Title

Synthesizing Executable Specifications to Improve AI Programming Agents

ISBN

9798189278945

Author

Jain, Naman

Publication Date

2026-05-01

Peer reviewed|Thesis/dissertation

Synthesizing Executable Specifications to Improve AI Programming Agents

by

Naman Jain

A dissertation submitted in partial satisfaction of the

requirements for the degree of

Doctor of Philosophy

in

Computer Science

in the

Graduate Division

of the

University of California, Berkeley

Committee in charge:

Professor Koushik Sen, Chair
Professor Ion Stoica
Professor Joey Gonzalez
Dr. Sida Wang

Spring 2026

Synthesizing Executable Specifications to Improve AI Programming Agents

Abstract

Synthesizing Executable Specifications to Improve AI Programming Agents

by

Naman Jain

Doctor of Philosophy in Computer Science

University of California, Berkeley

Professor Koushik Sen, Chair

Modern large language models are trained on large corpora of static source code but receive limited supervision about how that code behaves at run time. This gap surfaces at every stage of the AI programming stack: benchmarks that grade with weak or missing tests, training pipelines bounded by signals that never execute the candidate code, and inference-time verifiers that select for textual plausibility rather than execution evidence.

This dissertation studies how to synthesize and exploit *executable specifications*: runnable artifacts that expose program behavior to an automated system. In this thesis, they take three main forms: tests and input generators, performance harnesses, and dynamic-analysis annotations. Once these artifacts can be generated from problems, repositories, or executions, they make behavior quantitatively measurable: agents can be evaluated by running candidate programs and comparing the observed behavior against the specified behavior. The same mechanism is useful not only for evaluation, but also for constructing training data and guiding inference. The chapters follow this idea across three settings (evaluation, training, and agent design), one for each of the three parts of this dissertation.

Part I studies synthesized executable specifications for evaluation. In LiveCodeBench, the specification is an input generator paired with a reference solution: new tests can be sampled for each live-contest problem, making evaluation less dependent on static benchmark instances. In R2E, an LLM and static-analysis context synthesize repository-level equivalence harnesses that run a candidate function against the original implementation inside its GitHub context. In GSO, the specification adds a timed workload to the correctness harness, turning optimization into a continuous executable check on which evaluated agents score below 5% at a speedup threshold calibrated against expert optimization commits in real-world open-source codebases.

Part II uses synthesized executable specifications as training signal: as rejection sampling filters, as behavior-preservation constraints, and as rewards. In R2E-Gym, executable fail-to-pass tests are paired with synthesized issue descriptions to create a procedurally

curated training environment, and supervised fine-tuning on the resulting trajectories yields 34.4% Pass@1 on SWE-Bench-Verified, a +13 8-point gain over SWE-Gym at matched model scale. Test suites can also constrain training-data cleaning: candidate rewrites are accepted only if they preserve behavior under the original tests, matching full-dataset performance with 15% of the cleaned data. DeepSWE trains an agent end-to-end using the test suite as the only reinforcement-learning reward, reaching 42.2% Pass@1 and, with hybrid test-time scaling, 59% Best@16.

Part III uses executable specifications at inference time. One verifier scores sampled patches by running synthesized reproduction tests, while another scores whole agent trajectories without execution. Individually, these verifiers reach 43.7% and 42.8% Best@26 on SWE-Bench-Verified; together, their hybrid reaches 51.0%. Syzygy synthesizes specifications at a finer granularity: top-level executions are mined into per-function equivalence tests and LLVM-derived dynamic-analysis annotations, which guide each step of a long-horizon C-to-Rust translation. This translates Google's Zopfli compression library (3,000 lines of C) into compilation-safe Rust, validated against the original over 1,000,000 equivalence inputs.

Across these settings, executable specifications provide a common way to turn execution into objective feedback for evaluation, training, and inference, and a unifying lens on AI programming agents that close the gap between static source code and runtime behavior.

Contents

List of Figures

List of Tables

Acknowledgments

Working at the leading edge of a fast-moving field has been both exhilarating and humbling. When I started in 2021, my first project generated a single snippet of a pandas program; one of my last generated entire codebases. What I am most grateful for, through all of this, is the people who walked alongside me: advisors and mentors who taught me how to think, collaborators who taught me how to build, and friends and family who taught me how to keep going. This thesis is, in every meaningful sense, theirs as much as it is mine.

First and foremost, I am immensely grateful to my advisor, Koushik Sen, for his steadfast support and mentorship. Koushik gave me the flexibility to pick my own research directions and chase the questions that excited me most. As language models continuously improved, he encouraged me to lean into the change: to take on harder problems, fail fast, and keep learning. That independence helped me grow as a researcher, and I will carry it with me long after this thesis.

I am also deeply grateful to the rest of my committee, Ion Stoica, Joseph Gonzalez, and Sida Wang. Ion consistently pushed me to look at the bigger picture: not just doing good work, but doing work that matters, and building things others find useful. It took me a while to internalize that lesson, but over time I realized that the most impactful work often requires seeking out rich and diverse perspectives. Joey brought energy to Sky and demonstrated the optimism needed to chase new ideas. I also had the chance to work closely with Sida during my internship at Meta, where I learned a lot about designing and analyzing experiments with rigor. One of our earliest conversations was about p-values and statistical significance for LiveCodeBench, and that habit of being statistically careful has shaped every evaluation and training experiment I have run since.

IIT Bombay was where I first got the chance to do research as an undergraduate, and I am deeply thankful for it. Prof. Arjun Jain took a chance on a curious second-year student and gave me my first exposure to computer vision research. I continued in machine learning with Prof. Sunita Sarawagi, whose infectious energy for connecting ideas across domains has stayed with me. I was also fortunate to be mentored by PhD students Rishabh and Abhijeet, who were always kind, welcoming, and generous with their time.

Before Berkeley, I spent two wonderful years at Microsoft Research India, working across machine learning and programming languages; that is where I started working on code generation. I am especially grateful to Natarajan Nagarajan and Prateek Jain for giving me that opportunity, and to Sriram Rajamani, Rahul Jain, Suresh Parthasarathy, Arun Iyer, and Aditya Kanade for their mentorship and the chance to learn from them. The atmosphere at MSR is what finally convinced me that grad school was the right decision, and I am grateful to everyone there.

I have been incredibly lucky to work with remarkable collaborators, each of whom taught me something different: optimism, discipline, curiosity, ambition, and above all the joy of building things together. I especially thank Manish, with whom I jointly led several ambitious projects throughout my PhD. What began as a simple question—"if writing tests is this easy, why isn't anyone doing it?"—grew into a two-year research agenda around our R2E project,

and I am proud of how we executed it. I also thank Tianjun Zhang, who has mentored me on nearly every project I have worked on at Berkeley, always sharing his insights about which directions were worth pursuing with infectious passion.

When I joined Berkeley, the AI-for-Code group was still small (we still called it ML for Code back then), and many of my closest collaborators came from outside Berkeley. I first met Alex through a DM about some overlapping work, and he quickly became both a collaborator and a friend. I connected with Wen-Ding in much the same way, mostly through the AI-for-Code reading group that Alex orchestrated. I have worked with both of them on many projects since, and they remain some of the first people I turn to for new papers, half-formed ideas, or the latest gossip.

I met Jaskirat during my internship at Meta, and we quickly became good friends and collaborators through our shared interest in coding agents, even if he still considers himself a vision researcher at heart. I have also been fortunate to collaborate with Sahil Shah from my undergraduate days at IIT Bombay; Ajaykrishna Karthikeyan and Skanda Vaidyanath at MSR; Wei-Lin Chiang, Michael Luo, Sijun Tan, Theo Olausson, and Celine Lee at Berkeley and other institutions; and to mentor two exceptionally motivated Berkeley undergraduates, King Han and Fanjia Yan.

I am deeply thankful to my Henry 5 family, Adwait Godbole and Jathushan Rajasekaran, for making our apartment welcoming and fun to live in. On the rare occasions when we were all awake at the same time, we spent evenings over movies, dinners, hikes, cooking, and the running joke of never quite buying a TV. Beyond being a flatmate, Jathushan was also a frequent partner in research and philosophical conversations, and a generous mentor over the years (not to mention generous with his space when I was moving to SF). I also collaborated with Adwait on our C-to-Rust work, sprinting through the project over two months and trading breakthrough ideas on our couch every weekend; it remains one of my fondest research collaborations, in part because we managed to pull a strictly formal-methods person into the LLM world.

Beyond collaborators, my PhD would have been incomplete without my friends. Manish Shetty, Jiwon Park, Shu Liu, Altan Haan, Shangyin Tan, Justin Wong, Tianjun Zhang, and Hao Wang at SKY quickly became my immediate support system and sounding board. My undergraduate friends Syamantak Kumar, Aman Bansal, Animesh Bohara, and Nitish Joshi remained the people I could always reach out to when I needed a break. My FAIR internship gave me a second grad-school-like community, and I now reconnect with Weijia Shi, Melanie Sclar, Thao Nguyen, Zhaofeng Wu, Amanda Bertsch, and Ram Ramrakhya at nearly every conference. I am also deeply grateful to the SkyLab staff and admins (Ivan, Katt, John, and Kailee) and to my CS graduate advisor Carissa, who helped me navigate Berkeley bureaucracy with surprising ease.

Finally, I thank my family, especially my parents and grandparents, for the sacrifices that made this journey possible long before I understood their cost. You gave me the freedom to dream, the steadiness to keep going, and a home to return to when the work felt uncertain. Whatever curiosity, persistence, and courage I have brought to this work, I learned it first from you.

Chapter 1

Executable Specifications

1.1 The Static-Code-vs-Runtime-Behavior Gap

Modern large language models (LLMs) are pre-trained on large corpora of static code from public sources such as GitHub projects, documentation, blogs, and package indexes, but receive limited supervision about runtime program behavior at run time. They model what code *looks* like: API signatures, syntactic patterns, documentation conventions, and boilerplate structures. What the pre-training pipeline rarely ingests is what happens once that code is executed: test outputs, crash reports, wall-clock measurements, memory-allocation traces, coverage maps, and sanitizer diagnostics. This thesis terms the resulting asymmetry the *static-code-vs-runtime-behavior gap*, and it is the central focus of this thesis.

This gap has practical consequences for the programming agents we build on top of these models. Code LLMs may generate syntactically plausible functions that crash on their first input. They may call signatures that do not exist or return incorrect types. We study these interface misunderstandings in Chapter 3. Code LLMs also confuse *correctness* with *performance*. They propose rewrites that preserve semantics but fail to actually reduce runtimes or memory allocations. We quantify these optimization failures in Chapter 4. Finally, code LLMs may rely on textual plausibility rather than machine-verifiable evidence. They assume a fix works based on text alone, a limitation we expose in Chapter 8. All three issues share a single root cause: the training signal never *ran* the code.

