LiveCodeBench-80*, mis-simulated the call `bin(10000000000000000)`, botching the integer-to-binary conversion.
- *Bitwise operations*: Gemini, on *LiveCodeBench-14*, computed the XOR $(2^{64}-1)\oplus(-2^{63})$ incorrectly.

- *Counting and length*: off-by-collection-size errors – Gemini, on *HumanEval-111*, counted a 100-element list (fifty `as`, fifty `bs`) as $48+48=96$ elements.
- *Numeric representation*: floating-point precision confusion – Gemini, on *HumanEval-2*, carried the value 72999331.86348532 through its trace as 72999331.8634853214025497.
- *Special values*: `NaN`/`Inf` mishandling – Gemini, on *HumanEval-21*, evaluated $-inf+inf$ as 0 instead of `NaN`.

2. Indexing Errors – mistakes in accessing or slicing sequences:

- *Iteration window miscalculation*: on *LiveCodeBench-36*, Gemini conflated $s[13]$ and $s[14]$ while walking an array, and missed a transition as a result.
- *Loop direction mix-up*: on *HumanEval-59*, Claude read `for i in range(-11, 0, -1)` as yielding $-11,-10,\ldots,-1$; with a negative step and a stop above the start, the loop in fact yields nothing and its body never runs.

3. Native API Misevaluation – wrong beliefs about built-ins and standard libraries, from default-argument assumptions to confusing similar functions. On *HumanEval-162*, Claude could not simulate `hashlib.md5()` and substituted an incorrect hash value of the wrong kind.

B. Execution Path and Flow Tracking Errors – failures of sequencing and branching: the model takes the wrong path through the program, or silently drops steps, rather than miscomputing a value.

4. Control Flow Errors:

- *Predicate mischeck (if)*: on *LiveCodeBench-332*, Gemini logged the comparison `s[9] == s[8]` as `True` when it is not.
- *Predicate mischeck (loop)*: on *LiveCodeBench-75*, DeepSeek continued into a loop iteration that the real condition rules out, and returned accordingly.

5. Skipping Statements – necessary steps silently omitted, often as a premature leap to the answer. On *LiveCodeBench-76*, Gemini dropped the value-update statements inside a loop, and the final state never incorporated them.

C. I/O and Boundary Disconnects – failures at the edges of the computation: reading the input, or delivering the output.

6. Misreporting Final Output – the trace computes the right thing and then reports the wrong one. On *LiveCodeBench-54*, Gemini derived `"aaayyyaba"` correctly and then wrote `"aaayyyaab"` while assembling the answer. The reasoning was fine; the handoff failed.

7. Input Misread – the problem input itself is misperceived: strings misread, lengths miscounted, constraints overlooked. On *LiveCodeBench-289*, Gemini took a 28-character binary string to have length 26 and reasoned from there.

D. Systemic Reasoning Failures – epistemic breakdowns that transcend any single operation, and often sit upstream of the mechanical errors above.

8. Hallucination – output unmoored from execution:

- *Numerical value hallucination*: on *HumanEval-106*, Claude computed even-index terms faithfully up to $i = 28$, then began inventing large numbers rather than computing them.
- *Hallucinated runtime error*: on *HumanEval-36*, Gemini declared a "Time Limit Exceeded" failure for a program that is merely slow – and correct.

9. Lack of Fact Verification / Logic Following – plausible reasoning that skips a required check. On *HumanEval-59*, Claude found that 2639 divides 13195's cofactor and stopped there, never verifying that 2639 is prime (it is $7 \times 13 \times 29$); the true largest prime factor, 29, was never reached.

How do these classes distribute over models and datasets? Table 21 gives the counts, and the dominant fact is visible at a glance: *Computation Errors lead everywhere*. They account for 53 regular / 133 edge failures on HumanEval+, a much larger 138 / 343 on LiveCodeBench, and 63 / 98 on REPOEXEC. Whatever else varies, numerical simulation is the reasoning bottleneck.

The remaining classes tell more textured stories. Indexing errors rise with data-manipulation complexity (15 regular-input cases on HumanEval+ versus 56 on LiveCodeBench) but hold steady on REPOEXEC (20 regular, 23 edge). Control-flow errors scale steeply with difficulty – 4 to 31 regular, 8 to 46 edge, moving from HumanEval+ to LiveCodeBench, and 24 to 35 from regular to edge on REPOEXEC – confirming that intricate branch logic is a genuine stressor. Skipping statements stay comparatively flat across all conditions (15–38 cases). Misreported outputs spike specifically on HumanEval+ edge cases (49, against 17 for LiveCodeBench): on easy tasks the reasoning tends to survive and the *presentation* fails, whereas on harder benchmarks the failures live deeper in the reasoning itself. Input misreads surge under edge inputs everywhere – most dramatically on REPOEXEC, from 6 to 29 – reflecting the difficulty of parsing irregular, length-sensitive inputs. Native-API misevaluation is concentrated where library semantics decide the answer: frequent on HumanEval+ (17–28) and REPOEXEC (20), rare on LiveCodeBench (4–6). Hallucinations are few but grow under edge conditions, and logic-following failures expand sharply with task difficulty – 5 to 31 (regular) and 12 to 51 (edge) from HumanEval+ to LiveCodeBench, with a solid REPOEXEC presence (24 and 18) – evidence that the harder benchmarks stress logical discipline, not just mechanics.

Aggregating per model (Figure 17): Claude 4 accumulates the most failures (594), DeepSeek-R1 follows (541), and Gemini and GPT-o4 commit roughly half as many.

Table 21: Distribution of reasoning failures for each LLM according to the identified failure patterns.

Category	LLM	Regular Inputs			Edge Inputs		
		HumanEval+	LiveCodeBench	REPOEXEC	HumanEval+	LiveCodeBench	REPOEXEC
Computation Errors	Claude	16	46	13	27	101	29
	DeepSeek	20	62	19	28	114	25
	Gemini	7	20	17	34	59	21
	GPT	10	10	14	47	69	23
Control Flow Errors	Claude	2	13	7	5	20	10
	DeepSeek	-	9	12	1	11	13
	Gemini	2	6	4	2	10	4
	GPT	-	3	1	-	5	8
Hallucination	Claude	1	1	1	6	3	15
	DeepSeek	1	1	6	5	-	1
	Gemini	1	-	5	6	4	-
	GPT	-	-	3	3	2	1
Indexing Errors	Claude	6	23	3	7	16	7
	DeepSeek	4	12	7	8	5	-
	Gemini	4	18	5	3	10	9
	GPT	2	3	5	4	2	7
Input Misread	Claude	8	1	-	9	9	4
	DeepSeek	2	2	2	12	11	5
	Gemini	8	5	1	8	6	8
	GPT	-	-	3	3	13	12
Lack of Fact Verification/Logic Following	Claude	1	7	7	1	10	2
	DeepSeek	2	10	6	3	17	4
	Gemini	1	8	4	5	15	6
	GPT	1	6	7	3	9	6
Misreporting Final Output	Claude	4	9	4	3	4	5
	DeepSeek	2	12	6	13	4	6
	Gemini	6	12	8	18	5	6
	GPT	2	3	5	15	4	2
Native API Misevaluation	Claude	4	3	4	10	1	5
	DeepSeek	-	1	4	11	3	4
	Gemini	8	-	4	2	2	9
	GPT	5	-	8	5	-	2
Skipping Statements	Claude	8	23	2	15	14	14
	DeepSeek	5	9	3	9	9	3
	Gemini	3	7	6	2	1	8
	GPT	4	-	4	4	-	4

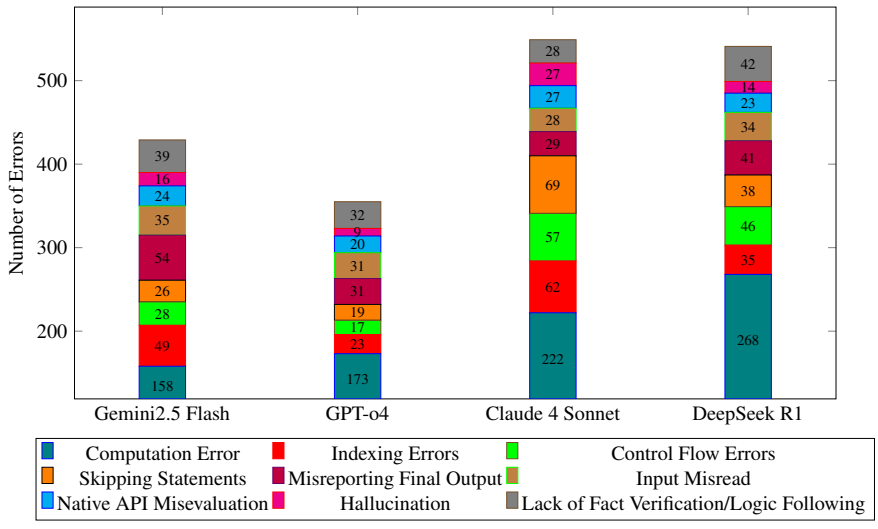


Figure 17: The distribution of error categories per LLM. Each bar represents an LLM, and each segment shows the contribution of a specific error category.

RQ8 in brief. The failures organise into nine classes – Computation, Indexing, Control Flow, Skipped Statements, Misreported Output, Input Misread, Native API Misevaluation, Hallucination, and Lack of Verification – under four parent categories spanning mechanical state simulation through systemic epistemic breakdowns. Computation Errors dominate every model and dataset, with Control Flow and Indexing Errors next.

4.2.4 Further Analyses

Four follow-up questions sharpen the results: does code complexity predict failure (4.2.4)? can the dominant error class be fixed with tools (4.2.4)? do correct answers guarantee sound reasoning (4.2.4)? and are the failures stable enough to trust a single-shot analysis (4.2.4)?

Does Complexity Predict Failure? For each of four metrics – Lines of Code (LOC), Halstead Difficulty (HD), Cyclomatic Complexity (CC), and Maintainability Index (MI) – a Mann–Whitney U test compares the programs each model solved against those it failed (Table 22).

Table 22: Mann–Whitney U test comparing code complexity metrics between tasks where LLM reasoning failed and those where it succeeded across function-level and repository-level datasets. The last two columns show the average metric values for tasks with reasoning failures and successful reasoning, respectively.

Dataset	Complexity Metric	p-value	Avg. (Failed Tasks)	Avg. (Successful Tasks)
HE+ & LCB	Lines of Code (LOC)	0.4025	11.69	10.55
	Halstead Difficulty (HD)	0.001959	2.21	2.84
	Cyclomatic Complexity (CC)	2.40×10^{-9}	2.78	4.10
	Maintainability Index (MI)	0.00015575	72.96	68.15
REPOEXEC	Lines of Code (LOC)	3.2×10^{-8}	301.10	109.80
	Halstead Difficulty (HD)	0.8198	6.60	5.34
	Cyclomatic Complexity (CC)	0.0055	7	3.6
	Maintainability Index (MI)	1.3×10^{-8}	42.05	58.01

The answer depends on scale, and at function level it is genuinely counterintuitive. On HumanEval+ and LiveCodeBench, models succeed *more* often on code with higher Cyclomatic Complexity and lower maintainability – the significant differences run in the "wrong" direction – while Halstead Difficulty is significantly *lower* among failed tasks, and LOC does not matter at all. At repository scale the relationships flip into the expected pattern: on REPOEXEC, failed tasks are significantly less maintainable, significantly more cyclomatically complex, and nearly three times longer than solved ones, while HD loses significance amid the broader architectural complexity.

What explains the function-level inversion? The confound is *what kind of code* carries high complexity scores in these datasets. Structurally complex functions there tend to embody familiar algorithmic templates – standard traversals, predictable nested loops – that saturate pre-training corpora, so models can ride pattern recognition and remembered execution shapes to the right answer despite the nominal complexity. Deceptively *simple* code cuts the other way: short, dense, template-free logic offers nothing to pattern-match,

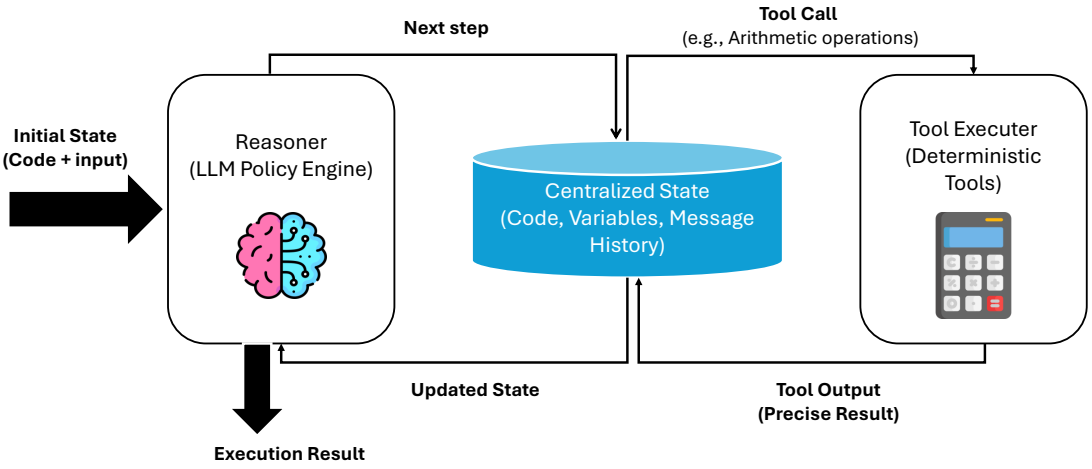


Figure 18: The tool-augmented reasoning pipeline: an LLM reasoning policy and a deterministic tool executor alternating over a centralised, append-only state.

forcing genuine statement-by-statement state simulation – exactly the regime where the taxonomy's mechanical errors (Computation, Native API Misevaluation) strike. High complexity scores flatter the models; low ones expose them.

Can Tools Fix What Reasoning Gets Wrong? The taxonomy is prescriptive as well as descriptive: each class suggests its own remedy. Computation and indexing errors call for exact calculation that an external tool can provide; control-flow and skipped-statement errors call for program analysis over ASTs and CFGs; API misevaluations are best resolved by actually calling the API; misreports and misreads yield to disciplined pre- and post-processing; hallucination and verification failures demand external checking. The experiment here tests the first and largest of these opportunities: can tool support eliminate *Computation Errors*?

The vehicle is an agentic, neuro-symbolic pipeline (Figure 18) that pairs the LLM – the semantic reasoning engine – with deterministic symbolic tools. Control moves through a two-node directed graph, a *reasoning policy* and a *tool executor*, over a centralised state that persists the source code, the evolving environment, and an append-only log of reasoning steps. On each cycle the LLM consults the state and advances the program's control flow one logical step; whenever the next step demands strict numerical accuracy, the policy binds the operands and hands off to the tool node instead of computing in-token. Quarantining arithmetic from language this way removes the autoregressive slips that plague in-context calculation, and the append-only log yields a fully inspectable trace – with room to bolt on static analysers or AST tooling later without disturbing the loop.

The evaluation set was sampled (95% confidence level) from the computation-error failures on HumanEval+

and LiveCodeBench, spanning regular and edge inputs. Table 23 reports errors before and after the pipeline: every model improves, with per-model reductions between roughly 64% and 72%, and **69%** of the sampled computation failures resolved overall.

Table 23: Impact of tool-augmented reasoning on *Computation Errors*

LLM Model	**# errors w/o tool-augmentation**	**# errors with tool-augmentation**
Gemini 2.5	82	26
GPT-o4	71	22
Claude 4 Sonnet	131	46
DeepSeek R1	155	43
Total	439	137

The 31% that survive are instructive. Residual analysis groups them under two structural limits:

- **Integration disconnects** – the tool returns the right number, but the model fails to thread it back into the ongoing reasoning, losing it to context limits or misaligning it with the surrounding program semantics;
- **Cascading logical flaws** – the "computation error" was only the visible symptom of an upstream mistake (a wrong branch, a bad loop decision) that no downstream correct value can repair.