The hypothesis is that execution-derived signals encode behavioral semantics that static corpora alone cannot supply, and that synthesizing and exploiting these signals can reduce this gap. The rest of this thesis is about *how* to do so: how to synthesize *executable specifications* (tests, performance harnesses, and dynamic-analysis annotations) and how to use them to improve AI programming agents across the full stack comprising evaluation, training, and agent design. The remainder of this chapter outlines this approach. Section 1.2 defines what an executable specification is and is not in the sense used here. Section 1.3 lays out the three-part structure of the thesis across evaluation, training, and agent design. Section 1.4 gives an organization map for the rest of the dissertation.

1.2 What is an Executable Specification?

We use *executable specification* to refer to any runnable artifact that turns program behavior into a usable signal. Natural language requirements are ambiguous and cannot be verified automatically. Formal contracts (like Coq theorems) are precise but expensive to write and require specialized expertise. An executable specification is an intermediate form: it is machine-checkable by simply running the code. It acts as a procedure that takes a candidate program and produces a signal (a pass/fail verdict, a speedup ratio, or a runtime invariant) by executing the program or observing its traces. Every chapter of this thesis constructs a different type of executable specification and evaluates how that specification supports model evaluation, training, or inference.

Tests and input generators. The most familiar executable specification is a test. We use this term broadly. A *unit test* checks a fixed input and output. An *input generator* creates random and edge-case inputs to test a program more thoroughly. A *fail-to-pass regression test* captures a real-world software engineering issue (like a GitHub bug): it fails on the buggy code and passes once the agent fixes the bug. Finally, an *equivalence harness* compares a model's code against a known good reference solution by running both on the same inputs.

These different types of tests power the chapters that follow. Chapter 2 uses input generators and reference solutions to evaluate models on algorithmic problems. Chapter 3 uses equivalence harnesses to turn GitHub functions into evaluation tasks. Fail-to-pass regression tests from real GitHub issues are used to filter training data in Chapter 5 and provide reward signals for reinforcement learning in Chapter 7. Chapter 8 uses tests generated by the model itself to verify its own code. Finally, Chapter 9 uses equivalence tests to check each step of translating C code to Rust.

Performance harnesses. A *performance harness* measures runtime, memory usage, or throughput while ensuring the code still produces the correct output. Here, the specification is quantitative rather than binary. Instead of just saying "pass" or "fail", it tells us exactly how much faster or more efficient the new code is compared to the original. To do this reliably, a performance harness pairs a correctness test with a timed workload. This approach is used in Chapter 4, which evaluates whether models can optimize code by measuring their speedup against human-authored reference patches.

Dynamic-analysis annotations. The third category is less familiar, but useful: observing program runs exposes *structural* properties that static source alone does not reveal: aliasing relationships between pointers, nullability of dereferenced addresses, dynamic element sizes of opaque allocations, and the runtime types stored inside polymorphic containers. These are not input/output examples; they are *machine-checkable invariants about runtime behavior*, collected by instrumenting the program and aggregating observations across many executions. In Chapter 9, our SYZYGY system applies LLVM-14 instrumentation through a component

called SpecMiner to extract four families of annotation (`TypeInfo`, `SizeInfo`, `NullInfo`, `AliasInfo`) that disambiguate the many Rust type choices a single C pointer admits and drive the translation model toward safer Rust output.

Table 1.1: Taxonomy of executable specifications used in this thesis. The role column names the part of the pipeline the specification serves; the chapters column lists where the instance appears.

Spec type	**Role**	**Chapters**
Input generators + reference solutions	Evaluation	Chapter 2
Equivalence harnesses	Evaluation	Chapter 3
Performance harnesses	Evaluation	Chapter 4
F2P test filters, seeds, and rewards	Training	Chapter 5, Chapter 7
Tests as behaviour-preservation constraints	Training	Chapter 6
Self-generated tests and learned judges	Agent design	Chapter 8
Per-step equivalence + dynamic-analysis annotations	Agent design	Chapter 9

1.3 Three Roles of Executable Specifications

The rest of this dissertation is organized around three distinct roles that executable specifications play in building AI programming agents: what specs *measure*, what specs *supply*, and how specs are *used at inference*. These three roles become the three parts of the thesis (Figure 1.1). Part I treats specs as evaluation instruments; Part II treats them as the raw material of training pipelines; and Part III treats them as signals that agents consult while solving a task.

Specs for Evaluation

The first role of an executable specification is to make a programming task *measurable*. Part I shows how specs turn ambiguous natural-language or code-editing requests into outcomes that can be checked, timed, and compared. Depending on the task, this might mean using input generators to check algorithmic correctness (Chapter 2), equivalence harnesses to evaluate repository-level code edits (Chapter 3), or performance harnesses to quantify execution speedups (Chapter 4).

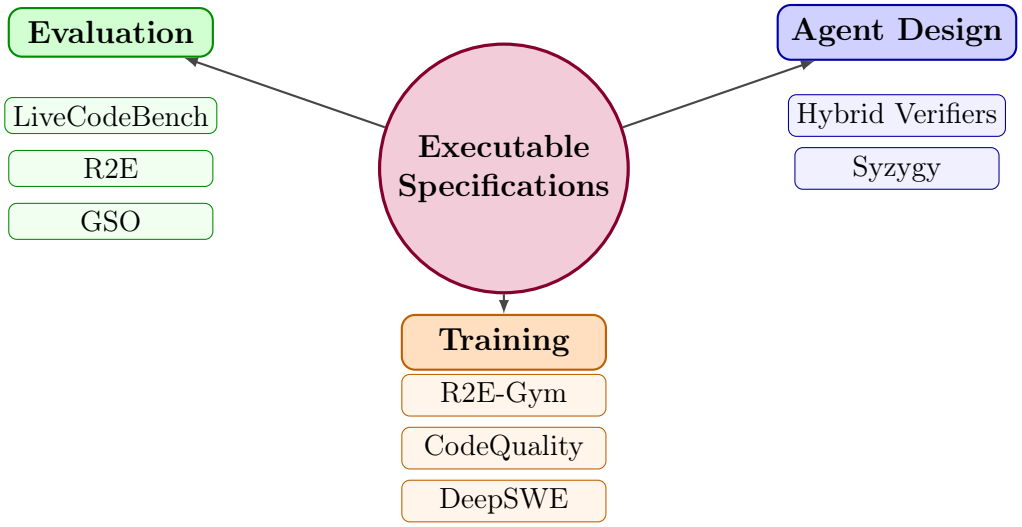


Figure 1.1: Hub-and-spoke taxonomy of executable specifications across the three thesis parts. The same underlying artifact (an executable check on program behavior) plays three roles: it measures agent capability (Evaluation), it supplies learning signal (Training), and it guides decisions at inference (Agent Design).

Specs for Training

Executable specifications also serve as training machinery. Part II explores how specs provide the raw material for learning. They can act as reward filters and environments for supervised fine-tuning (Chapter 5), as behavior-preservation constraints that prevent models from introducing bugs during stylistic rewrites (Chapter 6), and as sparse reinforcement learning rewards that enable end-to-end agent training without human supervision (Chapter 7).

Specs for Agent Design

During inference, specs become an active part of the agent's decision-making process. Part III demonstrates how agents can consult specs while solving a task. Specs can provide verifier signals that help aggregate multiple agent rollouts into a single, highly accurate answer (Chapter 8). They can also act as intermediate decomposition signals, providing local correctness checks that allow an agent to tackle long-horizon tasks one step at a time (Chapter 9).

1.4 Thesis Organization

Part I introduces three evaluation benchmarks that differ in what they measure and in what form of executable specification they use. Chapter 2 presents **LiveCodeBench**, a

continuously-updated algorithmic-programming benchmark whose executable specification is a pair of (input generator, reference solution) refreshed weekly from competition platforms; contamination is prevented by construction rather than by after-the-fact detection. Chapter 3 presents **R2E**, a framework that turns any GitHub repository into an executable test environment by synthesizing an equivalence harness around a target function; this is the setting in which interactive agents, which can actually *execute* the harness feedback, recover failures that static prompting cannot. Chapter 4 presents **GSO**, the first cross-language repository-level performance benchmark, whose executable specification has two parts — correctness tests and a performance harness — and whose metric is continuous rather than binary.

Part II shifts from measuring progress to *supplying* training signal. Chapter 5 presents **R2E-Gym**, a procedural environment generator in which fail-to-pass tests play a triple role simultaneously: filter on synthesized tasks, back-translation seed for problem statements, and reward for supervised fine-tuning trajectories. Chapter 6 presents **LMDT**, where tests are repurposed as a novel *constraint* — behaviour-preservation constraints that keep LLM-driven training-data cleaning from silently rewriting semantics. Chapter 7 presents **DeepSWE**, the first open-weight software-engineering agent trained end-to-end with pure reinforcement learning, where the entire test suite collapses to a single scalar reward signal.

Part III moves the executable-specification substrate from training back to inference. Chapter 8 presents **Hybrid Verifiers**, which combine execution-based verifiers built from self-generated tests with execution-free learned trajectory judges for inference-time scaling; it is the empirical complementarity of the two that improves over prior open-weight verifier results. Chapter 9 presents **Syzygy**, a C-to-Rust translation system that mines per-step equivalence tests and dynamic-analysis annotations — two new kinds of executable specification — to address long-horizon translation through test-driven decomposition.

Chapter 10 closes the thesis by returning to the taxonomy introduced in this chapter and previewing directions where execution-based specifications remain open research problems.

Part I

Evaluation

Chapter 2

LiveCodeBench: Holistic and Contamination-Free Evaluation

In 2023, evaluating the code capabilities of LLMs relied on a few static Python benchmarks, each with significant flaws. HumanEval and MBPP only measured basic natural-language-to-code translation on simple problems. They were also heavily contaminated. Fine-tuned open models improved on HumanEval much faster than on unseen code, and decontamination methods based on string matching were easily defeated by simple rephrasings [1, 2]. Harder competitive-programming benchmarks like APPS and Code-Contests had different problems: weak test coverage, ambiguous problem statements, and static datasets that also leaked into training corpora. This chapter addresses these issues with LiveCodeBench, a benchmark that continuously collects fresh, timestamped problems from competitive-programming contests and evaluates models across four coding scenarios.

2.1 Introduction

Code has emerged as a common application area for LLMs, with a proliferation of code-specific models [3–14] and an equally wide set of downstream uses: program repair [15, 16], optimization [17], test generation [18], documentation [19], tool use [20, 21], and natural-language-to-SQL [22]. In contrast with these rapid advancements, evaluations have remained relatively stagnant. Current benchmarks paint a skewed or misleading picture of code capability for two distinct reasons. First, popular suites like HumanEval and MBPP concentrate on a single slice of a multi-faceted skill—translating a natural-language problem statement into a short Python program—and are heavily contaminated by their inclusion in modern training corpora. Second, harder competitive-programming benchmarks like APPS and Code-Contests suffer from ambiguous problem statements, weak test coverage, and static datasets that eventually leak into training data as well. Motivated by these shortcomings, we build LiveCodeBench (LCB) on four principles.