Tool augmentation, in other words, cleanly removes isolated arithmetic failures but cannot substitute for sound control-flow tracking and state verification – those require deeper program-analysis integration.

Looking past the experiment, the same architecture points at practice. An agent that invokes tools mid-inference has obvious value for live debugging of production systems, where fragmented logs and microservice boundaries starve human debuggers of context: the LLM supplies the semantic map of the suspected execution path, while tools query observability platforms, inspect read-only state, or inject temporary logging to test each hypothesis. Study II's contribution is to show precisely which reasoning failures such a hybrid must guard against.

Right Answers, Wrong Reasons? Everything so far treats a wrong answer as the symptom of a flawed trace. The converse deserves scrutiny too: is a *right* answer evidence of a sound trace? Traces are trained to end in correct answers, not to be internally valid [29, 56], so correctness could in principle be coincidental.

To measure this, 368 traces with *correct* final answers were sampled – a size chosen for 95% confidence with a 5% margin – and drawn entirely from the edge-case tier, where superficial reasoning is least likely to survive undetected. Each trace was manually checked against the true execution.

The result: 36 of 368 (≈**9.7%**) contained genuine logical errors or clear departures from the real execution path, yet still landed on the right answer. Two readings follow. The reassuring one: over 90% of correct answers on hard edge cases rest on faithful simulation – these models are mostly right for the right reasons. The cautionary one: output accuracy alone overstates reasoning quality by a small but real margin, so

trace-level auditing has independent value. Notably, every unfaithful trace was classifiable with the existing taxonomy (Table 24) – it captures hidden failures inside successes as readily as visible ones inside failures. Control-flow errors supply nearly half the unfaithful traces, typically as *coincidental correctness*: the model mentally skips or mis-evaluates a branch that, for this particular input, happens not to affect the target variable. The scattered verification failures and hallucinations mark answers rescued by memorised patterns or lucky guesses rather than simulation.

Table 24: Distribution of hidden reasoning errors in unfaithful traces (correct final output but flawed reasoning) mapped to our proposed taxonomy.

Taxonomy Category	**Number of Instances**
Control Flow Errors	15
Lack of Fact Verification/Logic Following	8
Hallucination	5
Indexing Errors	4
Skipping Statements	3
Native API Misevaluation	1
Total Unfaithful Traces	**36**

Are the Failures Stable, or Just Noise? LLMs sample their outputs, so any single-shot failure could in principle be generative noise rather than a real deficit. A reproducibility experiment closes this loophole. From the 2,385 edge program–input pairs, 93 were sampled (95% confidence, 10% margin) from among those that had defeated all four models, and each was re-run four independent times on each model – 1,488 fresh traces. Two statistics summarise stability:

- **Error reproduction rate**: how often a re-run fails again – **80.2%**;
- **Category consistency rate**: among reproduced failures, how often the error lands in the same taxonomy class as the original – **71%**.

The interpretation is clear. When a model lacks the semantic machinery for a particular edge case, resampling does not conjure it: four times out of five the failure recurs, and when it recurs it usually takes the same form. Surface details of a trace vary run to run; the underlying weaknesses – sliding-window index tracking, complex branch evaluation – do not. The single-shot analysis of RQ6–RQ8 therefore measures systematic properties of these models, and the taxonomy's categories are stable enough to serve as a longitudinal instrument.

4.2.5 What Study II Establishes, and Where It Points

Study II opened the black box that Study I showed every benchmark leaves closed. Across 22,896 inferences – four reasoning LLMs, 477 programs, three input tiers – the models proved strong on typical inputs (62–98% accuracy) and reliably weaker on boundaries, where failures double. Annotating every failed trace at two levels of granularity produced a nine-class taxonomy whose headline is that mundane computation, not

exotic complexity, is the dominant failure mode; that harder code stresses control flow and logical discipline; and that even output reporting fails on its own. A neuro-symbolic pipeline that delegates arithmetic to a deterministic tool erased 69% of the dominant error class; a faithfulness audit found fewer than one in ten correct edge-case answers standing on flawed reasoning; and a reproduction study (80.2% recurrence, 71% category consistency) certified the failures as systematic rather than stochastic.

For the thesis, one property of these results matters above all: the failures are *localisable and correctable*. A failed trace usually goes wrong at one identifiable statement, and deterministically repairing that single step – as the tool experiment did – usually repairs the final answer. That transforms the limitations catalogued here from diagnosis into raw material: a fine-grained signal, attributable to individual steps, that a training procedure can optimise against. Study III, whose results follow in Section 4.3, builds exactly that procedure – carrying the signal to the harder, end-to-end task of repository-level program repair.

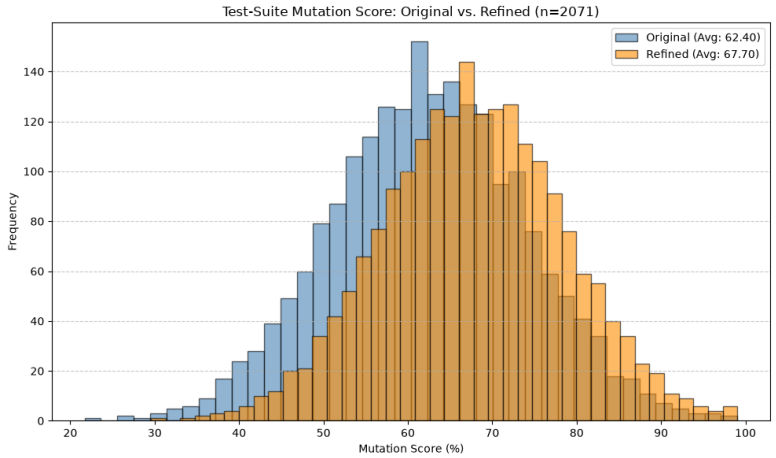
4.3 Study III: Adaptive Test Selection for RL on Issue Resolution

4.3.1 RQ11: Effectiveness of Test Refinement

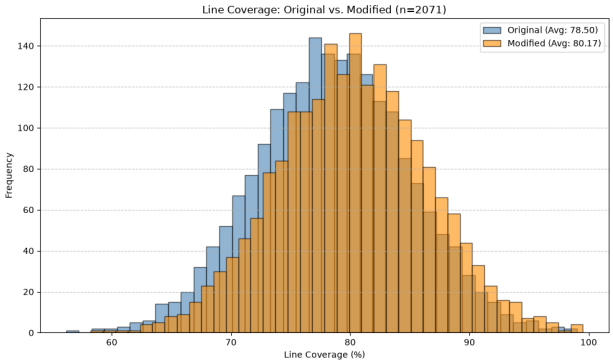
Before asking whether adaptive suites help training, a prior question must be answered: does the refinement operator actually produce better tests? If the gate of Section 3.3.2 merely swaps one test for an equivalent one, everything downstream is noise. We therefore measured the 2,071 tests that passed through refinement and its validation gate, comparing each original test against its refined replacement on three policy-independent quality criteria: *mutation score*, the fraction of seeded faults the test detects – the standard measure of a test's discriminative power (Chapter 2) – together with *line coverage* and *branch coverage* of the code under test, which measure how much of the code, and how many of its decision paths, the test exercises.

Figure 19 shows all three distributions before and after refinement. The largest movement is in mutation score (Figure 19a): the entire distribution shifts right, the mean rises from 62.4% to 67.7% – a gain of 5.3 points – and the improvement comes from thinning the weak left tail, where tests below roughly 55% mutation score become markedly rarer, rather than from a few outliers at the top. The two coverage measures move in the same direction but far more modestly, and by the same amount: line coverage (Figure 19b) rises from a mean of 78.5% to 80.2%, and branch coverage (Figure 19c) from 66.5% to 68.2%, a gain of 1.7 points on each. The shapes of the two coverage shifts differ in an instructive way. Line coverage tightens around its centre, with mass moving out of the 65–77% range and into the 80–90% range. Branch coverage, by contrast, is nearly unchanged across the lower half of its distribution; its gain is concentrated at the upper end, where refined tests populate the 78–88% bins that original tests largely left empty – refinement does not rescue tests that exercise few decision paths, but it pushes already-reasonable tests toward more complete path coverage.

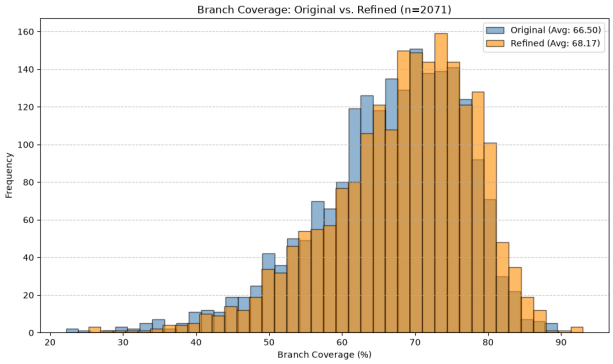
The asymmetry between the mutation-score gain and the coverage gains is itself informative. Refinement's mandate is to rewrite tests that are over-specific, incidental, or entangled with unrelated behaviour – defects of *discrimination*, not of reach – so it should move mutation score more than either coverage measure, and it



(a) Mutation score.



(b) Line coverage.



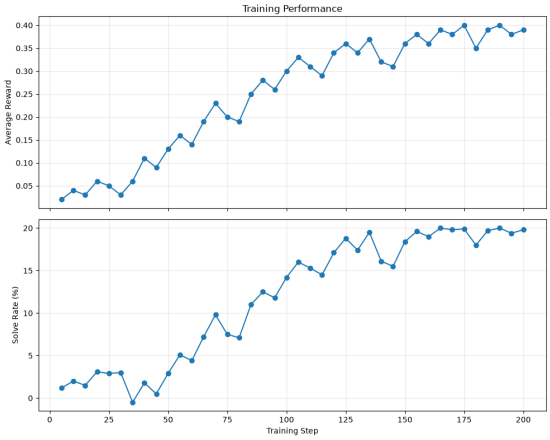
(c) Branch coverage.

Figure 19: Quality of the $n = 2{,}071$ tests that passed through refinement, before (blue) and after (orange) the operator: distributions of (a) test-suite mutation score, (b) line coverage, and (c) branch coverage.

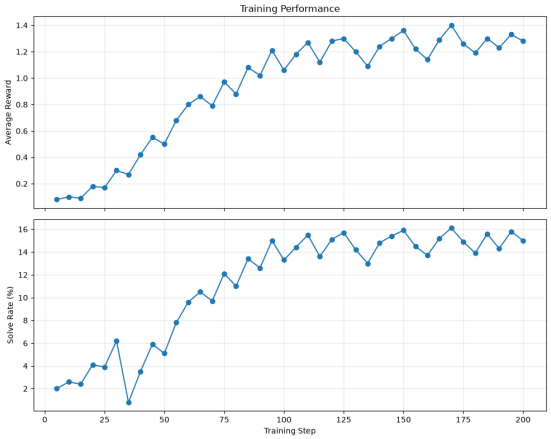
does, by a factor of three. A refined test typically touches roughly the same code as its original; what changes is how sharply it distinguishes a correct implementation from an incorrect one. The result also bears directly on the circularity concern raised in Chapter 5: although refinement is *triggered* by a policy-conditioned signal (a stalled test), what it produces is better by three policy-*independent* standards. The operator is not carving easier targets for the current policy; it is repairing the estimator.

4.3.2 RQ9 and RQ10: Training Dynamics across Five Conditions

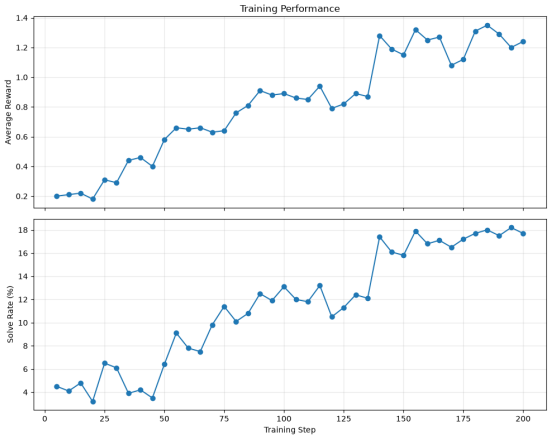
Figure 20 shows the training dynamics of the five conditions defined in Section 3.3.3 – SWE-Master RL, RL with Test Selection, RL with Predefined Tests, and RL with Dynamic Test Refinement under each of the two trigger rules – with each panel stacking average training reward over instance solve rate for the same 200 optimiser steps.



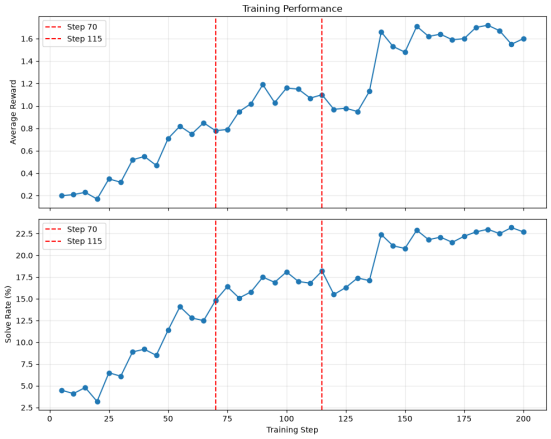
(a) SWE-Master RL (binary reward).



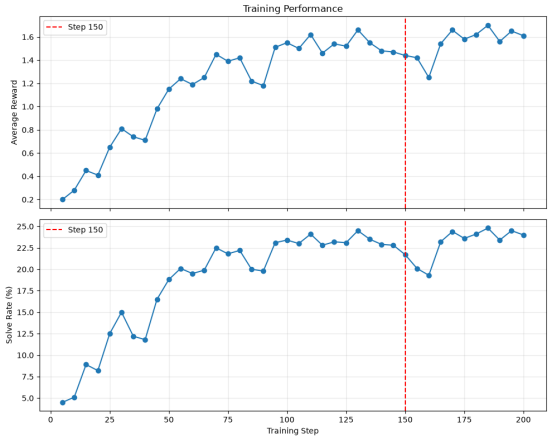
(b) RL with Test Selection.



(c) RL with Predefined Tests.



(d) Dynamic Test Refinement, reward-band trigger.



(e) Dynamic Test Refinement, reward-trend trigger.

Figure 20: Training dynamics of the five experimental conditions over 200 optimiser steps. Each panel shows average training reward (top) and training solve rate (bottom): (a) the SWE-Master baseline under the binary reward of Equation (3); (b) the dense reward with test selection over original suites; (c) test selection over suites refined offline before step one; (d) the full method with refinement waves fired by the reward-band rule – dashed lines mark the waves at steps 70 and 115; (e) the full method with waves fired by the reward-trend rule – the dashed line marks the wave at step 150.

A caution about the reward axes. Reward magnitudes are *not* comparable between the baseline and the other conditions, and we draw no conclusion from their difference. The baseline optimises the binary reward, bounded on $[0, 1]$ by construction, while the remaining four conditions optimise Equation (5), which is scaled by w_s – the vertical axes measure different quantities. The comparable quantity is the *solve rate*, computed identically in every run as the fraction of rollouts resolving the complete fail-to-pass set of their instance.

The dense reward opens faster; selection alone does not finish higher. Every dense-reward condition escapes the flat opening that pins the binary baseline (Figure 20a) below 5% solve rate for its first 45 steps, and Section 3.3.1 says why: early training is the regime in which every rollout in a group fails, so the binary reward yields $\sigma_G = 0$, the group is discarded by the variance filter, and the step is wasted, whereas a continuous reward distinguishes among failing rollouts and restores gradient. The selection-only condition (Figure 20b) shows what that advantage is worth on its own – and what it is not. It rises earlier than the baseline but climbs more slowly, crossing 15% solve rate only around step 100 and settling near 15% by step 200 (peaking at 16.1%), several points *below* the baseline's terminal 19.8%. Narrowing the acceptance criterion buys early gradient, but on the original, never-repaired suites the policy runs into the same defective tests that stall every other condition, and selection alone has no way to fix them.

Dynamic refinement leads, and the trend trigger leads decisively. The two full-method conditions dominate on both speed and destination, and they separate from each other by trigger rule. Under the reward-band trigger (Figure 20d) the run reaches 15% at roughly step 70, plateaus between 15% and 18% through its two refinement waves at steps 70 and 115, and then surges – from 17.1% at step 135 to 22.4% at step 140 – as the refined tests produced for stalled instances enter the scored suites, finishing at approximately 22.7%. Under the reward-trend trigger (Figure 20e) the run is faster still: it crosses 15% by step 30, 20% by step 55, and 22.5% by step 70, then holds a plateau between 23% and 24.5% that the band-trigger run only reaches after its late surge. Its single refinement wave, at step 150, produces a visible transient – solve rate dips from 21.7% to 19.3% over the next two evaluations as newly refined, harder tests enter the suites – followed by recovery to 24–25% and a terminal rate of approximately 24.0%, the highest of all five conditions and more than four points above the baseline. Relative to the baseline's step 100, the trend-trigger run reaches a 15% solve rate in under a third of the optimiser steps, and therefore of rollout compute.

The difference between the two triggers is instructive. The band rule fires only when reward has settled near 0.25, which happens twice and late; until then the policy is left to exhaust the original tests on its own. The trend rule keys on the reward's derivative rather than its level, so it can fire at whatever reward the policy has reached when progress stalls – and, equally, it stays quiet while the policy is still improving, which is why the trend-trigger run shows one wave rather than two. Timing refinement to stalls in progress, rather than to a fixed reward level, appears to be what lets the trend-trigger run climb early and plateau high.

Offline refinement is not enough. The predefined-tests condition (Figure 20c) starts from strictly better suites – every improvement of Section 4.3.1, delivered at step zero – yet trains worse than either dynamic

condition at every point after the opening steps, finishing at approximately 17.7%, below even the binary baseline. Its curve mirrors the baseline's shape (a slow open, a long climb, a late plateau) rather than the dynamic runs' stepwise profile. This is the study's central ablation speaking: the predefined and dynamic conditions consume the same refiner, so their separation – five to six percentage points of terminal training solve rate – measures the value of *when* refinement happens, not whether it happens. A suite refined against a step-zero policy is stale by the time the policy has learned enough to be stopped by different tests; refinement pays when it is aimed at the tests the *current* policy is stuck on. One caveat attaches to this comparison: because the predefined arm's fail-to-pass sets are the refined ones from the outset, its solve-rate curve is computed against a slightly different (and, by Section 4.3.1, slightly *harder*) target than the baseline's, so its ordering relative to the binary baseline should be read with more caution than its ordering relative to the dynamic conditions.

The dynamic conditions also display larger step-to-step variance in solve rate than the others, and the transient after each refinement wave is the visible cause: the refinement operator alters the scored test set for stalled instances between steps, so consecutive measurements are not taken against an identical target. The predefined and selection-only arms, whose suites never change during training, show correspondingly smoother curves.

What the training curves cannot say is whether a higher training solve rate survives contact with a held-out benchmark scored on original tests. The next section addresses that.

4.3.3 RQ12: Held-Out Evaluation on SWE-bench Verified

Table 25 reports resolve rate on the 500-instance SWE-bench Verified benchmark – evaluated, as stipulated in Section 3.3.3, against the original and never-refined test suites – together with the median number of tokens consumed on instances the model resolved, for the supervised checkpoint and all five trained conditions.

Table 25: Held-out evaluation of 4B policies on SWE-bench Verified, against original (never-refined) test suites. Best value in each column in bold.

Model	Resolve rate	Median tokens (solved instances)
SWE-Master 4B-SFT	25.4%	18,754
SWE-Master 4B-RL	27.2%	18,849
RL with Test Selection	26.4%	**18,113**
RL with Predefined Tests	26.8%	18,843
RL with Dynamic Test Refinement (reward-band trigger)	27.8%	18,458
RL with Dynamic Test Refinement (reward-trend trigger)	**28.2%**	18,493

Resolve rate. Both dynamic-refinement conditions surpass every other system on the held-out benchmark, and their ordering matches the training curves. The reward-trend trigger attains 28.2% – 2.8 points above the supervised checkpoint and a full point above the reinforcement-learned baseline – and the reward-band

trigger 27.8%. The predefined-tests condition (26.8%) and the selection-only condition (26.4%) both improve on the supervised checkpoint by 1.0–1.4 points but fall 0.4–0.8 points *short* of the binary baseline. The picture is consistent across training and evaluation: the dense reward and selection curriculum alone, or refinement delivered offline, are not sufficient to beat the published recipe on resolve rate; refinement delivered in-loop is.

Token cost. On the efficiency axis the ordering is different, and revealing. The baseline's reinforcement learning purchases its 1.8-point gain over the supervised checkpoint at a slight *increase* in consumption, from 18,754 to 18,849 median tokens. Every dense-reward condition with the token penalty of Equation (5) reverses that: the selection-only condition is the most frugal of all, at 18,113 tokens – 3.9% below the baseline – and the two dynamic conditions sit at 18,458 and 18,493, 2.1% and 1.9% below the baseline respectively. The exception is the predefined-tests condition, which at 18,843 tokens is indistinguishable from the baseline despite optimising the same penalised reward; its harder step-zero suites appear to have absorbed the efficiency the penalty would otherwise have bought.

Reading the two axes together. The five conditions decompose the method's contribution cleanly. Test selection with the dense reward supplies the *efficiency*: it produces the cheapest policy in the comparison, but not a more accurate one. In-loop refinement supplies the *accuracy*: it is what lifts resolve rate above the baseline, at a modest cost in tokens relative to selection alone (roughly 350–380 tokens per solved instance) that still leaves both dynamic conditions well below the baseline. The trend trigger is the better of the two on resolve rate (28.2% against 27.8%) at a negligible token difference (35 tokens), so it is the configuration the thesis reports as its full method. The result the thesis claims, then, is that the full method improves on the published recipe on both axes simultaneously – more issues resolved, fewer tokens per resolution – rather than trading one against the other, which is exactly the behaviour the reward of Equation (5) was designed to produce.

Study III in brief (RQ9 to RQ12). The refinement operator demonstrably improves the tests it touches: across 2,071 refined tests, mean mutation score rises by 5.3 points (62.4% to 67.7%) and mean line and branch coverage by 1.7 points each, by policy-independent measurement. In the five-condition training comparison, every dense-reward condition escapes the binary baseline's flat opening, but only in-loop refinement finishes higher: the reward-trend trigger reaches a 15% training solve rate in under a third of the baseline's optimiser steps and converges at approximately 24% against the baseline's 19.8%, while selection alone ($\sim$15%) and offline refinement ($\sim$17.7%) both finish below it. On held-out SWE-bench Verified the trend-trigger method leads at 28.2% resolved against the baseline's 27.2% while consuming 1.9% fewer tokens; selection alone yields the cheapest policy (3.9% fewer tokens) but not a more accurate one, and offline refinement yields neither.

4.3.4 What Study III Establishes

Study III completes the arc. Study I showed that the community measures coding models by their final outputs alone; Study II showed that the failures behind those outputs are localisable, and that repairing one faulty step repairs the result. Study III carried that principle into training. Where the standard recipe compresses an entire agentic rollout into one bit, the method here scores it per test, lets a policy-responsive scheduler decide which tests define that score at each RL step, and repairs tests that prove defective – with a container-validated gate ensuring that repair removes verifier error instead of introducing it. All of this happens on the reward side of the loop: the agent's prompt and observation space are untouched, and it sees only the reward.

The empirical case rests on three legs. Refinement measurably improves the tests it produces, by mutation score and by line and branch coverage – criteria the policy has no hand in. The five-condition training comparison decomposes the method: the dense reward and selection curriculum supply early gradient and the cheapest policy, but not a higher ceiling; in-loop refinement supplies the ceiling, with the separation from the baseline concentrated exactly in the early regime where the theory predicts the binary reward yields no gradient; and the predefined-tests ablation shows that the same refined tests delivered statically train worse than delivered adaptively, so the timing of refinement, not merely its output, carries value – a conclusion the two trigger rules sharpen further, since firing on stalled progress outperforms firing on a fixed reward level. Held-out evaluation preserves every one of these orderings on original, never-refined tests, with the full method exceeding a strong reinforcement-learned baseline on resolve rate and token cost at once. All of these figures come from a single training run and a single evaluation run per condition, and Section 5.3 explains why they should be treated as preliminary: the held-out margins are not yet separable from noise under the tests performed, and the comparison is at matched optimiser steps rather than matched rollout compute.

What the study does establish, and what matters most for the thesis, is the feasibility of the underlying idea: the verifier that defines a coding agent's reward need not be a fixed artefact inherited from dataset construction. It can be treated as an estimator of a latent correctness objective – an estimator the training loop is allowed to improve while it learns.

5 Discussion

No empirical result is stronger than the validity of the process that produced it. This chapter examines, for each study in turn, the threats to that validity and the limitations that bound what the results support. The studies face different kinds of threat – coverage and classification bias for a literature survey, annotation subjectivity and generalisability for an empirical analysis, and statistical strength and unimplemented components for a training method – so they are treated separately.

5.1 Study I: Threats to Validity

External validity. The survey's scope is intentionally narrow in one dimension: only benchmarks created in the LLM era (from early 2020) and aimed at LLM evaluation are included. Valuable legacy datasets – Defects4J [267], CodeSearchNet [301], QuixBugs [302], CoNaLa [303] – fall outside this scope; they remain in active use as baselines but are already well covered by earlier surveys [33, 34], and mixing them in would blur the picture of how the community evaluates generative models specifically.

Internal validity. Classifying papers by title, abstract, and full text involves judgement, and judgement invites bias. The dual-independent-review protocol (95% agreement, $\kappa = 0.89$) with third-party arbitration was designed to contain this, but some residual subjectivity in category assignments is unavoidable.

Construct validity. Two questions bear on whether the survey measures what it claims. Coverage: did the search find everything relevant? Casting the net across four major databases (2,377 initial records), snowballing in both directions, and manually searching flagship venues reduces – but cannot eliminate – the chance of missed datasets. Depth: is each study characterised fairly? Analysing every paper along four fixed axes (statistics, construction and quality, evaluation, shortcomings) enforces consistency. A final caveat concerns the included arXiv preprints, which had not completed peer review at the time of writing; the quality checklist of Section 3.1.2 filters them, but filtered is not refereed.

5.2 Study II: Threats to Validity

External validity. The study is Python-only – a defensible choice given Python's dominance in both research and the benchmark landscape (Section 4.1), but strict typing, manual memory management, and concurrency could reshape the error profile in other languages. The methodology itself carries over unchanged, so replication in Java, C++, or JavaScript ecosystems is straightforward future work. Time is the second threat: as reasoning models grow more agentic and tool-native, the *prevalence* of individual classes will shift – future architectures may design away Computation Errors or API misevaluation entirely. The taxonomy is offered as a baseline instrument for tracking exactly such shifts, not as a fixed catalogue of how models will always fail.

Internal validity. Manual annotation is judgement-laden. The mitigations were procedural: written guidelines per category, three independent annotators, consensus reconciliation, and reported pre-consensus reliability (Fleiss' $\kappa = 0.875$). One irreducible source of variation remains: while the inspection point is uniquely defined (earliest state divergence), the amount of trace an annotator reads before assigning a *pattern* label is not fixed, and different readers may frame borderline cases differently – another reason the labels were triple-annotated and reconciled.

Construct validity. Prompt wording, dataset formatting, and the execution environment all potentially colour the results. Several prompt formulations were piloted, and the one producing the most consistent cross-model behaviour was adopted for the full study.

5.3 Study III: Limitations and Threats to Validity

Threats to validity. Study III should be read as a preliminary investigation rather than a settled result. Every number reported for it comes from a single execution of the pipeline: each of the five conditions was trained once, and each trained policy was evaluated once on SWE-bench Verified. Both stages are stochastic. RL training samples rollouts from the policy, the refiner is itself an LLM whose output varies between calls, and the held-out evaluation samples agent trajectories, so a repeated run of the same configuration would not reproduce these figures exactly and could shift them by amounts comparable to the margins in Table 25. Time and compute constraints made repetition infeasible within the scope of this thesis: each condition costs 200 optimiser steps of 64 agentic rollouts on eight H100 GPUs, with container-based validation for every rollout. The correct protocol is nonetheless clear, and it is the study's most important piece of unfinished work: several seeds per condition, with means and confidence intervals reported in place of single-run values, together with a paired per-instance significance test on the held-out benchmark. Until that is done, the ordering among conditions is best treated as a hypothesis the present runs support, not as an established effect, and the specific solve rates and token counts as indicative rather than definitive.

Incomplete ablations. The five-condition comparison of Section 4.3.2 settles the study's central ablation – offline versus in-loop refinement – and separates the contribution of selection from that of refinement, but two gaps remain. The dense reward and the selection curriculum are never separated from each other, since no condition uses the dense reward over the *full* fail-to-pass set without selection; and all conditions are compared at matched optimiser-step count rather than matched *rollout compute*, which slightly favours the refinement arms, since refiner calls and gate validations consume compute the baseline does not spend. A compute-matched replication, with a dense-reward-only arm, remains to be done. The two trigger rules were likewise compared at a single setting each – one band width, one window length – so the advantage of the trend rule over the band rule has not been shown to be robust to those hyperparameters.

Unimplemented components. Two elements the design calls for remain unbuilt. Candidate tests are not yet screened for flakiness by repeated execution against the gold patch – an inexpensive check that would remove an obvious source of reward noise. And the refiner operates in a checkout of the *unpatched* repository, which lets it verify the fail-on-buggy condition by execution but leaves it unable to verify the pass-on-gold condition, the one criterion on which it must currently rely on inference alone; supplying a gold-applied checkout is the natural remedy.

Circularity in selection. The selection operator conditions on the current policy's rollouts, which risks a self-confirming criterion: tests may be selected because the policy fails them for incidental reasons. The intended composition is for policy-conditioned selection to choose only among tests that have already passed a policy-*independent* quality filter, such as a mutation-based discriminative-power criterion in the spirit of MIST-RL [96]. Making that composition explicit, and measuring the discriminative power of refined tests against the policy's own failed patches as a mutant pool, is the subject of ongoing work.

6 Conclusion and Future Work

6.1 Summary of the Thesis

This thesis set out from a simple observation: improving LLMs for coding tasks is not one problem but a pipeline of three – measuring what models can do, understanding why they fail, and converting that understanding into better models. The three studies of this thesis instantiate that pipeline, and each produced the finding that motivated the next.