1. **Live updates to prevent contamination.** LLMs are trained on massive, inscrutable

corpora that may include public benchmark problems. Prior decontamination attempts based on exact or fuzzy matches [8, 23] are fragile: GEMINI 1 5 documents problems that survive fuzzy filters because they were ingested in rephrased form [2], and matching is evadable by simple strategies like rephrasing [1]. We avoid reliance on string matching by collecting problems from weekly competitive-programming contests, tagging each with its *contest release date*, and restricting every model's evaluation to problems released *after* its training cutoff.

2. **Holistic evaluation.** Programming is multi-faceted: a working engineer also debugs programs from execution feedback, reasons about what an unfamiliar function does on a particular input, and writes tests that pin down expected behavior. LIVECODEBENCH evaluates four complementary scenarios – **code generation**, **self-repair** from execution feedback, **code execution** (predicting the output of a program on an input), and **test-output prediction** (producing the expected output for a problem and input without seeing a candidate program). As Section 2.5 will show, model rankings reorder non-trivially across these four axes.

3. **High-quality problems and tests.** Prior work has repeatedly surfaced insufficient tests and ambiguous problem statements in HUMANEVAL and MBPP [4, 24], with up to an 8% drop in PASS@1 when tests are strengthened. We source problems from LEETCODE, ATCODER, and CODEFORCES, whose statements and rubrics are vetted by thousands of contestants. Each problem is supplemented with tests produced by GPT-4-TURBO *input generators* whose outputs are validated against human-written accepted solutions, averaging roughly 59 tests per problem.

4. **Balanced problem difficulty.** Competition programming is challenging for even the strongest LLMS of the era, and averaging across uneven difficulty tiers reduces between-model separation [25]. We use per-platform rating brackets to classify problems as EASY, MEDIUM, and HARD, and we drop problems above each platform's rating ceiling rather than collapse them into HARD, preserving per-tier discriminative power.

With these principles in mind, we build LIVECODEBENCH, a continuously updated benchmark that avoids data contamination. Particularly, we have collected 400 problems from contests across three competition platforms – LEETCODE, ATCODER, and CODEFORCES occurring from May 2023 to the present (February 2024) and use them to construct the different LIVECODEBENCH scenarios.

Empirical Findings. We have evaluated 9 base models and 20 instruction-tuned models across different LIVECODEBENCH scenarios. Below, we present the empirical findings from our evaluations, which have not been revealed in prior benchmarks.

1. **Contamination.** We have observed a stark drop in performance of DEEPSEEK on LEETCODE problems released after Aug 2023, highlighting likely contamination in the older problems.

2. **Holistic Evaluation.** Our evaluations reveal that model performances are correlated across tasks, but the relative differences do vary. For example, the gap between open and closed models further increases on tasks like self-repair or test output prediction. Similarly, certain models like Claude-3-Opus and Mistral-L perform considerably better on code execution and test output prediction scenario compared to code generation with Claude-3-Opus surpassing GPT-4 on the test output prediction, highlighting the importance of a holistic evaluation.
3. **HumanEval Overfitting.** Upon comparing LiveCodeBench with HumanEval, we find that models cluster into two groups, ones that perform well on both benchmarks and others that perform well on HumanEval but not on LiveCodeBench (see Figure 2.5). The latter group primarily comprises fine-tuned open-access models while the former group comprises base models or closed models. This indicates that these models might be overfitting to HumanEval.
4. **Model Comparisons** (Figure 2.4)
    a) Among the open-access base models, we find that DeepSeek-Base models are the strongest, followed by StarCoder2 base and CodeLLaMa-Base. Their respective finetuned models, DS-Ins-33B and Phind-34B achieve strong performance.
    b) Closed API models such as GPT, Claude, and Gemini generally outperform open models (Figure 2.3 right). The open models that cross the performance gap are DS-Ins-33B and Phind-34B, instruction-tuned variants of large base models (both have $>$ 30B parameters).
    c) Existing benchmarks insufficiently highlight the gap between GPT-4 and other models. Particularly, smaller models achieve similar or often even better performance compared to GPT-4. In LiveCodeBench, GPT-4 (and GPT-4-Turbo) outperforms all other models (except Claude-3-Opus) with a large margin on all scenarios.

Concurrent Work. Huang et al. [26] also evaluate LLMs in a time-segmented manner. However, they only focus on CodeForces problems while we combine problems across platforms and additionally propose a holistic evaluation across multiple code-related scenarios. Li et al. [27] propose a large dataset of competitive programming problems with additional generated tests but do not study contamination or tasks beyond generation. Liu et al. [28] evaluate code comprehension capabilities of LLMs using execution task. AI [10] also evaluate DeepSeek on LeetCode problems and mention possibility of problem contamination. Singhal et al. [29] also propose evaluating LLMs on tasks beyond code generation, but they consider tasks that take into account the *non-functional-correctness aspects* of programming.

2.2 Benchmark Design

We curate our problems from three coding competition websites: LeetCode, AtCoder, and CodeForces. These websites periodically host contests containing problems that assess the

coding and problem-solving skills of participants. The problems consist of a natural-language statement along with example input–output examples, and the goal is to write a program that passes a set of hidden tests. Further, thousands of participants participate, solving these problems and thus ensuring that the problems are vetted for clarity and correctness. We have written HTML scrapers for each of the above websites to collect problems and the corresponding metadata. To ensure quality and consistency, we parse mathematical formulas and exclude problems with images. We also exclude problems that are not suitable for grading by input–output examples, such as those that accept multiple correct answers or require the construction of data structures. For each problem, we collect the tuple $P\ T\ S\ D$ of a natural-language statement P, a hidden test suite T, a reference program S, and the originating contest's release date D.

Scrolling through time. As noted, we associate the contest date D for each problem. The release date allows us to measure the performance of LLMs over different time windows by filtering problems based on whether the problem release date falls within a time window (referred to as "scrolling" through time). This is crucial for evaluating and comparing models trained at different times. Specifically, for a new model and the corresponding cutoff date (normalized to the release date if the training cutoff date is not published), we can measure the performance of the model on benchmark problems released after the cutoff date. We have developed a UI that allows comparing models on problems released during different time windows.

Test collection. Tests are crucial for assessing the correctness of the generated outputs and are used in all four scenarios. We collect public-facing tests whenever possible and use them for the benchmark. Otherwise, following Liu et al. [24], we use an LLM (here GPT-4-Turbo) to generate tests for the problems. A key difference between our test-generation approach and theirs is that instead of generating inputs directly using the LLM, we construct *generators* that sample inputs based on the problem specifications using few-shot prompting.

Generator-based test generation. We use GPT-4-Turbo to construct *input generators* that sample diverse test inputs from distributions consistent with the problem specification. Each generator produces a Python function that returns arguments for the target solution, and we execute these generators to build candidate inputs which we then validate against the collected correct programs. We use separate prompts for random and adversarial settings since contest problems often have corner cases that are missed by purely random sampling: each problem receives 2 random generators (to cover the specification broadly) and 4 adversarial generators (to target structured edge cases such as alternating extrema, boundary-length inputs, and equal-weight ties).

The prompt template is one-shot: we present GPT-4-Turbo with a curated example problem, a hand-written `construct_inputs()` function for it, and the target problem's statement, and ask the model to produce an analogous `construct_inputs()` for the target.

```
# System: You are an expert Python competitive programmer. Your goal is to
# construct input generators for testing programming-contest problems. Write
# a single function named ‘construct_inputs‘ that returns a list of diverse
# inputs sampled from the generator(s) you define, respecting the problem’s
# stated constraints.

# <USER: one-shot exemplar problem statement>
# <ASSISTANT: one-shot exemplar construct_inputs()>
import numpy as np
def random_input_generator(weight_min, weight_max, size_min, size_max):
    weights_size = np.random.randint(size_min, size_max + 1)
    weights = np.random.randint(weight_min, weight_max,
                                size=weights_size).tolist()
    k = np.random.randint(1, len(weights) + 1)
    return weights, k

def construct_inputs():
    inputs = []
    for _ in range(15): # small
        inputs.append(random_input_generator(1, 10**3, 1, 10))
    for _ in range(15): # medium
        inputs.append(random_input_generator(1, 10**6, 1, 10**3))
    for _ in range(15): # large
        inputs.append(random_input_generator(1, 10**9, 1, 10**5))
    return inputs

# <USER: target problem statement>
# <ASSISTANT: model-synthesized construct_inputs() for the target problem>
```

Figure 2.1: Sketch of the random-input-generator prompt. The adversarial variant replaces the exemplar `construct_inputs` with hand-crafted corner cases (alternating small/large weights, all-equal weights, pivots at k 1 n 2 $n-1$ n , and so on). The full one-shot demonstrations and the adversarial-generator prompt are reproduced in the appendix of Jain et al. [30].

A sketch of the random-generator prompt, with both the demonstration and the model's expected output shape, is given in Figure 2.1.

Once generators are obtained, we *execute* them to produce candidate inputs and accept only those inputs that run to completion on every reference solution without errors. For each accepted input, we compute the output by executing multiple correct user-submitted programs and take the intersection of their outputs as the oracle. Inputs on which accepted programs disagree are dropped. The accepted test set is capped at 100 inputs per problem

Platform	Total Count	#Easy	#Medium	#Hard	Average Tests
LCB (May–Feb)	400	142	168	90	59.2
LCB (Sep–Feb)	238	85	98	55	56.9
AtCoder	210	76	76	58	27.9
LeetCode	181	62	90	29	96.3
CodeForces	9	4	2	3	44.3
LCB-Easy	142	142	0	0	51.2
LCB-Medium	168	0	168	0	66.8
LCB-Hard	90	0	0	90	57.8

Table 2.1: Per-slice problem counts, difficulty distributions, and average tests per problem for the LiveCodeBench (LCB) collection. We report statistics on (a) the full May–Feb window and the primary Sep–Feb evaluation window, (b) the per-platform sources, and (c) the cross-platform difficulty subsets LCB-Easy, LCB-Medium, and LCB-Hard used in Section 2.5.

(uniformly subsampled) to keep grading cost bounded. For CodeForces, whose hidden-test displays are truncated and which contributes only 9 problems, we construct the generators semi-autonomously with human inspection of edge cases.

Problem difficulty. Competition programming has remained a challenge for LLMs, with GPT-4 achieving an average CodeForces rating (ELO) of 392, placing it in the bottom 5 percentile [25]. This makes it difficult to compare LLMs, as the variation in performance across models is low. In LiveCodeBench, we collect problems of diverse difficulties as labeled on the competition platforms, excluding problems rated above a per-platform threshold that are likely too difficult for even the best models. We use these ratings to classify problems as Easy, Medium, and Hard for more granular model comparisons.