Study I – Benchmarking (Sections 3.1 and 4.1). We conducted a systematic survey of 156 benchmark datasets from 142 papers spanning 32 coding tasks published between January 2020 and June 2025. The analysis mapped a rapidly growing but uneven landscape: benchmark creation has surged since 2023, yet the ecosystem is dominated by Python (77% of datasets), single-function algorithmic tasks, GitHub-sourced data (46%), and the *Pass@k* metric, while 43% of datasets apply no quality control and 86 of 156 take no measure against data contamination. From the literature that actually uses these benchmarks, we distilled five recurring shortcomings – limited complexity and realism, weak data quality, inadequate test coverage, narrow evaluation, and leakage risk – and translated them into concrete future directions, a unified benchmarking framework, and a recommended five-dimensional evaluation suite. The finding that carried forward: benchmarks score only final outputs and leave intermediate reasoning unmeasured.

Study II – Analysing (Sections 3.2 and 4.2). We opened that black box with the first white-box study of LLM reasoning on code-execution inference. Across 22,896 inferences by four state-of-the-art reasoning LLMs on 477 programs with regular, edge, and invalid inputs, the models reached 62–98% accuracy but failed roughly twice as often on edge cases. Annotating every failed trace down to the earliest diverging statement yielded a nine-class error taxonomy under four parent categories, with *Computation Errors* dominant everywhere. A tool-augmented neuro-symbolic pipeline corrected 69% of those dominant errors; a faithfulness audit showed only 9.7% of correct edge-case predictions rest on flawed traces; and a reproducibility study (80.2% error reproduction, 71% category consistency) established that the failures are systematic. The finding that carried forward: reasoning failures are localisable, and fixing the single faulty step usually fixes the output.

Study III – Fine-tuning (Sections 3.3 and 4.3). We carried that localisation principle into reinforcement learning for repository-scale issue resolution, where the standard recipe compresses an entire agentic rollout into a single bit and inherits its test suite, unexamined, from dataset construction. Study III instead treats the reward-defining suite as adaptable: a dense test-level reward with a regression guard and a token penalty restores group variance in exactly the regime where the binary reward provably yields none; an in-loop, policy-conditioned curriculum decides at the end of each RL step which tests are executed and scored during validation; and a refinement operator rewrites stalled tests behind a gold-patch-validated gate that admits a

candidate only if it fails on the buggy repository with a genuine assertion failure and passes once the gold patch is applied. The refinement operator demonstrably repairs the estimator it targets: across the 2,071 tests it touched, mean mutation score rose by 5.3 points and line and branch coverage by 1.7 each, under criteria the policy has no hand in. At 4B scale on 800 instances across 706 repositories, and compared across five conditions, the full method reached a 15% training solve rate in under a third of the optimiser steps of the binary baseline and converged highest, at approximately 24% against the baseline's 19.8% – while selection alone finished lower than the baseline, and the central ablation showed that delivering the same refined tests offline at step zero trains *worse* than delivering them dynamically, so the value lies in policy-conditioned timing, not merely in better tests. Of the two rules for deciding when to refine, firing on stalled reward progress outperformed firing on a fixed reward level. On held-out SWE-bench Verified, scored against original never-refined suites, the full method exceeded the reinforcement-learned baseline at 28.2% resolved against 27.2% while using 1.9% fewer tokens on solved instances; selection alone produced the cheapest policy but not a more accurate one. The honest reading is that these are single-run results from a stochastic pipeline: the ordering is consistent across training and held-out evaluation, but repeated runs across seeds, and a paired significance test, are needed before it can be called established.

6.2 Implications

Beyond the individual results, the thesis argues for a way of working: treat benchmarking, failure analysis, and post-training as one connected loop rather than three separate literatures. Benchmarks designed without regard for what analyses need (e.g. intermediate reasoning) cannot expose the failures that matter; analyses that stop at taxonomies leave their insight unexploited; and post-training that optimises generic signals wastes the precision that careful analysis can provide. Each study of this thesis strengthens the next stage of the loop, and the loop itself – benchmark, analyse, fine-tune – is reusable for capabilities beyond code.

A second implication is narrower but sharper, and it emerges from Study III. In verifiable-reward training we habitually speak of the verifier as though it were the objective. It is not: it is an estimator of an objective we cannot observe, and like any estimator it has error, which in this domain is measurable, asymmetric, and persistent per instance. Once the verifier is understood as an estimator rather than an oracle, improving it during training stops looking like reward hacking and starts looking like what it is – reducing the error of the measurement while the objective itself stays fixed. The gate that validates each refined test against the gold patch is what keeps that distinction real rather than rhetorical.

6.3 Future Work

Several directions follow naturally from this thesis. First, strengthening Study III's empirical case: replicating the five-condition comparison at matched rollout compute rather than matched optimiser steps, adding a dense-reward-only arm to separate the reward from the selection curriculum, sweeping the band width and window length of the two refinement triggers, adding seeds and scales, and completing the paired per-instance significance test on the held-out benchmark. Two design components also remain to be built: flaki-

ness screening of candidate tests by repeated execution against the gold patch, and a gold-applied checkout that would let the refiner verify the pass-on-gold condition by execution rather than inference. A deeper open question is the circularity of policy-conditioned selection, whose intended remedy is to filter candidates by a policy-independent criterion such as mutation-based discriminative power before selection ever sees them. Second, extending Study II beyond Python to ecosystems with strict typing, manual memory management, and concurrency, and re-running the taxonomy on successive model generations to track how the error distribution shifts as models become more agentic. Third, realising the benchmark directions of Study I – particularly dynamic, cutoff-aware benchmarks and reasoning-aware evaluation protocols that score the process, not only the output. Finally, the learned test-selection idea of Study III generalises beyond repair: any code-agent loop that consumes executable feedback (test generation, refactoring validation, regression triage) could benefit from a policy that learns which signals are worth reading.

Together, the three studies form a single arc – benchmark, analyse, and fine-tune LLMs for coding tasks – and each step of that arc has been designed so that its output is precisely the input the next step needs.

Bibliography

[1] M. Abdollahi, R. Zhang, N. Shiri Harzevili, J. Shin, S. Wang, and H. Hemmati, "Surveying the benchmarking landscape of large language models in code intelligence," *ACM Transactions on Software Engineering and Methodology*, 2026.

[2] Y. Hu *et al.*, "A survey of benchmarks for evaluating large language models in software engineering," *arXiv preprint*, 2024.

[3] W. Jin *et al.*, "LLMs and LLM-based agents in software engineering," *arXiv preprint*, 2024.

[4] M. Abdollahi, K. R. Tasnia, S. K. Saha, J. Yang, S. Wang, and H. Hemmati, "Demystifying errors in LLM reasoning traces: An empirical study of code execution simulation," *ACM Transactions on Software Engineering and Methodology*, 2026.

[5] A. Vaswani, N. Shazeer, N. Parmar, J. Uszkoreit, L. Jones, A. N. Gomez, L. Kaiser, and I. Polosukhin, "Attention is all you need," in *NeurIPS*, 2017.

[6] A. Radford, J. Wu, R. Child, D. Luan, D. Amodei, and I. Sutskever, "Language models are unsupervised multitask learners," *OpenAI Technical Report*, 2019.

[7] T. Brown, B. Mann, N. Ryder, *et al.*, "Language models are few-shot learners," in *Advances in Neural Information Processing Systems (NeurIPS)*, 2020.

[8] M. Chen, J. Tworek, H. Jun, *et al.*, "Evaluating large language models trained on code," *arXiv preprint arXiv:2107.03374*, 2021.

[9] B. Rozière, J. Gehring, *et al.*, "Code llama: Open foundation models for code," *arXiv preprint arXiv:2308.12950*, 2023.

[10] R. Li, L. B. Allal, Y. Zi, *et al.*, "Starcoder: May the source be with you!," *arXiv preprint arXiv:2305.06161*, 2023.

[11] D. Guo, Q. Zhu, D. Yang, *et al.*, "Deepseek-coder: When the large language model meets programming," *arXiv preprint arXiv:2401.14196*, 2024.

[12] J. Austin, A. Odena, M. Nye, *et al.*, "Program synthesis with large language models," *arXiv preprint arXiv:2108.07732*, 2021.

[13] M. Schäfer *et al.*, "An empirical evaluation of using large language models for automated unit test generation," *IEEE Transactions on Software Engineering*, 2024.

[14] S. Kang *et al.*, "Large language models are few-shot testers," *arXiv preprint*, 2023.

[15] C. S. Xia *et al.*, "Automated program repair with large language models," *arXiv preprint*, 2024.

[16] N. Jiang *et al.*, "Impact of large language models on automated program repair," *arXiv preprint*, 2023.

[17] H. Zhu *et al.*, "A survey of benchmarks for software defect detection," *arXiv preprint*, 2024.

[18] S. Koohestani *et al.*, "BenchScout: A search tool and discoverability study for AI4SE benchmarks," *arXiv preprint*, 2024.

[19] J. Yang, C. E. Jimenez, A. Wettig, K. Lieret, S. Yao, K. Narasimhan, and O. Press, "SWE-agent: Agent-computer interfaces enable automated software engineering," in *Advances in Neural Information Processing Systems (NeurIPS)*, 2024.

[20] Y. Zhang, H. Ruan, Z. Fan, and A. Roychoudhury, "AutoCodeRover: Autonomous program improvement," in *ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA)*, 2024.

[21] C. S. Xia, Y. Deng, S. Dunn, and L. Zhang, "Agentless: Demystifying LLM-based software engineering agents," *arXiv preprint arXiv:2407.01489*, 2024.

[22] D. Hendrycks, S. Basart, S. Kadavath, *et al.*, "Measuring coding challenge competence with APPS," in *NeurIPS Datasets and Benchmarks Track*, 2021.

[23] S. Lu, D. Guo, S. Ren, *et al.*, "CodeXGLUE: A machine learning benchmark dataset for code understanding and generation," *arXiv preprint arXiv:2102.04664*, 2021.

[24] F. Cassano, J. Gouwar, D. Nguyen, *et al.*, "MultiPL-E: A scalable and polyglot approach to benchmarking neural code generation," in *IEEE Transactions on Software Engineering*, 2023.

[25] T. Liu, C. Xu, and J. McAuley, "RepoBench: Benchmarking repository-level code auto-completion systems," *arXiv preprint arXiv:2306.03091*, 2023.

[26] Y. Ding, Z. Wang, W. Ahmad, *et al.*, "CrossCodeEval: A diverse and multilingual benchmark for cross-file code completion," *arXiv preprint arXiv:2310.11248*, 2023.

[27] N. L. Hai *et al.*, "REPOEXEC: A benchmark for repository-level code execution reasoning," *arXiv preprint*, 2024.

[28] C. E. Jimenez, J. Yang, A. Wettig, *et al.*, "SWE-bench: Can language models resolve real-world github issues?," *ICLR*, 2024.

[29] A. Gu, B. Rozière, H. Leather, A. Solar-Lezama, G. Synnaeve, and S. Wang, "CRUXEval: A benchmark for code reasoning, understanding and execution," in *International Conference on Machine Learning (ICML)*, 2024.

[30] J. Chen, X. Hu, Z. Liu, *et al.*, "REval: Evaluating code reasoning of large language models," *arXiv preprint*, 2024.

[31] N. Jain, K. Han, A. Gu, *et al.*, "LiveCodeBench: Holistic and contamination free evaluation of large language models for code," *arXiv preprint arXiv:2403.07974*, 2024.

[32] H. Zhou *et al.*, "LessLeak-Bench: Quantifying data leakage in code benchmarks," *arXiv preprint*, 2024.

[33] X. Hu, F. Niu, J. Chen, X. Zhou, J. Zhang, J. He, X. Xia, and D. Lo, "Assessing and advancing benchmarks for evaluating large language models in software engineering tasks," 2025.

[34] K. Wang, T. Li, X. Zhang, C. Wang, W. Sun, Y. Liu, and B. Shi, "Software development life cycle perspective: A survey of benchmarks for code large language models and agents," 2025.

[35] R. Koohestani, P. de Bekker, and M. Izadi, "Benchmarking ai models in software engineering: A review, search tool, and enhancement protocol," *arXiv preprint arXiv:2503.05860*, 2025.

[36] H.-N. Zhu, R. M. Furth, M. Pradel, and C. Rubio-González, "From bugs to benchmarks: A comprehensive survey of software defect datasets," *arXiv preprint arXiv:2504.17977*, 2025.

[37] J. Cao, Y.-K. Chan, Z. Ling, W. Wang, S. Li, M. Liu, R. Qiao, Y. Han, C. Wang, B. Yu, *et al.*, "How should we build a benchmark? revisiting 274 code-related benchmarks for llms," 2025.

[38] S. Chen, Y. Chen, Z. Li, Y. Jiang, Z. Wan, Y. He, D. Ran, T. Gu, H. Li, T. Xie, *et al.*, "Recent advances in large langauge model benchmarks against data contamination: From static to dynamic evaluation," *arXiv preprint arXiv:2502.17521*, 2025.

[39] X. Zhou, M. Weyssow, R. Widyasari, T. Zhang, J. He, Y. Lyu, J. Chang, B. Zhang, D. Huang, and D. Lo, "Lessleak-bench: A first investigation of data leakage in llms across 83 software engineering benchmarks," *arXiv preprint arXiv:2502.06215*, 2025.

[40] Z. Zheng, K. Ning, Y. Wang, J. Zhang, D. Zheng, M. Ye, and J. Chen, "A survey of large language models for code: Evolution, benchmarking, and future trends," *arXiv preprint arXiv:2311.10372*, 2023.

[41] L. Chen, Q. Guo, H. Jia, Z. Zeng, X. Wang, Y. Xu, J. Wu, Y. Wang, Q. Gao, J. Wang, *et al.*, "A survey on evaluating large language models in code generation tasks," *arXiv preprint arXiv:2408.16498*, 2024.

[42] Z. Zhang, C. Chen, B. Liu, C. Liao, Z. Gong, H. Yu, J. Li, and R. Wang, "Unifying the perspectives of nlp and software engineering: A survey on language models for code," *arXiv preprint arXiv:2311.07989*, 2023.

[43] H. Jin, L. Huang, H. Cai, J. Yan, B. Li, and H. Chen, "From llms to llm-based agents for software engineering: A survey of current, challenges and future," *arXiv preprint arXiv:2408.02479*, 2024.

[44] J. Wei, X. Wang, D. Schuurmans, *et al.*, "Chain-of-thought prompting elicits reasoning in large language models," in *Advances in Neural Information Processing Systems (NeurIPS)*, 2022.

[45] T. Kojima, S. S. Gu, M. Reid, Y. Matsuo, and Y. Iwasawa, "Large language models are zero-shot reasoners," in *Advances in Neural Information Processing Systems (NeurIPS)*, 2022.

[46] X. Wang, J. Wei, D. Schuurmans, *et al.*, "Self-consistency improves chain-of-thought reasoning in language models," in *International Conference on Learning Representations (ICLR)*, 2023.

[47] S. Yao, D. Yu, J. Zhao, I. Shafran, T. Griffiths, Y. Cao, and K. Narasimhan, "Tree of thoughts: Deliberate problem solving with large language models," in *Advances in Neural Information Processing Systems (NeurIPS)*, 2023.

[48] S. Yao, J. Zhao, D. Yu, N. Du, I. Shafran, K. Narasimhan, and Y. Cao, "React: Synergizing reasoning and acting in language models," in *International Conference on Learning Representations (ICLR)*, 2023.

[49] T. Schick, J. Dwivedi-Yu, R. Dessì, *et al.*, "Toolformer: Language models can teach themselves to use tools," in *Advances in Neural Information Processing Systems (NeurIPS)*, 2023.

[50] L. Gao, A. Madaan, S. Zhou, *et al.*, "PAL: Program-aided language models," in *International Conference on Machine Learning (ICML)*, 2023.

[51] W. Chen, X. Ma, X. Wang, and W. W. Cohen, "Program of thoughts prompting: Disentangling computation from reasoning for numerical reasoning tasks," in *Transactions on Machine Learning Research (TMLR)*, 2023.