Platform-specific curation. LeetCode contributes every weekly and biweekly contest held since May 2023; the platform provides a native {Easy, Medium, Hard} label that we use as-is. Since LeetCode provides starter code for each problem, we collect it and provide it to the LLM in the STDIN format. Since hidden tests are not directly available, we apply the generator pipeline above. AtCoder contributes the `abc` beginner rounds held after April 2023, restricted to problems with a platform rating at most 500; ratings are bucketed as [0 200), [200 400), and [400 500], and the platform's own hidden tests are used directly. CodeForces contributes a small hand-curated slice of Division 3 and 4 problems – which we find skew harder than the other two platforms even after filtering – with rating buckets 800 , (800 1000], and (1000 1300]; tests are built by semi-automatic generator construction because the platform's accepted tests are truncated in display. The code-generation and self-repair scenarios consume the full 400-problem (respectively 238-problem post-September)

slice; code execution uses 479 input–program samples distilled from 85 LeetCode problems in the May–November window; test-output prediction uses 442 instances drawn from 181 LeetCode problems. The four scenarios thereby share a common benchmark structure but consume different projections of each problem's executable specification.

2.3 Four Evaluation Scenarios

Code capabilities of LLMs are evaluated and compared using natural-language-to-code generation tasks. However, this only captures one dimension of code-related capabilities. Real-world software engineering requires expertise in tasks beyond just *generation*, such as synthesizing informative test cases, debugging incorrect code, understanding existing code, and writing documentation. These tasks are not just additional bookkeeping; they are crucial parts of the software development process and contribute to improving the quality, maintainability, and reliability of the code [31]. This also applies to LLMs, and adopting similar workflows can enable the models to perform better code generation. For example, AlphaCodium [13] is an intricate LLM pipeline for solving competition coding problems. By combining natural-language reasoning, test-case generation, code generation, and self-repair, they achieve significant improvements over a naive direct code-generation baseline, showcasing the importance of these broader capabilities. Motivated by this, we propose a more holistic evaluation of LLMs using a suite of evaluation setups that capture a broader range of code-related capabilities.

Specifically, we evaluate code LLMs in four scenarios, namely code generation, self-repair, code execution, and test-output prediction. Our selection criterion was to pick settings that are useful components in code LLM workflows and in addition have clear and automated evaluation metrics. Each problem ships with an executable specification – a natural-language statement, a reference solution, and a hidden test suite (Section 2.2) – and each scenario is a different way of *consuming* that specification. **Code generation** asks the model to produce a program the specification accepts. **Self-repair** hands the model a failing point of the specification as a concrete counterexample and asks it to mend its own earlier attempt. **Code execution** queries the specification at a single point: given a program and an input, what is the output? **Test-output prediction** inverts that query: the model must *construct* a point of the specification directly from the natural-language description, without ever seeing a program.

Code Generation. The generation scenario follows the standard natural-language-to-code setup. The model receives the problem statement together with example tests (input-output pairs), and must produce a solution that passes the hidden test suite. We report Pass@1 as the fraction of problems for which a sampled program passes every hidden test; sampling hyperparameters are fixed across scenarios and discussed in Section 2.4. The scenario uses the full pool of **400** problems, with a primary post-September-2023 slice of **238** problems used to neutralize the contamination effects analyzed in Section 2.5.

Self-Repair. Self-repair builds on prior work on LLM self-correction [16, 32, 33]. The model first attempts generation; if the attempt fails, we re-prompt it with the original problem, the incorrect program, one concrete failing test case or exception message, and the error signal. The model then returns a fixed solution. The scenario draws from the 238-problem post-September slice. One design choice matters for downstream comparisons: we report Pass@1 on the *final* program — the original generation if it already passed, otherwise the repair — rather than a repair-only delta. This makes self-repair an end-to-end metric that subsumes generation, so any improvement over plain generation can be read off as a gap between two numbers in the same column.

Code Execution. Execution adapts the output-prediction setup of CRUXEval [34] to live competition code. The model is given a Python function `f` and an input `x`, and must emit the literal output `y`; grading runs `assert f(x) == y` inside a Python interpreter. The scenario uses **479** samples drawn from **85** problems released in the May–November window. The design choice worth flagging is the source of the programs: rather than short CodeLlama-distilled snippets, we apply compile-time and run-time filters to *human-written* competition solutions to ensure samples are reasonable. The resulting functions carry the natural structure of hand-written algorithmic code (nested loops, complex numerical computations, and a large number of execution steps) and are less synthetic than CRUXEval's pool. This matters once chain-of-thought is enabled: Section 2.5 shows that closed models extract large gains from verbalized reasoning on this harder distribution while several open models regress.

2.4 Evaluation Setup

The four scenarios share a common evaluation protocol, with a small number of scenario-specific choices that we make explicit so that later numbers are unambiguous.

Metric and sampling. We report Pass@1 throughout, computed with the standard unbiased estimator of Chen et al. [3] and Kulal et al. [35]. For every problem we draw 10 candidate completions using nucleus sampling with temperature 0 2 and top-$p = 0$ 95; closed models are queried through their provider APIs and open models through vLLM [36].

Correctness checks. Each scenario has a single, automated grader. For code generation and self-repair, the candidate program is run against the associated test cases (on average 59 tests per problem, as in Section 2.2), and a sample is counted correct iff it passes every test. For code execution, correctness is decided by executing `assert f(input) == generated_output` in a Python interpreter. For test-output prediction, we parse the model's response to extract the predicted literal and equivalence-check it against the ground-truth output collected when the tests were built.

Model pool. We evaluate 29 models across various sizes, ranging from 1 3B to 34B, including base models, instruction models, and both open and closed models. From OpenAI we include `GPT-3.5-Turbo` at `0301`, `0613`, `1106`, and `0125`, `GPT-4-0314`, `GPT-4-0613`, and `GPT-4-Turbo-1106`; from Anthropic, `Claude-Instant-1`, `Claude-2`, `Claude-3-Opus`, and `Claude-3-Sonnet`; Google's `Gemini-Pro`; and `Mistral-Large`. Among open code models we cover DeepSeek-Coder at 1 3B, 6 7B, and 33B in both base and instruct form, CodeLlama at 7B, 13B, and 34B also in base and instruct form, and StarCoder2 at 3B, 7B, and 15B (base only). Five further fine-tunes build on the CodeLlama and DeepSeek-Coder bases: WizardCoder-34B-V1, WizardCoder-33B-V1.1, Phind-CodeLlama-34B-V2, Magicoder-S-CL-7B, and Magicoder-S-DS-6.7B. Approximate training cutoffs are taken from each provider's release notes and pinned per model so that the time-stratified evaluation can be applied uniformly.

Decontamination window. For the primary results, we restrict to problems released between September 2023 and February 2024, a window of 238 problems out of the 400 collected. The particular cutoff is chosen based on a sharp drop in the DeepSeek-Coder time series; we defer the argument to Section 2.5 and only note here that every main table uses this slice.

Scenario-specific setup

Base models are evaluated only on code generation, as they do not reliably follow the instruction formats required by the other three scenarios. Full prompt templates and the per-model table of release and cutoff dates are in the appendix of Jain et al. [30].

Code Generation. For the instruction-tuned models, we use a zero-shot prompt and follow the approach of Hendrycks et al. [37] by adding appropriate instructions to generate solutions in either functional or stdin format. For the base models, we use a constant one-shot example, with a separate example provided for problems that accept stdin input and for problems that accept functional output.

Self-Repair. Similar to prior work [16], we use the programs generated during the code generation scenario along with the corresponding error feedback to build the zero-shot prompt for the self-repair scenario. The types of error feedback include syntax errors, runtime errors, wrong answers, and time-limit errors, as applicable.

Code Execution. We use few-shot prompts for the code execution scenario, both with and without chain-of-thought prompting (COT). Particularly, we use a 2-shot prompt without COT and a 1-shot prompt with COT with manually detailed steps.

Test-Output Prediction. We use a zero-shot prompt that queries the model to complete assertions, given the problem, function signature, and test input.

2.5 Results

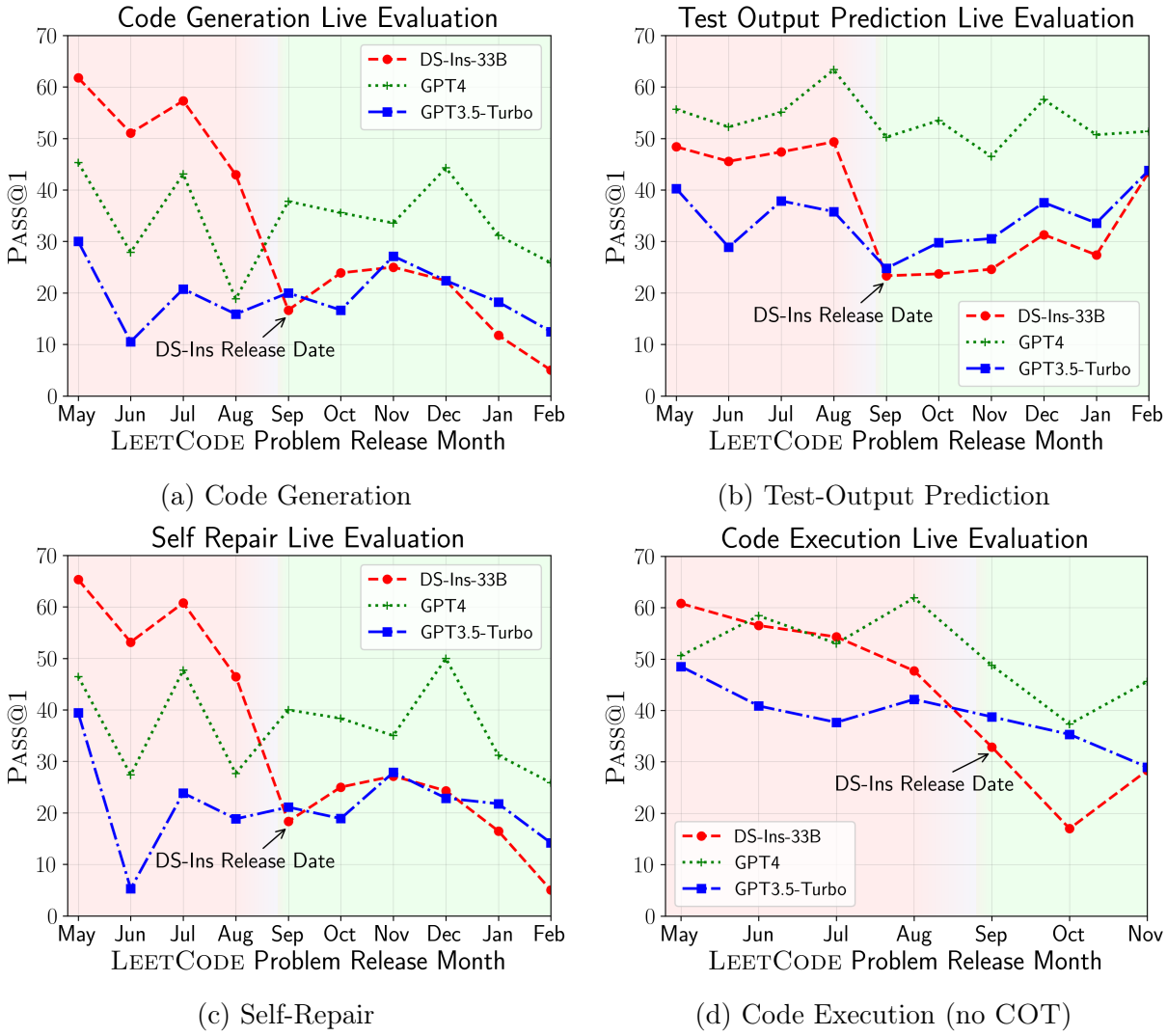
Avoiding Contamination

A distinguishing aspect of our benchmark is the ability to evaluate models on problems released over different time windows. This allows us to measure the model performance on problems released after the cutoff date, thereby giving a performance estimate on *unseen* problems.

Contamination in DeepSeek. LiveCodeBench comprises problems released since May 2023. However, DeepSeek models were released in Sep 2023 and might have already been trained on some of the problems in our benchmark. We can measure the performance of the model on the benchmark using problems released after the cutoff date, thereby estimating the performance of the model on previously unseen problems. Figure 2.2 shows the performance of DS-Ins-33B on all four LiveCodeBench scenarios on LeetCode problems released in different months from May 2023 to Jan 2024. We notice a stark drop in performance after Aug. 2023 (right before its release date), which suggests that the earlier problems might indeed be contaminated. The drop is consistent across code generation, test-output prediction, self-repair, and code execution – which makes a monthly-problem-difficulty explanation implausible. Concurrently, AI [10] (Section 4.1, last paragraph) also acknowledges the possibility of LeetCode contamination, noting that "*models achieved higher scores in the LeetCode Contest held in July and August*".

Interestingly, we find that this drop in performance primarily occurs for the LeetCode problems only, and that the model performance is relatively smooth across the months for problems from other platforms. Figure 2.3 (left) shows a relatively stable performance for all models on AtCoder problems released over different periods, with the possible exception of the months of May and June.

Performances of other models. We study performance variations in other models released recently. Particularly, GPT-4-Turbo, Gemini-Pro, Mistral-L, and Claude-3s models were released in November 2023, December 2023, February 2024, and March 2024 respectively. Note that GPT-4-Turbo (1106-preview variant) and Claude-3s have cutoff dates April 2023 and August 2023 respectively. For these models, we do not find any drastic performance variations across the months. Interestingly, we find that even the DS-Base-33B model also suffers from contamination, dropping from peak ∼60 on May problems to ∼0 on September LeetCode problems. This also points out the likely inclusion of competition problems in the pretraining corpus for the DeepSeek models.



(a) Code Generation

(b) Test-Output Prediction

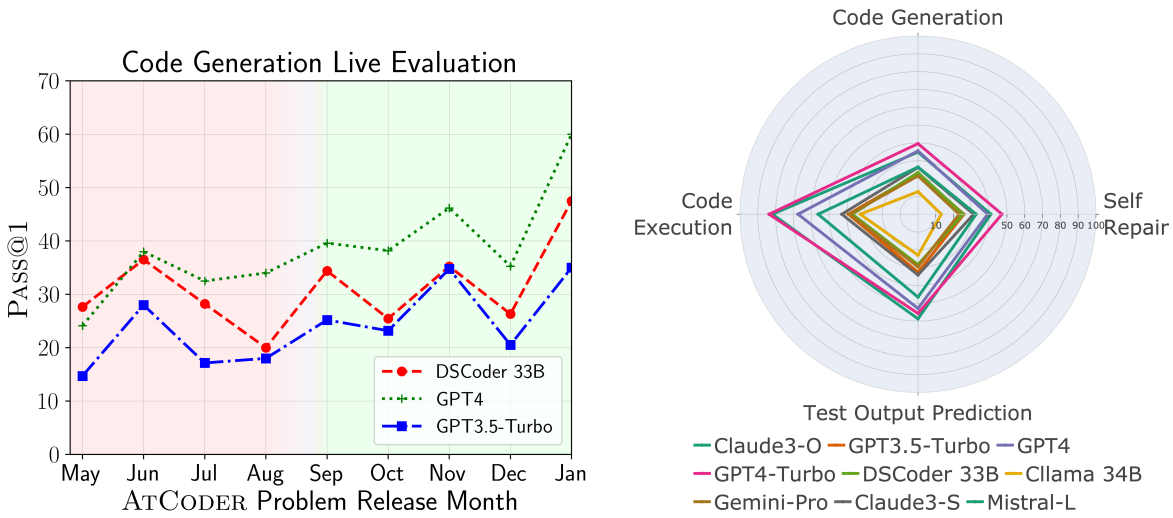
(c) Self-Repair

(d) Code Execution (no COT)

Figure 2.2: Month-by-month PASS@1 of DS models on LEETCODE problems across all four LIVECODEBENCH scenarios. Performance is high on problems released before DEEPSEEK's September 2023 release date and drops sharply on problems released immediately after, consistent with LEETCODE contamination in the pretraining corpus. The same post-August sharp drop appears across all four scenarios; code execution is currently evaluated between May and November only.

Performance and Model Comparisons

We evaluate 20 instruction-tuned models (and 9 base models used in the code generation scenario) on LIVECODEBENCH. These models range from closed access to open access with their various fine-tuned variants. To overcome contamination issues in DEEPSEEK models, we



(a) AtCoder: negative control for contamination

(b) Per-scenario performance profile

Figure 2.3: (left) Month-by-month Pass@1 on AtCoder problems: performance is essentially flat across months, confirming that the contamination signal in Figure 2.2 is specific to LeetCode. (right) Per-scenario polar chart across LiveCodeBench's four scenarios; relative rankings of models reorder across the four axes, as discussed in the results that follow.

only consider problems released since Sep 2023 for all evaluations below. Figure 2.4 shows the performance of a subset of models across the four scenarios. We highlight our key findings below.

Holistic Evaluations. We have evaluated the models across the four scenarios currently available in LiveCodeBench. Figure 2.3 (right) displays the performance of models on all scenarios along the axes of the polar chart. First, we observe that the relative order of models remains mostly consistent across the scenarios. This is also supported by high correlations between the Pass@1 metric across the scenarios – over 0 9 across all pairs. Interestingly, the correlations are larger for related tasks, 0 99 for generation and self-repair, and 0 96 for test-output prediction and code execution. This correlation drops to 0 93 for generation and test-prediction scenarios.

However, despite the strong correlation, the relative differences in performance do vary across the scenarios. For example, GPT-4-Turbo further gains performance gap over Claude-3-Opus in the self-repair scenario after already leading in the code-generation scenario. Similarly, Claude-3-Opus and Mistral-L particularly do better in tasks involving COT, particularly in the code-execution and test-output-prediction scenarios. For instance, Claude-3-Opus outperforms even GPT-4-Turbo in the test-output-prediction scenario. Similarly, Mistral-L outperforms Claude-3-Sonnet in both these scenarios after trailing behind in

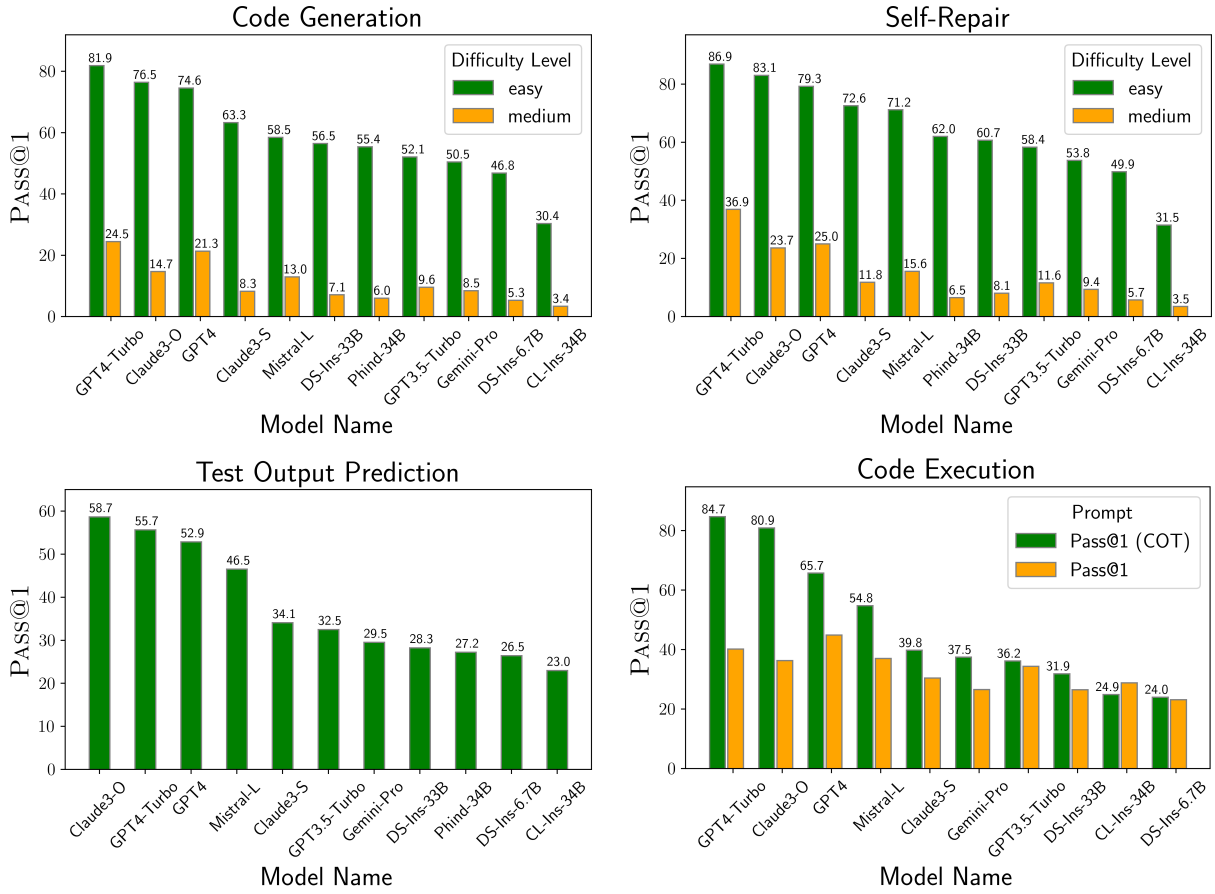


Figure 2.4: Model performances across the four scenarios available in LiveCodeBench (filtering on the September–February time window). The top-left and top-right plots depict Pass@1 of models on easy and medium splits across the code generation and self-repair scenarios respectively (results on hard subset deferred to the appendix of Jain et al. [30]). The bottom-left and bottom-right plots depict Pass@1 of models across the test-output-prediction and code-execution scenarios respectively.

code generation and repair. These differences highlight the need for holistic evaluations beyond measuring code-generation capabilities.

Comparison to HumanEval. Next, we compare how code-generation performance metrics translate between LiveCodeBench and HumanEval, the primary benchmark used for evaluating coding capabilities. Note that we use the HumanEval+ version of HumanEval problems as it is more reliable with more exhaustive test cases. Figure 2.5 shows a scatter plot of Pass@1 on HumanEval+ versus LCB-Easy code generation scenario. We find only a moderate correlation 0 77, with much larger performance variations on LCB-Easy.