[52] OpenAI, "Openai o4-mini system card," 2025. Technical report.

[53] D. Guo, D. Yang, H. Zhang, *et al.*, "Deepseek-r1: Incentivizing reasoning capability in LLMs via reinforcement learning," *arXiv preprint arXiv:2501.12948*, 2025.

[54] Google DeepMind, "Gemini 2.5: Pushing the frontier with advanced reasoning," 2025. Technical report.

[55] Anthropic, "Claude 4 sonnet system card," 2025. Technical report.

[56] C. Liu, S. D. Bhattacharya, Z. Yu, and M. Pradel, "CodeMind: A framework to evaluate and improve code reasoning of large language models," in *International Conference on Software Engineering (ICSE)*, 2024.

[57] Y. Zhang *et al.*, "On the limits of chain-of-thought reasoning for code," *arXiv preprint*, 2024.

[58] F. F. Xu *et al.*, "A multilingual benchmark for code reasoning," *arXiv preprint*, 2024.

[59] Y. Dong *et al.*, "Probing type inference reasoning in LLMs," *arXiv preprint*, 2024.

[60] C. Liu *et al.*, "ExeRScope: Linking execution behaviour with program properties for code reasoning," *arXiv preprint*, 2024.

[61] E. La Malfa *et al.*, "Algorithmic reasoning of language models," in *Proceedings*, 2024.

[62] T. Xue *et al.*, "Human assessment of reasoning quality in LLM code inference," *arXiv preprint*, 2024.

[63] C. Authors, "CoCoNUT: Structural code understanding does not fall out of a tree," in *Proceedings*, 2024.

[64] H. Lightman, V. Kosaraju, Y. Burda, *et al.*, "Let's verify step by step," in *International Conference on Learning Representations (ICLR)*, 2024.

[65] J. Uesato, N. Kushman, R. Kumar, *et al.*, "Solving math word problems with process- and outcome-based feedback," *arXiv preprint arXiv:2211.14275*, 2022.

[66] E. Zelikman, Y. Wu, J. Mu, and N. D. Goodman, "STaR: Self-taught reasoner – bootstrapping reasoning with reasoning," in *Advances in Neural Information Processing Systems (NeurIPS)*, 2022.

[67] L. C. Magister, J. Mallinson, J. Adamek, E. Malmi, and A. Severyn, "Teaching small language models to reason," in *Annual Meeting of the Association for Computational Linguistics (ACL)*, 2023.

[68] C.-Y. Hsieh, C.-L. Li, C.-K. Yeh, *et al.*, "Distilling step-by-step! outperforming larger language models with less training data and smaller model sizes," in *Findings of ACL*, 2023.

[69] L. Ouyang, J. Wu, X. Jiang, *et al.*, "Training language models to follow instructions with human feedback," in *Advances in Neural Information Processing Systems (NeurIPS)*, 2022.

[70] H. Lee, S. Phatale, H. Mansoor, *et al.*, "RLAIF: Scaling reinforcement learning from human feedback with AI feedback," *arXiv preprint arXiv:2309.00267*, 2023.

[71] H. Le, Y. Wang, A. D. Gotmare, S. Savarese, and S. C. H. Hoi, "CodeRL: Mastering code generation through pretrained models and deep reinforcement learning," in *Advances in Neural Information Processing Systems (NeurIPS)*, 2022.

[72] P. Shojaee, A. Jain, S. Tipirneni, and C. K. Reddy, "PPOCoder: Execution-based code generation using deep reinforcement learning," in *International Conference on Machine Learning (ICML)*, 2023.

[73] J. Liu, Y. Zhu, K. Xiao, *et al.*, "RLTF: Reinforcement learning from unit test feedback," *arXiv preprint arXiv:2307.04349*, 2023.

[74] D. Amodei, C. Olah, J. Steinhardt, P. Christiano, J. Schulman, and D. Mané, "Concrete problems in AI safety," *arXiv preprint arXiv:1606.06565*, 2016.

[75] S. Rajan, "Auditing reward hackability in code RL training environments," *arXiv preprint arXiv:2606.16062*, 2026.

[76] R. Mitsuhashi, P. Chen, I. Tseng, J. Cekinmez, and A. J. Wu, "Quantifying empirical compute-supervision tradeoffs in RLVR," *arXiv preprint arXiv:2605.25252*, 2026.

[77] J. Schulman, F. Wolski, P. Dhariwal, A. Radford, and O. Klimov, "Proximal policy optimization algorithms," *arXiv preprint arXiv:1707.06347*, 2017.

[78] R. Rafailov, A. Sharma, E. Mitchell, *et al.*, "Direct preference optimization: Your language model is secretly a reward model," in *Advances in Neural Information Processing Systems (NeurIPS)*, 2023.

[79] Z. Shao, P. Wang, Q. Zhu, *et al.*, "Deepseekmath: Pushing the limits of mathematical reasoning in open language models," *arXiv preprint arXiv:2402.03300*, 2024.

[80] C. Le Goues, T. Nguyen, S. Forrest, and W. Weimer, "GenProg: A generic method for automatic software repair," *IEEE Transactions on Software Engineering*, vol. 38, no. 1, pp. 54–72, 2012.

[81] S. Mechtaev, J. Yi, and A. Roychoudhury, "Angelix: Scalable multiline program patch synthesis via symbolic analysis," in *ICSE*, 2016.

[82] M. Balog, A. L. Gaunt, M. Brockschmidt, S. Nowozin, and D. Tarlow, "DeepCoder: Learning to write programs," in *ICLR*, 2017.

[83] OpenAI, "Introducing SWE-bench verified," *OpenAI Technical Report*, 2024. `https://openai.com/index/introducing-swe-bench-verified/`.

[84] R. Aleithan, H. Wang, M. M. Mohajer, S. Wang, A. E. Hassan, *et al.*, "SWE-Bench+: Enhanced coding benchmark for LLMs," *arXiv preprint arXiv:2410.06992*, 2024.

[85] J. Pan, X. Wang, G. Neubig, J. Yang, O. Press, D. Klein, and A. Suhr, "SWE-Gym: An open environment for training software engineering agents and verifiers," *arXiv preprint arXiv:2412.21139*, 2024.

[86] N. Jain, J. Singh, M. Shetty, L. Zheng, K. Sen, and I. Stoica, "R2E-Gym: Procedural environments and hybrid verifiers for scaling open-weights SWE agents," *arXiv preprint arXiv:2504.07164*, 2025.

[87] I. Badertdinov, A. Golubev, M. Nekrashevich, A. Shevtsov, S. Karasik, A. Andriushchenko, M. Trofimova, D. Litvintseva, and B. Yangel, "SWE-rebench: An automated pipeline for task collection and decontaminated evaluation of software engineering agents," *arXiv preprint arXiv:2505.20411*, 2025.

[88] Y. Wei, S. Sukhbaatar, Y. Hadji, G. Neubig, G. Synnaeve, *et al.*, "SWE-RL: Advancing LLM reasoning via reinforcement learning on open software evolution," *arXiv preprint arXiv:2502.18449*, 2025.

[89] H. Song, L. Huang, S. Sun, J. Jiang, R. Le, D. Cheng, G. Chen, Y. Hu, Z. Chen, Y. Jia, W. X. Zhao, Y. Song, T. Zhang, and J.-R. Wen, "SWE-Master: Unleashing the potential of software engineering agents via post-training," *arXiv preprint arXiv:2602.03411*, 2026.

[90] Agentica and Together AI, "DeepSWE: Training a fully open-sourced state-of-the-art coding agent by scaling RL," *Technical report*, 2025.

[91] J. Da, C. Wang, X. Deng, Y. Ma, N. Barhate, and S. Hendryx, "Agent-RLVR: Training software engineering agents via guidance and environment rewards," *arXiv preprint arXiv:2506.11425*, 2025.

[92] N. Jiang, X. Li, S. Wang, Q. Zhou, S. B. Hossain, B. Ray, V. Kumar, X. Ma, and A. Deoras, "LeDex: Training LLMs to better self-debug and explain code," in *NeurIPS*, 2024.

[93] K. Shum, B. Hui, J. Chen, L. Zhang, J. Yang, Y. Huang, J. Lin, and J. He, "SWE-RM: Execution-free feedback for software engineering agents," *arXiv preprint arXiv:2512.21919*, 2025.

[94] Y. Wang, L. Yang, Y. Tian, K. Shen, and M. Wang, "Co-evolving LLM coder and unit tester via reinforcement learning," *NeurIPS*, 2025.

[95] D. Lee, C. Hwang, and K. Lee, "Learning to generate unit test via adversarial reinforcement learning," *arXiv preprint arXiv:2508.21107*, 2025.

[96] S. Zhu, J. Wang, J. Ai, and X. Li, "MIST-RL: Mutation-based incremental suite testing via reinforcement learning," *arXiv preprint arXiv:2603.01409*, 2026.

[97] C. Ruan, D. Jiang, H. Zeng, P. Nie, and W. Chen, "EvolveCoder: Evolving test cases via adversarial verification for code reinforcement learning," *arXiv preprint arXiv:2603.12698*, 2026.

[98] A. Wang, Y. Yan, N. Zhou, Z. Lu, W. Lu, J. Xiao, Y. Zhuang, and Y. Shen, "Code-A1: Adversarial evolving of code LLM and test LLM via reinforcement learning," *arXiv preprint arXiv:2603.15611*, 2026.

[99] J. Cai, J. Zhu, R. Sun, K. Zhao, D. Xue, M. Feng, W. Zhou, and H. Li, "CodeContests-O: Powering LLMs via feedback-driven iterative test case generation," *arXiv preprint arXiv:2601.13682*, 2026.

[100] S. Yoo and M. Harman, "Regression testing minimization, selection and prioritization: A survey," *Software Testing, Verification and Reliability*, vol. 22, no. 2, pp. 67–120, 2012.

[101] G. Rothermel, R. H. Untch, C. Chu, and M. J. Harrold, "Prioritizing test cases for regression testing," *IEEE Transactions on Software Engineering*, vol. 27, no. 10, pp. 929–948, 2001.

[102] Y. Jia and M. Harman, "An analysis and survey of the development of mutation testing," *IEEE Transactions on Software Engineering*, vol. 37, no. 5, pp. 649–678, 2011.

[103] Y. Bengio, J. Louradour, R. Collobert, and J. Weston, "Curriculum learning," in *ICML*, 2009.

[104] Z. Zeng, H. Ivison, Y. Wang, L. Yuan, S. C. Li, Z. Ye, S. Li, J. He, R. Zhou, T. Chen, C. Zhao, Y. Tsvetkov, S. S. Du, N. Jaques, H. Peng, P. W. Koh, and H. Hajishirzi, "RLVE: Scaling up reinforcement learning for language models with adaptive verifiable environments," *ICML*, 2026.

[105] Q. Yu *et al.*, "DAPO: An open-source LLM reinforcement learning system at scale," *NeurIPS*, 2025.

[106] H. Zheng, Y. Zhou, B. R. Bartoldson, B. Kailkhura, F. Lai, J. Zhao, and B. Chen, "Act only when it pays: Efficient reinforcement learning for LLM reasoning via selective rollouts," *arXiv preprint arXiv:2506.02177*, 2025.

[107] A. Fan, B. Gokkaya, M. Harman, M. Lyubarskiy, S. Sengupta, S. Yoo, and J. M. Zhang, "Large language models for software engineering: Survey and open problems," in *ICSE-FoSE'23*, pp. 31–53, 2023.

[108] S. Keele *et al.*, "Guidelines for performing systematic literature reviews in software engineering," 2007.

[109] K. Petersen, S. Vakkalanka, and L. Kuzniarz, "Guidelines for conducting systematic mapping studies in software engineering: An update," *Information and Software Technology*, 2015.

[110] Y. Wu, Z. Li, J. M. Zhang, and Y. Liu, "Condefects: A complementary dataset to address the data leakage concern for llm-based fault localization and program repair," in *FSE'24*, FSE 2024, p. 642–646, 2024.

[111] J. Y. Lee, S. Kang, J. Yoon, and S. Yoo, "The github recent bugs dataset for evaluating llm-based debugging applications," in *ICST'24*, pp. 442–444, 2024.

[112] S. Arasteh, G. Nikitopoulos, W.-C. Wu, N. Weideman, A. Portnoy, M. Raghothaman, and C. Hauser, "Binpool: A dataset of vulnerabilities for binary security analysis," 2025.

[113] C. Lu, T. Li, T. Dehaene, and B. Lagaisse, "Icvul: A well-labeled c/c++ vulnerability dataset with comprehensive metadata and vccs," in *MSR'25*, pp. 154–158, IEEE, 2025.

[114] B. Ruan, J. Liu, W. Zhao, and Z. Liang, "Vulzoo: A comprehensive vulnerability intelligence dataset," 2024.

[115] J. Liu, C. S. Xia, Y. Wang, and L. Zhang, "Is your code generated by chatgpt really correct? rigorous evaluation of large language models for code generation," *Advances in Neural Information Processing Systems*, vol. 36, pp. 21558–21572, 2023.

[116] C. Ebert, J. Cain, G. Antoniol, S. Counsell, and P. Laplante, "Cyclomatic complexity," *IEEE software*, vol. 33, no. 6, pp. 27–29, 2016.

[117] T. Hariprasad, G. Vidhyagaran, K. Seenu, and C. Thirumalai, "Software complexity analysis using halstead metrics," in *2017 international conference on trends in electronics and informatics (ICEI)*, pp. 1109–1113, IEEE, 2017.

[118] T. Vagoun, "Input domain partitioning in software testing," in *Proceedings of HICSS-29: 29th Hawaii International Conference on System Sciences*, vol. 2, pp. 261–268 vol.2, 1996.

[119] A. Y. Ng, D. Harada, and S. Russell, "Policy invariance under reward transformations: Theory and application to reward shaping," in *ICML*, 1999.

[120] S. Devlin and D. Kudenko, "Dynamic potential-based reward shaping," in *AAMAS*, 2012.

[121] M. Chen, J. Tworek, H. Jun, Q. Yuan, H. P. D. O. Pinto, J. Kaplan, H. Edwards, Y. Burda, N. Joseph, G. Brockman, *et al.*, "Evaluating large language models trained on code," *arXiv preprint arXiv:2107.03374*, 2021.

[122] J. Austin, A. Odena, M. Nye, M. Bosma, H. Michalewski, D. Dohan, E. Jiang, C. Cai, M. Terry, Q. Le, *et al.*, "Program synthesis with large language models," *arXiv preprint arXiv:2108.07732*, 2021.

[123] Z. Pan, R. Cao, Y. Cao, Y. Ma, B. Li, F. Huang, H. Liu, and Y. Li, "Codev-bench: How do llms understand developer-centric code completion?," *arXiv preprint arXiv:2410.01353*, 2024.

[124] H. Yu, B. Shen, D. Ran, J. Zhang, Q. Zhang, Y. Ma, G. Liang, Y. Li, Q. Wang, and T. Xie, "Codereval: A benchmark of pragmatic code generation with generative pre-trained models," in *ICSE'24*, pp. 1–12, 2024.

[125] N. Shah, Z. Genc, and D. Araci, "Stackeval: Benchmarking llms in coding assistance," *Advances in Neural Information Processing Systems*, vol. 37, pp. 36976–36994, 2024.

[126] S. Lu, D. Guo, S. Ren, J. Huang, A. Svyatkovskiy, A. Blanco, C. Clement, D. Drain, D. Jiang, D. Tang, *et al.*, "Codexglue: A machine learning benchmark dataset for code understanding and generation," *arXiv preprint arXiv:2102.04664*, 2021.