Additionally, we observe that the models cluster into two groups, shaded in red and green. The models in the green-shaded region lie close to the $x = y$ line, indicating that they perform similarly on both benchmarks. On the other hand, the models shaded in red lie in the top-left region of the graph, indicating that they perform well on HumanEval+ but not as well on LiveCodeBench. Interestingly, the green-shaded cluster contains base models or closed-access models, while the red-shaded cluster primarily comprises fine-tuned variants of open-access models. The well-separated clusters suggest that many models that perform well on HumanEval might be overfitting on the benchmark, and their performances do not translate well to problems from other domains or difficulty levels like those present in LiveCodeBench.

Indeed, HumanEval is an easier benchmark with small and isolated programming problems and thus easier to overfit on. In contrast, LiveCodeBench problems are sourced from reputable coding platforms offering more challenging problems with a higher diversity and difficulty level. This potential overfitting is particularly exemplified by DS-Ins-1.3B, which achieves 59 8% Pass@1 on HumanEval+ but only 26 3% on LCB-Easy. Thus, while it scores higher than Gemini-Pro and Claude-Ins-1 on HumanEval+, it performs considerably worse on LCB-Easy. Similarly, DS-Ins-6.7B and MC-6.7B perform better than Mistral-L, Claude-2, and Claude-3-Sonnet on HumanEval+ but are considerably worse on LCB-Easy.

Highlighting the gap between SoTA and open models. One distinct observation from our evaluations is the large gap between SoTA models and open models across all scenarios. Particularly, GPT-4-Turbo, GPT-4, and Claude-3-Opus lead across the benchmarks with wide performance margins over other models. This distinguishes LiveCodeBench from prior benchmarks (like HumanEval) where various open models have achieved similar or better performance. For example, DS-Ins-33B is merely 4 3 points behind GPT-4-Turbo on HumanEval+ but 16 2 points (69%) on LCB code generation scenario. This gap either holds or sometimes even amplifies across other scenarios. For instance, consider test-output prediction and code execution (with COT), where GPT-4-Turbo leads the DS-Ins-33B model by 96% and 134% respectively.

We analyze GPT-4-Turbo, the leading model generated code samples qualitatively and find that it generates more readable code. Specifically, the code consists of more inline natural language comments that *reason* or *plan* before producing the code. We verify this quantitatively and find that GPT-4-Turbo uses 19 5× more comment tokens compared to GPT-4.

Comparing Base Models. We use three families of base models – DeepSeek, CodeLLaMa, and StarCoder2 – and compare them on the code generation scenario. A one-shot prompt is used for all models to avoid any formatting and answer-extraction issues. We find DS models are significantly better than both CodeLLaMa and StarCoder2 base models, with a DS-Base-6.7B model even outperforming both CL-Base-34B and SC2-Base-15B models. Next, we observe that SC2-Base-15B also outperforms the CL-Base-34B model (similar to

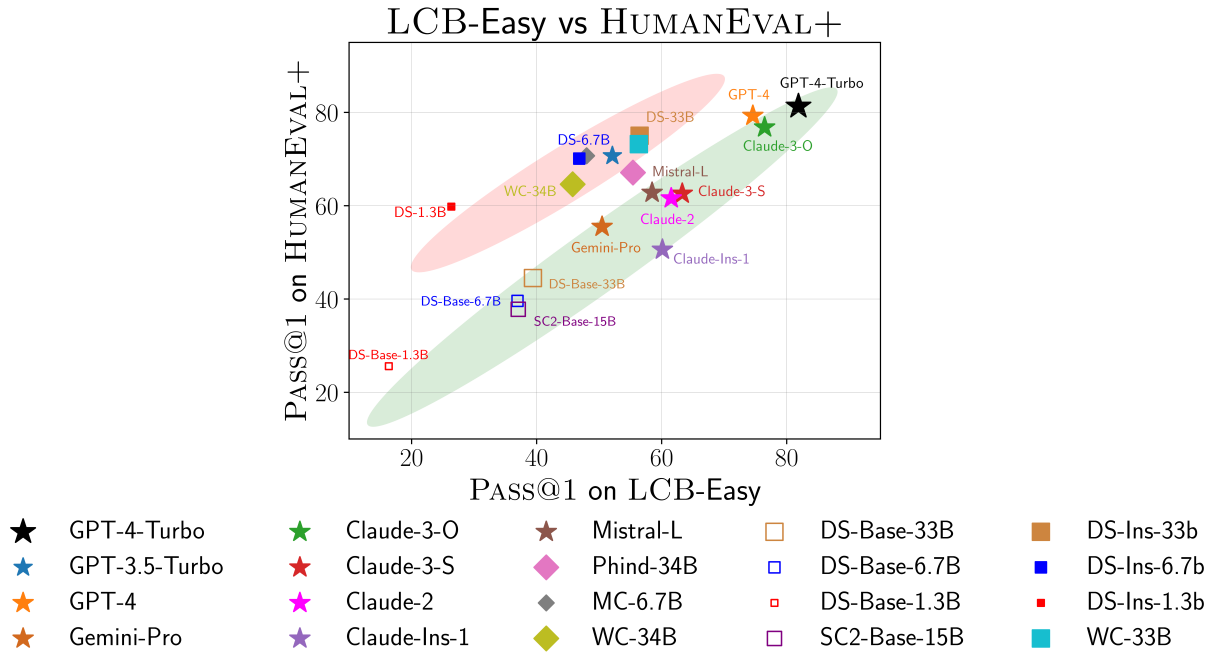


Figure 2.5: Scatter plot comparing Pass@1 of models on HumanEval+ versus Pass@1 on the easy subset of LiveCodeBench code generation scenario. Star markers denote the closed-access models while other markers denote different open model families. We find that the models are separated into two groups – the green-shaded region where performances on the two datasets are *aligned* and the red-shaded region where models perform well on HumanEval+ but perform poorly on LiveCodeBench. This indicates potential overfitting on HumanEval+ and primarily occurs in the fine-tuned variants of open-access models. For example, DS-Ins-1.3B achieves Pass@1 of 60 and 26 on HumanEval+ and the LCB-Easy subset, respectively. Thus, while it ranks above Gemini-Pro and Claude-Ins-1 on HumanEval+, it performs significantly worse on the LCB-Easy subset. Similarly, DS-Ins-6.7B outperforms Claude-2 on HumanEval+ but is 14 7 points behind on the LCB-Easy subset.

findings in [14]). Note that these differences can potentially be attributed to data curation approaches. For instance, StarCoder2 models (and potentially DeepSeeks as discussed in Section 2.5) use competition problems in the pretraining corpus, highlighting the need for diverse and clean data for training code LLMs. Similarly, training methodologies (like epochs) can also play a role in the difference, and we leave a thorough investigation of this for future work.

Role of Instruction Tuning. We find that fine-tuning improves performance on both HumanEval+ and LiveCodeBench code generation scenarios. Particularly, DS-Ins-33B and

Phind-34B achieve 23 4 and 22 3 Pass@1 on LiveCodeBench, improving over their base models by 7 3 and 9 5 points respectively. Similar gains are observed in previous benchmarks (like HumanEval+) as well. This highlights the importance of good instruction datasets for building strong code LLMs. Thus, while instruction tuning is often framed as shaping instruction-following behavior rather than adding new pretraining knowledge (e.g., [38, 39]), it is essential for improving performance, perhaps by eliciting the right model behavior.

At the same time, we note that the base models have *aligned* performances on LCB code generation and HumanEval+ benchmarks, like within or close to the green-shaded region in Figure 2.5. However, the fine-tuned models exhibit a large performance gap, with much better performance on HumanEval+. On the other hand, the closed-access models are still aligned across both benchmarks. This suggests that the fine-tuning data for open models might not be as diverse as that for closed models, leading to a lack of generalization to different kinds of problems.

Comparing open-access instruction-tuned models. Here, we compare various fine-tuned variants of the DeepSeek and CodeLLaMa base models across different model sizes. We find that fine-tuned DeepSeek models lead in performance, followed by Phind-34B and CL-Base-{7, 13, 34}B models across most scenarios. Broadly, we find that model performances correlate with model sizes. For example, Phind-34B model outperforms the 6 7B models across all scenarios.

Comparing Closed Models. We evaluate a range of closed (API access) models from different model families like GPTs, Claudes, Gemini, and Mistral. We find that GPT-4-Turbo and Claude-3-Opus rank at the top across all scenarios, followed by Mistral-L and Claude-3-Sonnet models. Finally, Gemini-Pro and GPT-3.5-turbo are the weakest among the closed models. The relative differences between the models vary across the scenarios. For example, GPT-4-Turbo demonstrates a larger improvement from self-repair (24 5 to 36 9 on the LCB-Medium problems), while Gemini-Pro only improves from 8 5 to 9 4.

Open-Access vs Closed-Access Models. In general, closed (API) access model families generally outperform the open access models. The gap is only closed by three models, namely DS-Ins-33B, WC-33B, and Phind-34B, which reach the performance levels of the closed models. For instance, in the code-generation scenario (Figure 2.4), these models reach close to or even outperform closed-access models like Gemini-Pro, GPT-3.5-turbo, and Claude-3-Sonnet. The performances do vary across scenarios, with the closed-access models performing better in test-output-prediction and code-execution scenarios. Overall, our findings confirm that a combination of strong base models and high-quality fine-tuning datasets is a viable recipe for good code LLMs.

2.6 Related Work

Language Models for Code Generation. Starting with Codex [3], there are over a dozen code LLMs. These include CodeT5 [40, 41], CodeGen [42], SantaCoder [7], StarCoder [8], AlphaCode [5], InCoder [43], and CodeGeeX [44]. As of early 2024, DeepSeek [45], StarCoder [8, 14] and CodeLLaMa [9] are the most popular open models. Many downstream models resulted from fine-tuning them on synthetically generated data, such as WizardCoder [11], MagiCoders [12], and Phind-34B.

Code generation benchmarks. Benchmarks for LLM-based code generation have proliferated along three axes. The natural-language-to-Python axis grows out of HumanEval [3], HumanEval+ [24], MBPP [4], APPS [37], Code-Contests [5], and L2CEval [46], with multilingual extensions [44, 47–49]. A second axis targets library and API usage: DS-1000 [50], ARCADE [51], NumpyEval [52], PandasEval [53], ODEX [54], and APIBench [20]. A third axis zooms out to repositories and software-engineering workflows [55–60]. Within competition programming specifically, APPS, Code-Contests, xCodeEval [61], LeetCode-Hard [32], and TACO [27] all use overlapping sources of contest problems – and a line of prompting and ensembling methods has developed alongside, from AlphaCode [5] to ALGO [62], Parsel [63], code cleaning [64], analogical reasoning [65], and AlphaCodium [13]. The biggest differentiating factor between LiveCodeBench and these benchmarks is that our benchmark is *continuously updated* to avoid contamination, and contains *more tasks* such as code repair, code execution, and test-output prediction, capturing more facets of code reasoning and code generation.