[127] M. Li, Y. Zhao, B. Yu, F. Song, H. Li, H. Yu, Z. Li, F. Huang, and Y. Li, "API-bank: A comprehensive benchmark for tool-augmented LLMs," in *EMNLP 2023*, pp. 3102–3116, Dec. 2023.

[128] Y. Song, W. Xiong, D. Zhu, W. Wu, H. Qian, M. Song, H. Huang, C. Li, K. Wang, R. Yao, *et al.*, "Restgpt: Connecting large language models with real-world restful apis," *arXiv preprint arXiv:2306.06624*, 2023.

[129] L. Zhong and Z. Wang, "Can llm replace stack overflow? a study on robustness and reliability of large language model code generation," vol. 38, pp. 21841–21849, Mar. 2024.

[130] B. Tan, J. Xu, Z. Zhu, and P. He, "Al-bench: A benchmark for automatic logging," 2025.

[131] M. K. Habib, D. Graziotin, and S. Wagner, "Reqbrain: Task-specific instruction tuning of llms for ai-assisted requirements generation," 2025.

[132] G. Voria, F. Casillo, C. Gravino, G. Catolino, and F. Palomba, " RECOVER: Toward Requirements Generation From Stakeholders' Conversations ," *IEEE Transactions on Software Engineering*, vol. 51, pp. 1912–1933, June 2025.

[133] S. Li, J. Jiang, T. Zhao, and J. Shen, "Osvbench: Benchmarking llms on specification generation tasks for operating system verification," 2025.

[134] N. Raihan, A. Anastasopoulos, and M. Zampieri, "mHumanEval - a multilingual benchmark to evaluate large language models for code generation," in *NAACL 2025* (L. Chiruzzo, A. Ritter, and L. Wang, eds.), Apr. 2025.

[135] J. Feng, J. Liu, C. Gao, C. Y. Chong, C. Wang, S. Gao, and X. Xia, "Complexcodeeval: A benchmark for evaluating large code models on more complex code," in *ASE '24*, ASE '24, p. 1895–1906, 2024.

[136] A. Matton, T. Sherborne, D. Aumiller, E. Tommasone, M. Alizadeh, J. He, R. Ma, M. Voisin, E. Gilsenan-McMahon, and M. Gallé, "On leakage of code generation evaluation datasets," in *EMNLP 2024*, pp. 13215–13223, Nov. 2024.

[137] N. L. Hai, D. M. Nguyen, and N. D. Q. Bui, "On the impacts of contexts on repository-level code generation," in *NAACL 2025* (L. Chiruzzo, A. Ritter, and L. Wang, eds.), pp. 1496–1524, Apr. 2025.

[138] S. Liang, Y. Hu, N. Jiang, and L. Tan, "Can language models replace programmers for coding? repocod says 'not yet'," 2025.

[139] N. Jain, S. Vaidyanath, A. Iyer, N. Natarajan, S. Parthasarathy, S. Rajamani, and R. Sharma, "Jigsaw: large language models meet program synthesis," in *ICSE '22*, ICSE '22, p. 1219–1231, 2022.

[140] H. Han, seung-won hwang, R. Samdani, and Y. He, "Convcodeworld: Benchmarking conversational code generation in reproducible feedback environments," in *ICLR'25*, 2025.

[141] D. Zan, B. Chen, D. Yang, Z. Lin, M. Kim, B. Guan, Y. Wang, W. Chen, and J.-G. Lou, "Cert: Continual pre-training on sketches for library-oriented code generation," in *IJCAI-22*, pp. 2369–2375, 7 2022.

[142] Z. Peng, X. Yin, R. Qian, P. Lin, Y. Liu, C. Ying, and Y. Luo, "Soleval: Benchmarking large language models for repository-level solidity code generation," 2025.

[143] J. J. Wu and F. H. Fard, "Humanevalcomm: Benchmarking the communication competence of code generation for llms and llm agent," *ACM Trans. Softw. Eng. Methodol.*, Jan. 2025.

[144] P. Wang, L. Zhang, F. Liu, L. Shi, M. Li, B. Shen, and A. Fu, "Codeif-bench: Evaluating instruction-following capabilities of large language models in interactive code generation," 2025.

[145] D. Liao, S. Pan, X. Sun, X. Ren, Q. Huang, Z. Xing, H. Jin, and Q. Li, "$\mathbf{A^3}$a3-codgen: A repository-level code generation framework for code reuse with local-aware, global-aware, and third-party-library-aware," *IEEE Trans. Softw. Eng.*, vol. 50, p. 3369–3384, Dec. 2024.

[146] J. Chen, Q. Zhong, Y. Wang, K. Ning, Y. Liu, Z. Xu, Z. Zhao, T. Chen, and Z. Zheng, "Rmcbench: Benchmarking large language models' resistance to malicious code," in *ASE'24*, ASE '24, p. 995–1006, 2024.

[147] F. Cassano, J. Gouwar, D. Nguyen, S. Nguyen, L. Phipps-Costin, D. Pinckney, M.-H. Yee, Y. Zi, C. J. Anderson, M. Q. Feldman, A. Guha, M. Greenberg, and A. Jangda, "Multipl-e: A scalable and polyglot approach to benchmarking neural code generation," *IEEE Transactions on Software Engineering*, vol. 49, no. 7, pp. 3675–3691, 2023.

[148] B. Athiwaratkun, S. K. Gouda, Z. Wang, X. Li, Y. Tian, M. Tan, W. U. Ahmad, S. Wang, Q. Sun, M. Shang, *et al.*, "Multi-lingual evaluation of code generation models," *arXiv preprint arXiv:2210.14868*, 2022.

[149] Q. Zheng, X. Xia, X. Zou, Y. Dong, S. Wang, Y. Xue, L. Shen, Z. Wang, A. Wang, Y. Li, T. Su, Z. Yang, and J. Tang, "Codegeex: A pre-trained model for code generation with multilingual benchmarking on humaneval-x," in *SIGKDD'23*, KDD '23, p. 5673–5684, 2023.

[150] J. Zheng, B. Cao, Z. Ma, R. Pan, H. Lin, Y. Lu, X. Han, and L. Sun, "Beyond correctness: Benchmarking multi-dimensional code generation for large language models," *arXiv preprint arXiv:2407.11470*, 2024.

[151] Y. Li, D. Choi, J. Chung, N. Kushman, J. Schrittwieser, R. Leblond, T. Eccles, J. Keeling, F. Gimeno, A. Dal Lago, *et al.*, "Competition-level code generation with alphacode," *Science*, vol. 378, no. 6624, pp. 1092–1097, 2022.

[152] C. White, S. Dooley, M. Roberts, A. Pal, B. Feuer, S. Jain, R. Shwartz-Ziv, N. Jain, K. Saifullah, S. Naidu, *et al.*, "Livebench: A challenging, contamination-free llm benchmark," *arXiv preprint arXiv:2406.19314*, 2024.

[153] W. Hu, J. Duan, C. Wei, L. Zhang, Y. Zhang, and K. Xu, "Dynacode: A dynamic complexity-aware code benchmark for evaluating large language models in code generation," 2025.

[154] C. Wu, Z. Liang, Y. Ge, Q. Guo, Z. Lu, J. Wang, Y. Shan, and P. Luo, "Plot2Code: A comprehensive benchmark for evaluating multi-modal large language models in code generation from scientific plots," in *NAACL 2025* (L. Chiruzzo, A. Ritter, and L. Wang, eds.), pp. 3006–3028, Apr. 2025.

[155] J. Dai, J. Lu, Y. Feng, D. Huang, G. Zeng, R. Ruan, M. Cheng, H. Tan, and Z. Guo, "Mhpp: Exploring the capabilities and limitations of language models beyond basic code generation," *arXiv preprint arXiv:2405.11430*, 2024.

[156] H. Wang, X. Zhou, Z. Xu, K. Cheng, Y. Zuo, K. Tian, J. Song, J. Lu, W. Hu, and X. Liu, "Code-vision: Evaluating multimodal llms logic understanding and code generation capabilities," 2025.

[157] Q. Peng, Y. Chai, and X. Li, "HumanEval-XL: A multilingual code generation benchmark for cross-lingual natural language generalization," in *LREC-COLING 2024*, pp. 8383–8394, May 2024.

[158] L. Chai, S. Liu, J. Yang, Y. Yin, K. Jin, J. Liu, T. Sun, G. Zhang, C. Ren, H. Guo, *et al.*, "Mceval: Massively multilingual code evaluation," *arXiv preprint arXiv:2406.07436*, 2024.

[159] R. Li, J. Fu, B.-W. Zhang, T. Huang, Z. Sun, C. Lyu, G. Liu, Z. Jin, and G. Li, "Taco: Topics in algorithmic code generation dataset," *arXiv preprint arXiv:2312.14852*, 2023.

[160] K. Yan, H. Guo, X. Shi, J. Xu, Y. Gu, and Z. Li, "Codeif: Benchmarking the instruction-following capabilities of large language models for code generation," 2025.

[161] J. Li, G. Li, Y. Zhao, Y. Li, H. Liu, H. Zhu, L. Wang, K. Liu, Z. Fang, L. Wang, J. Ding, X. Zhang, Y. Zhu, Y. Dong, Z. Jin, B. Li, F. Huang, Y. Li, B. Gu, and M. Yang, "DevEval: A manually-annotated code generation benchmark aligned with real-world code repositories," in *ACL 2024*, pp. 3603–3614, Aug. 2024.

[162] S. Chandel, C. B. Clement, G. Serrato, and N. Sundaresan, "Training and evaluating a jupyter notebook data science assistant," *arXiv preprint arXiv:2201.12901*, 2022.

[163] T. Y. Zhuo, M. C. Vu, J. Chim, H. Hu, W. Yu, R. Widyasari, I. N. B. Yusuf, H. Zhan, J. He, I. Paul, *et al.*, "Bigcodebench: Benchmarking code generation with diverse function calls and complex instructions," *arXiv preprint arXiv:2406.15877*, 2024.

[164] N. Shinn, F. Cassano, A. Gopinath, K. Narasimhan, and S. Yao, "Reflexion: Language agents with verbal reinforcement learning," *Advances in Neural Information Processing Systems*, vol. 36, pp. 8634–8652, 2023.

[165] D. Nichols, J. H. Davis, Z. Xie, A. Rajaram, and A. Bhatele, "Can large language models write parallel code?," in *HPDC'24*, pp. 281–294, 2024.

[166] Y. Xia, Y. Chen, T. Shi, J. Wang, and J. Yang, "Aicodereval: Improving ai domain code generation of large language models," *arXiv preprint arXiv:2406.04712*, 2024.

[167] P. Yin, W.-D. Li, K. Xiao, A. Rao, Y. Wen, K. Shi, J. Howland, P. Bailey, M. Catasta, H. Michalewski, O. Polozov, and C. Sutton, "Natural language to code generation in interactive data science notebooks," in *Proceedings of the 61st Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)* (A. Rogers, J. Boyd-Graber, and N. Okazaki, eds.), (Toronto, Canada), pp. 126–173, Association for Computational Linguistics, July 2023.

[168] J. Liu, C. S. Xia, Y. Wang, and L. Zhang, "Is your code generated by chatgpt really correct? rigorous evaluation of large language models for code generation," *Advances in Neural Information Processing Systems*, vol. 36, pp. 21558–21572, 2023.

[169] D. Guo, Q. Zhu, D. Yang, Z. Xie, K. Dong, W. Zhang, G. Chen, X. Bi, Y. Wu, Y. Li, *et al.*, "Deepseek-coder: When the large language model meets programming–the rise of code intelligence," *arXiv preprint arXiv:2401.14196*, 2024.

[170] K. Xu, Y. Mao, X. Guan, and Z. Feng, "Web-bench: A llm code benchmark based on web standards and frameworks," 2025.

[171] D. Zan, B. Chen, Z. Lin, B. Guan, W. Yongji, and J.-G. Lou, "When language model meets private library," in *EMNLP 2022* (Y. Goldberg, Z. Kozareva, and Y. Zhang, eds.), pp. 277–288, Dec. 2022.

[172] C. S. Xia, Y. Deng, and L. Zhang, "Top leaderboard ranking= top coding proficiency, always? evoeval: Evolving coding benchmarks via llm," *arXiv preprint arXiv:2403.19114*, 2024.

[173] W. Li, X. Zhang, Z. Guo, S. Mao, W. Luo, G. Peng, Y. Huang, H. Wang, and S. Li, "Fea-bench: A benchmark for evaluating repository-level code generation for feature implementation," 2025.

[174] J. S. Chan, N. Chowdhury, O. Jaffe, J. Aung, D. Sherburn, E. Mays, G. Starace, K. Liu, L. Maksin, T. Patwardhan, *et al.*, "Mle-bench: Evaluating machine learning agents on machine learning engineering," *arXiv preprint arXiv:2410.07095*, 2024.

[175] D. G. Paul, H. Zhu, and I. Bayley, "Sceneval: A benchmark for scenario-based evaluation of code generation," in *AITest'24*, pp. 55–63, 2024.

[176] W. Yan, H. Liu, Y. Wang, Y. Li, Q. Chen, W. Wang, T. Lin, W. Zhao, L. Zhu, H. Sundaram, and S. Deng, "CodeScope: An execution-based multilingual multitask multidimensional benchmark for evaluating LLMs on code understanding and generation," in *ACL'24*, pp. 5511–5558, Aug. 2024.

[177] J. Cao, Z. Chen, J. Wu, S.-C. Cheung, and C. Xu, "Javabench: A benchmark of object-oriented code generation for evaluating large language models," in *ASE'24*, ASE '24, p. 870–882, 2024.

[178] Z. Wang, S. Zhou, D. Fried, and G. Neubig, "Execution-based evaluation for open-domain code generation," in *EMNLP 2023*, pp. 1271–1290, Dec. 2023.

[179] E. Bogomolov, A. Eliseeva, T. Galimzyanov, E. Glukhov, A. Shapkin, M. Tigina, Y. Golubev, A. Kovrigin, A. van Deursen, M. Izadi, *et al.*, "Long code arena: a set of benchmarks for long-context code models," *arXiv preprint arXiv:2406.11612*, 2024.

[180] S. Kuhar, W. U. Ahmad, Z. Wang, N. Jain, H. Qian, B. Ray, M. K. Ramanathan, X. Ma, and A. Deoras, "LibEvolutionEval: A benchmark and study for version-specific code generation," in *NAACL 2025* (L. Chiruzzo, A. Ritter, and L. Wang, eds.), Apr. 2025.

[181] Z. Wang, R. Ling, C. Wang, Y. Yu, S. Wang, Z. Li, F. Xiong, and W. Zhang, "Maintaincoder: Maintainable code generation under dynamic requirements," 2025.

[182] Z. Zeng, Y. Wang, R. Xie, W. Ye, and S. Zhang, "Coderujb: An executable and unified java benchmark for practical programming scenarios," in *ISSTA 2024*, ISSTA 2024, p. 124–136, 2024.

[183] M. Du, L. A. Tuan, B. Ji, Q. Liu, and S.-K. Ng, "Mercury: A code efficiency benchmark for code large language models," in *Advances in Neural Information Processing Systems* (A. Globerson, L. Mackey, D. Belgrave, A. Fan, U. Paquet, J. Tomczak, and C. Zhang, eds.), vol. 37, pp. 16601–16622, Curran Associates, Inc., 2024.

[184] D. Hendrycks, S. Basart, S. Kadavath, M. Mazeika, A. Arora, E. Guo, C. Burns, S. Puranik, H. He, D. Song, and J. Steinhardt, "Measuring coding challenge competence with apps," in *Proceedings of the Neural Information Processing Systems Track on Datasets and Benchmarks* (J. Vanschoren and S. Yeung, eds.), vol. 1, 2021.

[185] C. Niu, C. Li, V. Ng, and B. Luo, "Crosscodebench: Benchmarking cross-task generalization of source code models," in *ICSE'23*, pp. 537–549, 2023.

[186] Y. Huang, J. Luo, Y. Yu, Y. Zhang, F. Lei, Y. Wei, S. He, L. Huang, X. Liu, J. Zhao, and K. Liu, "DA-code: Agent data science code generation benchmark for large language models," in *EMNLP'24* (Y. Al-Onaizan, M. Bansal, and Y.-N. Chen, eds.), pp. 13487–13521, Nov. 2024.

[187] Q. Zhu, J. Cao, Y. Lu, H. Lin, X. Han, L. Sun, and S.-C. Cheung, "Domaineval: An auto-constructed benchmark for multi-domain code generation," in *AAAI'25*, vol. 39, pp. 26148–26156, 2025.

[188] A. Daghighfarsoodeh, C.-Y. Wang, H. Taherkhani, M. Sepidband, M. Abdollahi, H. Hemmati, and H. V. Pham, "Deep-bench: Deep learning benchmark dataset for code generation," *arXiv preprint arXiv:2502.18726*, 2025.

[189] J. Li, G. Li, X. Zhang, Y. Dong, and Z. Jin, "Evocodebench: An evolving code generation benchmark aligned with real-world code repositories," *arXiv preprint arXiv:2404.00599*, 2024.

[190] Y. Lai, C. Li, Y. Wang, T. Zhang, R. Zhong, L. Zettlemoyer, W.-t. Yih, D. Fried, S. Wang, and T. Yu, "Ds-1000: a natural and reliable benchmark for data science code generation," in *ICML'23*, ICML'23, JMLR.org, 2023.

[191] J. Liu, S. Xie, J. Wang, Y. Wei, Y. Ding, and L. Zhang, "Evaluating language models for efficient code generation," *arXiv preprint arXiv:2408.06450*, 2024.

[192] N. Jain, K. Han, A. Gu, W.-D. Li, F. Yan, T. Zhang, S. Wang, A. Solar-Lezama, K. Sen, and I. Stoica, "Livecodebench: Holistic and contamination free evaluation of large language models for code," *arXiv preprint arXiv:2403.07974*, 2024.

[193] K. Liu, Y. Pan, Y. Xiang, D. He, J. Li, Y. Du, and T. Gao, "Projecteval: A benchmark for programming agents automated evaluation on project-level code generation," 2025.

[194] J. Li, B. Hui, G. Qu, J. Yang, B. Li, B. Li, B. Wang, B. Qin, R. Geng, N. Huo, X. Zhou, C. Ma, G. Li, K. C. Chang, F. Huang, R. Cheng, and Y. Li, "Can llm already serve as a database interface? a big bench for large-scale database grounded text-to-sqls," in *NIPS '23*, NIPS '23, (Red Hook, NY, USA), Curran Associates Inc., 2023.

[195] M. A. M. Khan, M. S. Bari, X. L. Do, W. Wang, M. R. Parvez, and S. Joty, "XCodeEval: An execution-based large scale multilingual multitask benchmark for code understanding, generation, translation and retrieval," in *ACL'24* (L.-W. Ku, A. Martins, and V. Srikumar, eds.), pp. 6766–6805, Aug. 2024.

[196] M. H. Nguyen, T. P. Chau, P. X. Nguyen, and N. D. Bui, "Agilecoder: Dynamic collaborative agents for software development based on agile methodology," *arXiv preprint arXiv:2406.11912*, 2024.

[197] D. Huang, J. M. Zhang, Y. Qing, and H. Cui, "Effibench: Benchmarking the efficiency of automatically generated code. corr abs/2402.02037 (2024)," 2024.

[198] X. Du, M. Liu, K. Wang, H. Wang, J. Liu, Y. Chen, J. Feng, C. Sha, X. Peng, and Y. Lou, "Classeval: A manually-crafted benchmark for evaluating llms on class-level code generation," *arXiv preprint arXiv:2308.01861*, 2023.

[199] Y. Han and C. Lyu, "Multi-stage guided code generation for large language models," *Engineering Applications of Artificial Intelligence*, vol. 139, p. 109491, 2025.

[200] E. Nijkamp, B. Pang, H. Hayashi, L. Tu, H. Wang, Y. Zhou, S. Savarese, and C. Xiong, "Codegen: An open large language model for code with multi-turn program synthesis," *arXiv preprint arXiv:2203.13474*, 2022.

[201] A. Yadav, H. Beniwal, and M. Singh, "PythonSaga: Redefining the benchmark to evaluate code generating LLMs," in *EMNLP 2024* (Y. Al-Onaizan, M. Bansal, and Y.-N. Chen, eds.), pp. 17113–17126, Nov. 2024.

[202] Y. Peng, J. Wan, Y. Li, and X. Ren, "Coffe: A code efficiency benchmark for code generation," 2025.

[203] S. Liu, H. Zhu, J. Liu, S. Xin, A. Li, R. Long, L. Chen, J. Yang, J. Xia, Z. Peng, *et al.*, "Fullstack bench: Evaluating llms as full stack coder," *arXiv preprint arXiv:2412.00535*, 2024.

[204] W. U. Ahmad, M. G. R. Tushar, S. Chakraborty, and K.-W. Chang, "AVATAR: A parallel corpus for Java-python program translation," in *ACL 2023* (A. Rogers, J. Boyd-Graber, and N. Okazaki, eds.), pp. 2268–2281, July 2023.

[205] W. Yan, Y. Tian, Y. Li, Q. Chen, and W. Wang, "Codetransocean: A comprehensive multilingual benchmark for code translation," *arXiv preprint arXiv:2310.04951*, 2023.

[206] Y. Wang, Y. Wang, S. Wang, D. Guo, J. Chen, J. Grundy, X. Liu, Y. Ma, M. Mao, H. Zhang, *et al.*, "Repotransbench: A real-world benchmark for repository-level code translation," *arXiv preprint arXiv:2412.17744*, 2024.

[207] G. Ou, M. Liu, Y. Chen, X. Peng, and Z. Zheng, "Repository-level code translation benchmark targeting rust," *arXiv preprint arXiv:2411.13990*, 2024.

[208] M. Jiao, T. Yu, X. Li, G. Qiu, X. Gu, and B. Shen, "On the evaluation of neural code translation: Taxonomy and benchmark," in *ASE'23*, pp. 1529–1541, IEEE, 2023.

[209] Y. Ding, Z. Wang, W. Ahmad, H. Ding, M. Tan, N. Jain, M. K. Ramanathan, R. Nallapati, P. Bhatia, D. Roth, *et al.*, "Crosscodeeval: A diverse and multilingual benchmark for cross-file code completion," *Advances in Neural Information Processing Systems*, vol. 36, pp. 46701–46723, 2023.

[210] J. Yang, J. Zhang, J. Yang, K. Jin, L. Zhang, Q. Peng, K. Deng, Y. Miao, T. Liu, Z. Cui, *et al.*, "Execrepobench: Multi-level executable code completion evaluation," *arXiv preprint arXiv:2412.11990*, 2024.

[211] T. Liu, C. Xu, and J. McAuley, "Repobench: Benchmarking repository-level code auto-completion systems," 2024.

[212] B. Li, W. Wu, Z. Tang, L. Shi, J. Yang, J. Li, S. Yao, C. Qian, B. Hui, Q. Zhang, Z. Yu, H. Du, P. Yang, D. Lin, C. Peng, and K. Chen, "Prompting large language models to tackle the full software development lifecycle: A case study," in *Proceedings of the 31st International Conference on Computational Linguistics* (O. Rambow, L. Wanner, M. Apidianaki, H. Al-Khalifa, B. D. Eugenio, and S. Schockaert, eds.), pp. 7511–7531, Jan. 2025.

[213] S. Deng, W. Xu, H. Sun, W. Liu, T. Tan, L. Liujianfeng, A. Li, J. Luan, B. Wang, R. Yan, and S. Shang, "Mobile-bench: An evaluation benchmark for LLM-based mobile agents," in *ACL'24*, pp. 8813–8831, Aug. 2024.

[214] B. LaBash, A. Rosedale, A. Reents, L. Negritto, and C. Wiel, "Res-q: Evaluating code-editing large language model systems at the repository scale," *arXiv preprint arXiv:2406.16801*, 2024.

[215] S. Fakhoury, S. Chakraborty, M. Musuvathi, and S. K. Lahiri, "Nl2fix: Generating functionally correct code edits from bug descriptions," in *ICSE-Companion'24*, ICSE-Companion '24, p. 410–411, 2024.

[216] M. Singhal, T. Aggarwal, A. Awasthi, N. Natarajan, and A. Kanade, "Nofuneval: Funny how code lms falter on requirements beyond functional correctness," *arXiv preprint arXiv:2401.15963*, 2024.

[217] Y. Zhao, Z. Luo, Y. Tian, H. Lin, W. Yan, A. Li, and J. Ma, "Codejudge-eval: Can large language models be good judges in code understanding?," *arXiv preprint arXiv:2408.10718*, 2024.

[218] D. M. Nguyen, T. C. Phan, N. Le Hai, T.-T. Doan, N. V. Nguyen, Q. Pham, and N. D. Bui, "Codemmlu: A multi-task benchmark for assessing code understanding & reasoning capabilities of codellms," in *ICLR'24*, 2024.

[219] A. Gu, B. Rozière, H. Leather, A. Solar-Lezama, G. Synnaeve, and S. I. Wang, "Cruxeval: a benchmark for code reasoning, understanding and execution," in *Proceedings of the 41st International Conference on Machine Learning*, ICML'24, JMLR.org, 2024.

[220] L. Ma, S. Liu, L. Bu, S. Li, Y. Wang, and Y. Liu, "Speceval: Evaluating code comprehension in large language models via program specifications," *arXiv preprint arXiv:2409.12866*, 2024.

[221] R. Hu, C. Peng, J. Ren, B. Jiang, X. Meng, Q. Wu, P. Gao, X. Wang, and C. Gao, "Codereporqa: A large-scale benchmark for software engineering question answering," *arXiv preprint arXiv:2412.14764*, 2024.

[222] C. Loughridge, Q. Sun, S. Ahrenbach, F. Cassano, C. Sun, Y. Sheng, A. Mudide, M. R. H. Misu, N. Amin, and M. Tegmark, "Dafnybench: A benchmark for formal software verification," *arXiv preprint arXiv:2406.08467*, 2024.

[223] J. Chen, Z. Pan, X. Hu, Z. Li, G. Li, and X. Xia, "Reasoning runtime behavior of a program with llm: How far are we?," *arXiv preprint arXiv:2403.16437*, 2024.

[224] S. Chen, P. Pusarla, and B. Ray, "Dycodeeval: Dynamic benchmarking of reasoning capabilities in code large language models under data contamination," in *Forty-second International Conference on Machine Learning*, 2025.

[225] V. Bhargava, R. Ghosh, and D. Dutta, "Cpp-ut-bench: Can llms write complex unit tests in c++?," *arXiv preprint arXiv:2412.02735*, 2024.

[226] N. Mündler, M. N. Müller, J. He, and M. Vechev, "Swt-bench: Testing and validating real-world bug-fixes with code agents," in *Advances in Neural Information Processing Systems* (A. Globerson, L. Mackey, D. Belgrave, A. Fan, U. Paquet, J. Tomczak, and C. Zhang, eds.), vol. 37, pp. 81857–81887, Curran Associates, Inc., 2024.

[227] T. Ahmed, M. Hirzel, R. Pan, A. Shinnar, and S. Sinha, "Tdd-bench verified: Can llms generate tests for issues before they get resolved?," *arXiv preprint arXiv:2412.02883*, 2024.

[228] Q. Zhang, Y. Shang, C. Fang, S. Gu, J. Zhou, and Z. Chen, "Testbench: Evaluating class-level test case generation capability of large language models," *arXiv preprint arXiv:2409.17561*, 2024.

[229] W. Wang, C. Yang, Z. Wang, Y. Huang, Z. Chu, D. Song, L. Zhang, A. R. Chen, and L. Ma, "TESTEVAL: Benchmarking large language models for test case generation," in *NAACL 2025* (L. Chiruzzo, A. Ritter, and L. Wang, eds.), Apr. 2025.

[230] K. Jain, G. Synnaeve, and B. Rozière, "Testgeneval: A real world unit test generation and test completion benchmark," *arXiv preprint arXiv:2410.00752*, 2024.

[231] Y. Wang, C. Xia, W. Zhao, J. Du, C. Miao, Z. Deng, P. S. Yu, and C. Xing, "Projecttest: A project-level llm unit test generation benchmark and impact of error fixing mechanisms," 2025.

[232] J. Liu, J. Yan, Y. Xie, J. Yan, and J. Zhang, "Fix the tests: Augmenting llms to repair test cases with static collector and neural reranker," in *ISSRE'23*, pp. 367–378, 2024.

[233] S. Rahman, S. Kuhar, B. Cirisci, P. Garg, S. Wang, X. Ma, A. Deoras, and B. Ray, "Utfix: Change aware unit test repairing using llm," *Proc. ACM Program. Lang.*, vol. 9, Apr. 2025.

[234] A. Saboor Yaraghi, D. Holden, N. Kahani, and L. Briand, "Automated test case repair using language models," *IEEE Trans. Softw. Eng.*, vol. 51, p. 1104–1133, Feb. 2025.

[235] R. Tian, Y. Ye, Y. Qin, X. Cong, Y. Lin, Y. Pan, Y. Wu, H. Haotian, L. Weichuan, Z. Liu, and M. Sun, "DebugBench: Evaluating debugging capability of large language models," in *ACL 2024*, pp. 4173–4198, Aug. 2024.

[236] N. Jiang, K. Liu, T. Lutellier, and L. Tan, "Impact of code language models on automated program repair," in *ICSE'23*, pp. 1430–1442, 2023.

[237] A. Silva and M. Monperrus, "Repairbench: Leaderboard of frontier models for program repair," Tech. Rep. 2409.18952, arXiv, 2024.

[238] S. Liu, L. Chai, J. Yang, J. Shi, H. Zhu, L. Wang, K. Jin, W. Zhang, H. Zhu, S. Guo, *et al.*, "Mdeval: Massively multilingual code debugging," *arXiv preprint arXiv:2411.02310*, 2024.

[239] G. Antal, N. Vándor, I. Kolláth, B. Mosolygó, P. Hegedűs, and R. Ferenc, "Pybughive: A comprehensive database of manually validated, reproducible python bugs," *IEEE Access*, vol. 12, pp. 123739–123756, 2024.

[240] B. Yang, H. Tian, W. Pian, H. Yu, H. Wang, J. Klein, T. F. Bissyandé, and S. Jin, "Cref: An llm-based conversational software repair framework for programming tutors," in *ISSTA'24*, p. 882–894, 2024.

[241] W. Yang, H. Wang, Z. Liu, X. Li, Y. Yan, S. Wang, Y. Gu, M. Yu, Z. Liu, and G. Yu, "COAST: Enhancing the code debugging ability of LLMs through communicative agent based data synthesis," in *NAACL 2025*, pp. 2570–2585, Apr. 2025.

[242] C. E. Jimenez, J. Yang, A. Wettig, S. Yao, K. Pei, O. Press, and K. Narasimhan, "Swe-bench: Can language models resolve real-world github issues?," *arXiv preprint arXiv:2310.06770*, 2023.

[243] J. Yang, C. E. Jimenez, A. L. Zhang, K. Lieret, J. Yang, X. Wu, O. Press, N. Muennighoff, G. Synnaeve, K. R. Narasimhan, *et al.*, "Swe-bench multimodal: Do ai systems generalize to visual software domains?," *arXiv preprint arXiv:2410.03859*, 2024.