Holistic evaluation. The scenarios in Section 2.3 systematize tasks that had been studied individually. Self-repair has been investigated as a prompting strategy over generation benchmarks [16, 32, 33, 66–68]; LiveCodeBench turns it into a standard, release-date-stratified scenario with a deterministic grader. Code execution was introduced as a probe of code understanding by Austin et al. [4] and Nye et al. [69] and sharpened by CRUXEval [34]; the LiveCodeBench execution task differs by being live and by using human-written competition solutions rather than the CodeLlama-distilled programs of CRUXEval, yielding functions that are both more natural and more complex. LLM-based test generation has been explored in CodeT [70], TeCo/TestGPT [71, 72], and Methods2Test [73, 74]. LiveCodeBench's test generation scenario differs by decoupling test inputs from expected outputs, allowing input generation and output prediction to be evaluated separately.

Contamination. A growing literature attempts to *detect* test-set leakage in pretraining corpora, via direct prompting [75], time-based canaries [76, 77], statistical tests [78–80], and membership inference [81]. LiveCodeBench instead avoids contamination through live updates: weekly scraping adds problems released after fixed training cut-offs, letting users build new tests and evaluations on problems that are less likely to be contaminated.

2.7 Limitations

The evaluation presented above is honest about four limits.

Benchmark Size. The primary evaluation window is small. Restricting to problems released after the DeepSeek cutoff leaves 238 problems for code generation and self-repair, 188 for code execution, and 254 for test-output prediction. Bootstrapping 238-sized subsets from the full 400-problem pool estimates a 1–1 5% standard deviation in aggregate Pass@1, so comparisons separated by less than that should be treated cautiously; HumanEval at 164 problems has the same concern. The issue is also more acute for models with more recent cutoffs, whose evaluation slice is even smaller. Two mitigations are already underway: we scale the benchmark horizontally by adding problems from more competition platforms each week, and we plan a held-out private test set in the spirit of Kaggle that will be used when models are submitted for rigorous evaluation without exposing the problems themselves.

Focus on Python. The benchmark is Python-only. The infrastructure is language-agnostic in principle—problem statements and serialized tests port directly—so multilingual extension is straightforward once appropriate execution engines are wired in.

Robustness to Prompts. Prompts are largely untuned across models, with only minor adjustments for system-prompt format and delimiter tokens. Rankings can shift by a few points under better prompt engineering. However, we find that rankings are stable across the four scenarios and largely track HumanEval trends. Similarly, in the code-execution scenario with COT, while we notice slight performance regression for some models, this could be partly due to the prompt.

Problem Domain. Competition programming is algorithmic and self-contained. It does not substitute for real-world software engineering or repository-level tasks, where problems are open-ended, specifications are ambiguous, and cross-file context matters. LiveCodeBench is positioned as a starting point, to be combined with domain-specific evaluations that target applied settings.

2.8 Chapter Summary

LiveCodeBench shows that an executable specification — a problem's input generator together with its reference solution — can be continuously synthesized and refreshed from live competition platforms, and that live problem collection can reduce contamination risk and supports evaluation beyond generation-only tasks. In the language of this thesis, input generators are the simplest form of executable specification; the chapters that follow extend the same recipe — synthesize the spec via LLM prompting and execute-to-validate — to richer objects, equivalence harnesses in Chapter 3 and performance harnesses in Chapter 4, that move toward repository-level tasks with dependencies, multi-file context, and custom object types.

In the next chapter, we move from algorithmic problems to the repository-level setting of real GitHub repositories, where the executable specification must account for library

dependencies, multi-file context, and custom object types.

Chapter 3

R2E: Turning any GitHub Repository into a Programming Agent Test Environment

The evaluations traditionally used to certify code LLMs (such as HumanEval, MBPP, and APPS) present models with short, isolated helpers using primitive types. In contrast, a working engineer integrates new code into an existing repository, uses domain-specific library types, and runs the interpreter while writing. This chapter introduces R_2E, a framework that turns any installable GitHub repository into an executable test environment by synthesizing equivalence test harnesses, and R_2E-Eval, the benchmark instantiated from applying the pipeline at GitHub scale.

3.1 Introduction

The rapid improvement of LLMs' performance on code-related tasks has enabled the development of coding assistants deployed in the real world.

However, evaluations on such real-world coding setups have not kept pace. Prior benchmarks [3, 54] used for evaluating coding capabilities of LLMs only consist of *short and isolated* functional code-completion problems. On the other hand, real-world software engineering requires more complex workflows involving integrating code with existing (large) codebases, using libraries, interacting with the interpreter, debugging errors, etc. In this work, to capture this interactive aspect (in contrast with single-shot code generation), we consider programming agents as AI systems that can similarly use interpreters and error feedback to improve their own outputs given a specification. *As such programming agents become more powerful, it motivates the need to build real-world test environments to evaluate them.*

In this work, we propose **Repository to Environment** (R_2E), a scalable framework for turning any GitHub repository into a test environment to evaluate the performance of code generation systems on real-world scenarios (Section 3.3). *We build on a key insight*

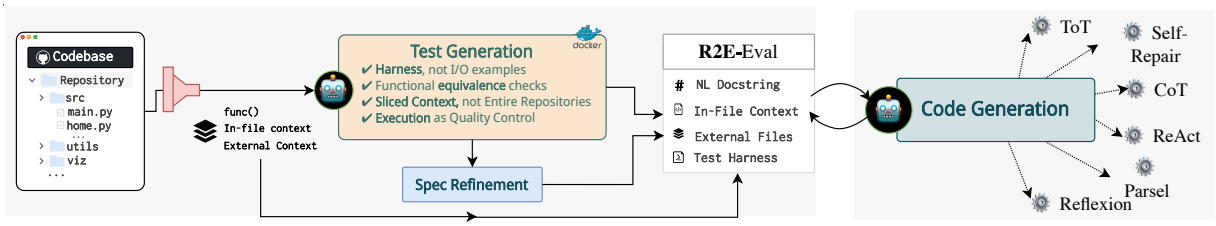


Figure 3.1: An overview of our R_2E framework that takes any GitHub repository and converts it into a programming-agent test environment. Given a repository, we first scan for *interesting* functions and collect corresponding in-file and external-file contexts from the repository. Next, we use our test-generation approach to build high-quality *test harnesses* for each function; the key insight is decoupling test outputs from inputs by relying on the ground-truth implementation to compute expected outputs. A specification-refinement step then improves the natural-language docstring, making it amenable to code generation. The framework yields problem instances comprising docstrings, test harnesses, and repository context (instantiated as the R_2E-Eval benchmark). Our benchmark can be used to evaluate static code-generation systems and interactive programming agents that use the test harness and interpreter to improve their code-generation performance on repository-level problems.

that test suites, if synthesized for real-world code, can act as checks as well as orchestrators for execution-guided programming environments. R_2E takes a function (from GitHub), constructs an equivalence test harness – a scaffold consisting of test cases and a setup that establishes dependencies for executing the function. R_2E further refines the docstring and uses the refined specification along with repository code and test harness as a problem instance for studying code generation. Figure 3.1 provides an end-to-end diagram of our approach.

These environments serve two evaluation purposes: First, a code generation system can be evaluated via the environment in these real-world scenarios. Secondly, even for an interactive programming agent, our environment can provide feedback to the agent using the interpreter. Notably, the R_2E framework is *scalable* and can be used to build web-scale open-domain coding datasets. Furthermore, R_2E requires minimal human supervision and can be updated in a *live* manner for contamination-free evaluation.

Using this framework, we construct R_2E-Eval (Section 3.4), the first large-scale benchmark of real-world coding problems consisting of natural-language docstrings, repository contexts, and equivalence test harnesses. Figure 3.2 shows an example of a function and corresponding synthesized test harness from our dataset. R_2E-Eval comprises 246 tasks extracted from 137 repositories containing 127 2 code tokens, 11 5 tests, and 3 7 dependencies per problem.

Finally, in Section 3.5, we evaluate current LLMs on real-world scenarios from our benchmark. We find that compared to HumanEval models perform worse on these problems,

[1] https://github.com/TorchDSP/torchsig

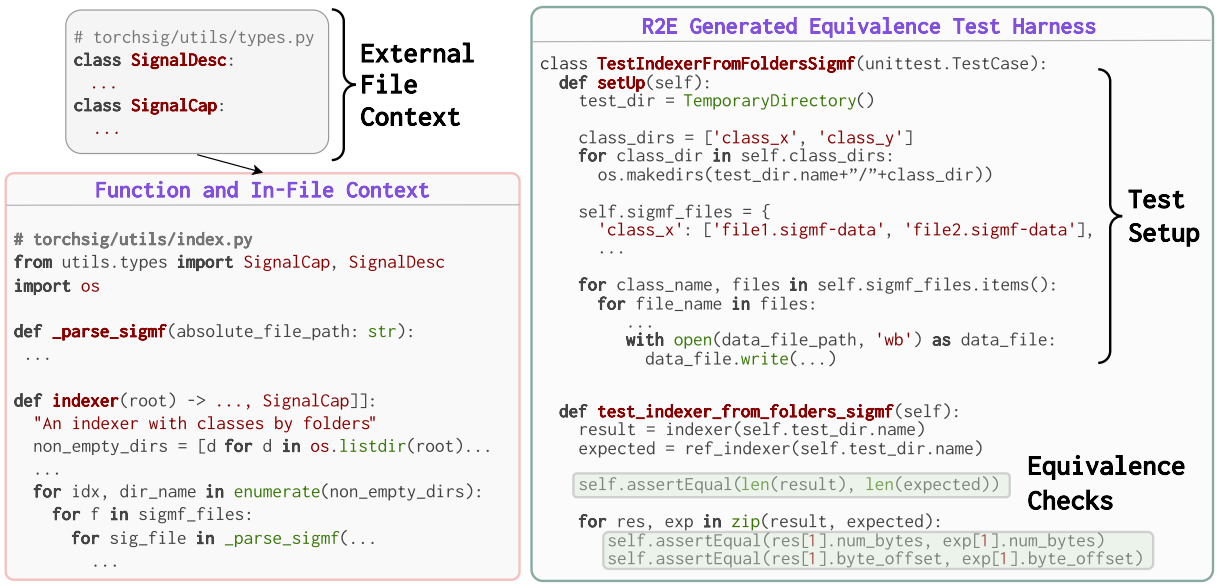


Figure 3.2: An example problem (left) in the R_2E-EVAL benchmark. The problem contains a function `indexer` from the `Torchsig` GITHUB repository.[1] `TorchSig` is an open-source signal-processing machine-learning toolkit based on the `PyTorch` data-handling pipeline. The function `indexer` has dependencies within its file (`_parse_sigmf`) and from external files (`SignalDesc` and `SignalCap` from `torchsig/utils/types.py`). On the right is the generated equivalence test harness from our R_2E framework. It contains a complex test setup where files expected by `indexer` are created and added to the file system. The test harness then performs functional-equivalence checks for various granular properties of the returned output. Our harnesses test the functional behaviour of a given function directly against the *ground truth* program available on GITHUB, instrumented in the Python namespace. Thus, we avoid predicting the expected output behaviour of a given function and only require constructing *diverse* inputs to test the function on.

highlighting the challenges of real-world programming. Popular techniques like Chain-of-Thought (COT) do not help with performance. On the other hand, LLM agents that interactively program using the test harness and execution feedback improve performance in the evaluated repair setting. We also provide insights into model behavior specific to real-world programs, such as challenges in understanding interfaces to existing functions and reasoning about complex objects.