[244] D. Zan, Z. Huang, A. Yu, S. Lin, Y. Shi, W. Liu, D. Chen, Z. Qi, H. Yu, L. Yu, *et al.*, "Swe-bench-java: A github issue resolving benchmark for java," *arXiv preprint arXiv:2408.14354*, 2024.

[245] R. Aleithan, H. Xue, M. M. Mohajer, E. Nnorom, G. Uddin, and S. Wang, "Swe-bench+: Enhanced coding benchmark for llms," *arXiv preprint arXiv:2410.06992*, 2024.

[246] D. Zan, Z. Huang, W. Liu, H. Chen, L. Zhang, S. Xin, L. Chen, Q. Liu, X. Zhong, A. Li, S. Liu, Y. Xiao, L. Chen, Y. Zhang, J. Su, T. Liu, R. Long, K. Shen, and L. Xiang, "Multi-swe-bench: A multilingual benchmark for issue resolving," 2025.

[247] S. Miserendino, M. Wang, T. Patwardhan, and J. Heidecke, "SWE-lancer: Can frontier LLMs earn $1 million from real-world freelance software engineering?," in *Forty-second International Conference on Machine Learning*, 2025.

[248] M. N. I. Opu, S. Wang, and S. Chowdhury, "Llm-based detection of tangled code changes for higher-quality method-level bug datasets," 2025.

[249] Z. Dai, B. Chen, Z. Zhao, X. Tang, S. Wu, C. Yao, Z. Gao, and J. Chen, "Less is more: Adaptive program repair with bug localization and preference learning," 2025.

[250] Z. Zhou, Z. Huang, Y. He, C. Wang, J. Wang, Y. Wu, X. Peng, and Y. Lou, "Benchmarking and enhancing llm agents in localizing linux kernel bugs," 2025.

[251] C. Liu, H. Y. Lin, and P. Thongtanunam, "Too noisy to learn: Enhancing data quality for code review comment generation," 2025.

[252] R. A. Dubniczky, K. Z. Horvát, T. Bisztray, M. A. Ferrag, L. C. Cordeiro, and N. Tihanyi, "Castle: Benchmarking dataset for static code analyzers and llms towards cwe detection," 2025.

[253] A. Yildiz, S. G. Teo, Y. Lou, Y. Feng, C. Wang, and D. M. Divakaran, "Benchmarking llms and llm-based agents in practical vulnerability detection for code repositories," 2025.

[254] Z. Li, S. Dutta, and M. Naik, "Iris: Llm-assisted static analysis for detecting security vulnerabilities," 2025.

[255] M. B. U. Ahmed, N. S. Harzevili, J. Shin, H. V. Pham, and S. Wang, "Secvuleval: Benchmarking llms for real-world c/c++ vulnerability detection," 2025.

[256] Y. Sun, D. Wu, Y. Xue, H. Liu, W. Ma, L. Zhang, Y. Liu, and Y. Li, "Llm4vuln: A unified evaluation framework for decoupling and enhancing llms' vulnerability reasoning," 2025.

[257] E. T. Liu, A. Wang, S. Mateega, C. Georgescu, and D. Tang, "Vader: A human-evaluated benchmark for vulnerability assessment, detection, explanation, and remediation," 2025.

[258] T. Brown, B. Mann, N. Ryder, M. Subbiah, J. D. Kaplan, P. Dhariwal, A. Neelakantan, P. Shyam, G. Sastry, A. Askell, *et al.*, "Language models are few-shot learners," *NeurIPS 2020*, vol. 33, pp. 1877–1901, 2020.

[259] J. Achiam, S. Adler, S. Agarwal, L. Ahmad, I. Akkaya, F. L. Aleman, D. Almeida, J. Altenschmidt, S. Altman, S. Anadkat, *et al.*, "Gpt-4 technical report," *arXiv preprint arXiv:2303.08774*, 2023.

[260] B. Roziere, J. Gehring, F. Gloeckle, S. Sootla, I. Gat, X. E. Tan, Y. Adi, J. Liu, R. Sauvestre, T. Remez, *et al.*, "Code llama: Open foundation models for code," *arXiv preprint arXiv:2308.12950*, 2023.

[261] R. Li, L. B. Allal, Y. Zi, N. Muennighoff, D. Kocetkov, C. Mou, M. Marone, C. Akiki, J. Li, J. Chim, *et al.*, "Starcoder: may the source be with you!," *arXiv preprint arXiv:2305.06161*, 2023.

[262] Y. Wang, W. Wang, S. Joty, and S. C. Hoi, "Codet5: Identifier-aware unified pre-trained encoder-decoder models for code understanding and generation," *arXiv preprint arXiv:2109.00859*, 2021.

[263] B. Hui, J. Yang, Z. Cui, J. Yang, D. Liu, L. Zhang, T. Liu, J. Zhang, B. Yu, K. Lu, *et al.*, "Qwen2. 5-coder technical report," *arXiv preprint arXiv:2409.12186*, 2024.

[264] H. Touvron, L. Martin, K. Stone, P. Albert, A. Almahairi, Y. Babaei, N. Bashlykov, S. Batra, P. Bhargava, S. Bhosale, *et al.*, "Llama 2: Open foundation and fine-tuned chat models," *arXiv preprint arXiv:2307.09288*, 2023.

[265] A. Grattafiori, A. Dubey, A. Jauhri, A. Pandey, A. Kadian, A. Al-Dahle, A. Letman, A. Mathur, A. Schelten, A. Vaughan, *et al.*, "The llama 3 herd of models," *arXiv preprint arXiv:2407.21783*, 2024.

[266] A. Q. Jiang, A. Sablayrolles, A. Mensch, C. Bamford, D. S. Chaplot, D. de las Casas, F. Bressand, G. Lengyel, G. Lample, L. Saulnier, L. R. Lavaud, M.-A. Lachaux, P. Stock, T. L. Scao, T. Lavril, T. Wang, T. Lacroix, and W. E. Sayed, "Mistral 7b," 2023.

[267] R. Just, D. Jalali, and M. D. Ernst, "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*, ISSTA 2014, (New York, NY, USA), p. 437–440, Association for Computing Machinery, 2014.

[268] K. Gu, R. Shang, R. Jiang, K. Kuang, R.-J. Lin, D. Lyu, Y. Mao, Y. Pan, T. Wu, J. Yu, *et al.*, "Blade: Benchmarking language model agents for data-driven science," *arXiv preprint arXiv:2408.09667*, 2024.

[269] M. Allamanis, S. Panthaplackel, and P. Yin, "Unsupervised evaluation of code llms with round-trip correctness," *arXiv preprint arXiv:2402.08699*, 2024.

[270] K. Deng, J. Liu, H. Zhu, C. Liu, J. Li, J. Wang, P. Zhao, C. Zhang, Y. Wu, X. Yin, *et al.*, "R2c2-coder: Enhancing and benchmarking real-world repository-level code completion abilities of code large language models," *arXiv preprint arXiv:2406.01359*, 2024.

[271] C. Zhang, X. D. Goh, D. Li, H. Zhang, and Y. Liu, "Planning with multi-constraints via collaborative language agents," in *Proceedings of the 31st International Conference on Computational Linguistics*, pp. 10054–10082, 2025.

[272] J. Wang, M. Zerun, Y. Li, S. Zhang, C. Chen, K. Chen, and X. Le, "Gta: a benchmark for general tool agents," in *The Thirty-eight Conference on Neural Information Processing Systems Datasets and Benchmarks Track*, 2024.

[273] M. Ding, C. Deng, J. Choo, Z. Wu, A. Agrawal, A. Schwarzschild, T. Zhou, T. Goldstein, J. Langford, A. Anandkumar, *et al.*, "Easy2hard-bench: Standardized difficulty labels for profiling llm performance and generalization," *Advances in Neural Information Processing Systems*, vol. 37, pp. 44323–44365, 2024.

[274] W. U. Ahmad, S. Narenthiran, S. Majumdar, A. Ficek, S. Jain, J. Huang, V. Noroozi, and B. Ginsburg, "Opencodereasoning: Advancing data distillation for competitive coding," *arXiv preprint arXiv:2504.01943*, 2025.

[275] M. L. Siddiq, S. Dristi, J. Saha, and J. C. Santos, "The fault in our stars: Quality assessment of code generation benchmarks," in *2024 IEEE International Conference on Source Code Analysis and Manipulation (SCAM)*, pp. 201–212, IEEE, 2024.

[276] L. Jing, Z. Huang, X. Wang, W. Yao, W. Yu, K. Ma, H. Zhang, X. Du, and D. Yu, "Dsbench: How far are data science agents to becoming data science experts?," *arXiv preprint arXiv:2409.07703*, 2024.

[277] C. Liu, S. D. Zhang, A. R. Ibrahimzada, and R. Jabbarvand, "Codemind: A framework to challenge large language models for code reasoning," *arXiv preprint arXiv:2402.09664*, 2024.

[278] L. Zhang, D. Zan, Q. Yang, Z. Huang, D. Chen, B. Shen, T. Liu, Y. Gong, P. Huang, X. Lu, *et al.*, "Codev: Issue resolving with visual data," *arXiv preprint arXiv:2412.17315*, 2024.

[279] S. Quan, J. Yang, B. Yu, B. Zheng, D. Liu, A. Yang, X. Ren, B. Gao, Y. Miao, Y. Feng, *et al.*, "Codeelo: Benchmarking competition-level code generation of llms with human-comparable elo ratings," *arXiv preprint arXiv:2501.01257*, 2025.

[280] B. Mehrbakhsh, D. Garigliotti, F. Martínez-Plumed, and J. Hernandez-Orallo, "Confounders in instance variation for the analysis of data contamination," in *Proceedings of the 1st Workshop on Data Contamination (CONDA)*, pp. 13–21, 2024.

[281] J. Li, N. Huo, Y. Gao, J. Shi, Y. Zhao, G. Qu, Y. Wu, C. Ma, J.-G. Lou, and R. Cheng, "Tapilot-crossing: benchmarking and evolving llms towards interactive data analysis agents," *arXiv preprint arXiv:2403.05307*, 2024.

[282] S. Zhang, H. Zhao, X. Liu, Q. Zheng, Z. Qi, X. Gu, X. Zhang, Y. Dong, and J. Tang, "Naturalcodebench: Examining coding performance mismatch on humaneval and natural user prompts.(2024)," *URL https://doi. org/10.48550/arXiv*, vol. 2405, 2024.

[283] A. Freimanis and P. Andersson Rhodin, "Tableanalyst: an llm-agent for tabular data analysis-implementation and evaluation on tasks of varying complexity," 2024.

[284] Z. Yang, S. Wang, Y. Yan, and Y. Deng, "Why stop at one error? benchmarking llms as data science code debuggers for multi-hop and multi-bug errors," *arXiv preprint arXiv:2503.22388*, 2025.

[285] A. Anand, A. Gupta, N. Yadav, and S. Bajaj, "A comprehensive survey of ai-driven advancements and techniques in automated program repair and code generation," *arXiv preprint arXiv:2411.07586*, 2024.

[286] C. Liu and R. Jabbarvand, "A tool for in-depth analysis of code execution reasoning of large language models," *arXiv preprint arXiv:2501.18482*, 2025.

[287] N. Nashid, I. Bouzenia, M. Pradel, and A. Mesbah, "Issue2test: Generating reproducing test cases from issue reports," *arXiv preprint arXiv:2503.16320*, 2025.

[288] M. Uniyal, M. Singh, G. Verbruggen, S. Gulwani, and V. Le, "One-to-many testing for code generation from (just) natural language," in *EMNLP 2024* (Y. Al-Onaizan, M. Bansal, and Y.-N. Chen, eds.), (Miami, Florida, USA), pp. 15397–15402, Association for Computational Linguistics, Nov. 2024.

[289] T. Galimzyanov, S. Titov, Y. Golubev, and E. Bogomolov, "Drawing pandas: A benchmark for llms in generating plotting code," *arXiv preprint arXiv:2412.02764*, 2024.

[290] J. Lu, T. Holleis, Y. Zhang, B. Aumayer, F. Nan, F. Bai, S. Ma, S. Ma, M. Li, G. Yin, *et al.*, "Toolsandbox: A stateful, conversational, interactive evaluation benchmark for llm tool use capabilities," *arXiv preprint arXiv:2408.04682*, 2024.

[291] M. Aghzal, E. Plaku, G. J. Stein, and Z. Yao, "A survey on large language models for automated planning," *arXiv preprint arXiv:2502.12435*, 2025.

[292] L. Yang, R. Jin, L. Shi, J. Peng, Y. Chen, and D. Xiong, "Probench: Benchmarking large language models in competitive programming," *arXiv preprint arXiv:2502.20868*, 2025.

[293] Z. Wang, S. Liu, Y. Sun, H. Li, and K. Shen, "Codecontests+: High-quality test case generation for competitive programming," *arXiv preprint arXiv:2506.05817*, 2025.

[294] T. Ridnik, D. Kredo, and I. Friedman, "Code generation with alphacodium: From prompt engineering to flow engineering," *arXiv preprint arXiv:2401.08500*, 2024.

[295] X. Wu, L. Pan, Y. Xie, R. Zhou, S. Zhao, Y. Ma, M. Du, R. Mao, A. T. Luu, and W. Y. Wang, "Antileakbench: Preventing data contamination by automatically constructing benchmarks with updated real-world knowledge," *arXiv preprint arXiv:2412.13670*, 2024.

[296] A. Wüst, W. Stammer, Q. Delfosse, D. S. Dhami, and K. Kersting, "Pix2code: Learning to compose neural visual concepts as programs," *arXiv preprint arXiv:2402.08280*, 2024.

[297] F. Zhang, L. Wu, H. Bai, G. Lin, X. Li, X. Yu, Y. Wang, B. Chen, and J. Keung, "Humaneval-v: Evaluating visual understanding and reasoning abilities of large multimodal models through coding tasks," *arXiv preprint arXiv:2410.12381*, 2024.

[298] F. Cassano, J. Gouwar, F. Lucchetti, C. Schlesinger, A. Freeman, C. J. Anderson, M. Q. Feldman, M. Greenberg, A. Jangda, and A. Guha, "Knowledge transfer from high-resource to low-resource programming languages for code llms," *Proceedings of the ACM on Programming Languages*, vol. 8, no. OOPSLA2, pp. 677–708, 2024.

[299] M. Roberts, H. Thakur, C. Herlihy, C. White, and S. Dooley, "To the cutoff... and beyond? a longitudinal perspective on llm data contamination," in *The Twelfth International Conference on Learning Representations*, 2023.

[300] C. Deng, Y. Zhao, Y. Heng, Y. Li, J. Cao, X. Tang, and A. Cohan, "Unveiling the spectrum of data contamination in language models: A survey from detection to remediation," *arXiv preprint arXiv:2406.14644*, 2024.

[301] H. Husain, H.-H. Wu, T. Gazit, M. Allamanis, and M. Brockschmidt, "Codesearchnet challenge: Evaluating the state of semantic code search," 2020.

[302] D. Lin, J. Koppel, A. Chen, and A. Solar-Lezama, "Quixbugs: a multi-lingual program repair benchmark set based on the quixey challenge," in *Proceedings Companion of the 2017 ACM SIGPLAN International Conference on Systems, Programming, Languages, and Applications: Software for Humanity*, SPLASH Companion 2017, (New York, NY, USA), p. 55–56, Association for Computing Machinery, 2017.

[303] P. Yin, B. Deng, E. Chen, B. Vasilescu, and G. Neubig, "Learning to mine aligned code and natural language pairs from stack overflow," 2018.