Overall, we find that repository-level programming remains challenging, even for SOTA LLMS (GPT-4), motivating the use of better workflows that mimic a typical developer's programming process. This underscores the need to move from static functional coding to interactive programming, the evaluation of which our framework enables. Finally, R_2E environments enable collecting interaction traces for code generation, optimization, repair

which can help improve various code-related abilities of LLMs.

The triple $\mathcal{I} = (\mathcal{D}\ \mathcal{R}\ \mathcal{T})$ – docstring, repository context, synthesised equivalence harness – is the repository-level instance of the executable-specification object that recurs throughout this dissertation: Chapter 4 specialises it to performance harnesses that sit on top of correctness oracles, Chapter 5 scales it into full issue-resolution environments for training SWE-Bench-style agents, and Chapter 9 extends it to per-step equivalence tests for `C`-to-`Rust` translation.

3.2 Background

Our R_2E pipeline is powered by a synergy of program analysis and LLMs. Here, we provide background on some concepts used in the following sections.

Testing. Testing for functional correctness extends beyond mere input-output pairs, encompassing the broader dependencies that real-world software relies on. A **test harness** encapsulates this by combining *test cases* (defining inputs and expected outputs) and a *setup* (establishing the operational conditions and dependencies like configuration files). The complexity of test harnesses, as illustrated in Figure 3.2, surpasses the simple input-output examples in previous benchmarks, like HumanEval [3]. For instance, in Figure 3.2, the test harness contains the required setup of files in a directory (i.e., file system dependency) that the program expects to run successfully.

Code Coverage. The quality of tests is widely measured by its *coverage* – the fraction of code elements (e.g., statements or branches) it exercises [82]. For example, a test that executes all lines of a function is said to have *line coverage* of 100%. A high coverage is desirable to ensure a function is tested thoroughly. We use **branch coverage** to evaluate the quality of our tests as it offers a more fine-grained measure than line coverage.

Program Analysis for Slicing Context. To effectively test repository code, we must grasp the function's operational context, which encompasses the functions and global variables it interacts with. We employ **dependency slicing** to construct this context.

Callgraphs. A callgraph [83] is a directed graph where nodes represent subroutines (functions, methods, constructors, etc.), and edges denote the *calling* relationships between them. If a subroutine u invokes (i.e., calls) a subroutine v, then there is a directed edge from u to v in the callgraph. For instance, in Figure 3.2, the function `indexer` *calls* the function `_parse_sigmf_capture`. This repository abstraction enables analyzing several properties of repository code. For instance, we use it to extract the dependencies that a function relies on for its execution—a valuable property for test generation. In this work, we use the PyCG tool [84] to generate callgraphs for Python repositories.

Dependency Slicing. While callgraphs abstract direct interactions between functions, a Python function can interact with parts of the repository through global variables, too–in the same file and imported from other files. We can summarize these interactions in a **dependency slice** D_f for a function f, as the set of all functions F that f calls, and all global variables G that f accesses, both directly and indirectly.

For a function f, we define a mapping called *depends* which identifies all functions F that f calls, and all global variables G that f accesses. Then a **dependency slice** D_f is the transitive closure of all functions and global variables that f depends on, directly or indirectly.

$$D_f = \underset{(F\ \ G\ )\ \ depends^*(f)}{(F \quad G\ )} \tag{3.1}$$

Computing this slice exactly is generally an undecidable problem, but R_2E makes simplifying assumptions to make it tractable. We begin by utilizing the callgraph to identify the functions that f calls. We then use bytecode analysis to identify the set of global variables that f accesses. We add these functions and global variables to the slice and recursively repeat the process for each function in the slice.

The resulting slice D_f provides the minimal context for understanding f's behaviour and indicates its *connectivity* within the repository, quantified by the slice size D_f . We utilize this context for test generation in Section 3.3.

3.3 The R2E Pipeline

GitHub is a rich data source for *realistic* code problems, but repositories vary in buildability, maintenance status, and dependency quality. We here propose R_2E, a framework that turns any GitHub repository into a test environment to evaluate the performance of code generation systems on real-world code.

Problem Curation

Repository Curation. We scraped Python repositories on GitHub created after July 2022 that are non-forks, have at least 40 stars, and contain either a `toml` or `setup.py` file. This date aligns with the reported cutoff data for OpenAI models GPT-3.5-turbo and GPT-4, thus preventing contamination. Each repository was saved in a common `Docker` image and built using `pdm` and `pip` install commands with cross-repository package-caching whenever possible to reduce the memory footprint. Further, `pipreqs` was used to add requirements missing in the build files. The uninstallable repositories were semi-manually installed as possible, resulting in a docker image with 429 installed repositories sizing over 400 GB.

Function Curation. We first extract all functions (and methods) from the collected repositories to identify problems suitable for natural-language-driven code generation and functional correctness evaluation. Particularly, we filter out functions lacking docstrings to ensure we have a natural language prompt equivalent. We then apply keyword-based filters to exclude functions associated with GPUs, cloud-tasks, etc., since they are not conducive to standard functional correctness evaluations. We estimate the complexity of the functions using its *connectivity* (detailed in Section 3.2). We filter out functions that do not call other components in the repository. Through these stages of filtering, we collected 2073 candidate problems from our 429 repositories.

Test Harness Generation: A Key to Environments

GitHub repositories *lack high-quality tests* necessary for evaluating code generation, thus requiring automated test harness generation to collect problems scalably. If generated, these tests can act as *checks* and *orchestrators* for execution-guided programming agents. As checks, they can evaluate the functional correctness of generated code. As orchestrators, they can run the generated code, capture compiler feedback, enable repair, and more. To tackle this, R_2E synthesizes tests for arbitrary GitHub code using a novel synergy of *program analysis* with LLM prompting. Below, we summarize some of the key design choices of R_2E's test generation approach.

Harnesses, not I/O pairs. R_2E generates *equivalence test harnesses* (Section 3.2) for each function, which contain the test cases and the required setup, such as database connections, external files, configurations, etc., that makes it possible to run functions in the wild. This is a departure from I/O examples with primitive types in traditional benchmarks such as HumanEval [3]. This is necessary because real-world code often requires more than simple input arguments to run. They may need several dependencies, such as access to files, environment variables, APIs, and user-defined functions or classes.

Equivalence Tests, not Output Prediction. R_2E decouples test outputs from inputs. It instead uses the original function as a reference implementation to generate expected outputs. This key insight dramatically simplifies test generation since it removes the need to predict outputs. Consequently, we generate *equivalence tests*—they check if the outputs of the original function and the generated function are equivalent against a given set of inputs. The prompt (detailed in the appendix of Jain et al. [85]) instructs the model that a compiled reference implementation is available at `fut_module`, allowing the generated test to simply execute the reference rather than hard-coding expected outputs.

Sliced Context, not Entire Repositories. Test generation using LLMs has been effective in prior work like HumanEval+ [24] for simple isolated functions. However, in a repository setting, prompting with the function alone is insufficient, and providing the entire repository is expensive. R_2E uses a novel *dependency slicing based prompt* to extract the minimal repository context required to understand the functionality of the function under test. As described in Section 3.2, it finds functions and global variables on which the function directly or indirectly depends.

Execution and Coverage for Quality Control. Finally, recent studies have shown that execution-based benchmarks can be flawed due to low-quality tests [24]. To avoid this, we execute the generated test harnesses in the docker container built for the repository. Equivalence tests are run in "self-equivalence" mode, so the function under test and the reference implementation are the same. Inoperative harnesses due to issues like missing packages are excluded. An equivalence test harness is deemed *valid* if all the equivalence tests pass. We further emphasize the *quality* of test cases by using branch coverage (Section 3.2). This check is critical to ensure that the generated tests cover the function's complete behavior and can be used for checking functional equivalence.

We encode our design decisions as guidelines to prompt GPT-4-Turbo and use the sliced context to generate high-quality test harnesses. Figure 3.2 shows the resulting harnesses that handle complex data types and unique setups, depending on the function's requirements.

	In-File		Out-File	
Strategy	**Validity**	**Coverage**	**Validity**	**Coverage**
Naïve	41.4%	95.1%	32.1%	93.6%
Full	48.4%	95.1%	39.4%	92.5%
Sliced	**50.0%**	**96.6%**	**46.3%**	**95.0%**

Table 3.1: Test generation validity and coverage across the three prompt strategies — `Naïve` (function alone), `Full` (file and imports up to 6000 tokens), and `Sliced` (dependency slice). Results are shown for `In-File` problems, where the function depends only on context within its file, and `Out-File` problems, where it also depends on external files. Higher is better on both metrics.

Test Harness Evaluation

Experiment Setup. We evaluate equivalence test harness generation on two fronts. First, measure *validity*, i.e., does it execute the original function while passing all equivalence tests? Then, we also evaluate the *quality* using branch coverage (Section 3.2) to identify how well the tests cover the function's behavior–a critical property for equivalence checking.

We consider three strategies for test generation in a repository context: `Naïve`, `Full`, and `Sliced`. The `Naïve` strategy prompt contains the function and no context. The `Full` strategy provides the file containing the function and all files it imports (until a 6000 token limit). Finally, the `Sliced` strategy implements our proposed dependency slicing to provide the minimal context required for the function. We compare these strategies in 2 problem settings: (1) `In-File`: where the function under test depends only on the context within its file and (2) `Out-File`: where it depends on external files in the repository. We generate all tests using the state-of-the-art GPT-4-Turbo model.

Validity and Quality Results. Table 3.1 shows the results of our evaluation. Focussed context improves coverage. The `Naïve` strategy performs relatively poorly on validity (as low as 32%), but the valid test harnesses it generates have high coverage (93 6%). For example, naïvely generated tests often fail to generate correct input argument types (e.g., schemas or custom classes) due to the lack of necessary context. Broader context improves validity. On the other hand, the `Full` strategy generates more valid tests (as high as 48 4%) but has lower coverage (92 5%). This indicates that a focused context can be more effective in covering corner cases in the function, but a broader context is necessary to understand the function's dependencies. Our sliced strategy strikes a good balance between the two and

achieves the best results in validity and coverage. Overall, we observe that R_2E's dependency slicing-based strategy generates $\approx 50\%$ valid test harnesses with a high $\approx 95\%$ code coverage.

Failure Modes. We collected and classified invalid equivalence test harnesses, and study their failure modes. We discovered that 40% of errors were due to `ValueErrors` and `TypeErrors`, reflecting improper key, attribute, or type usage in tests. Additionally, 15% were `DataFormatErrors`, caused by incorrect data formats or schemas, highlighting the complexity of testing GitHub code beyond primitive types. `AssertionErrors` (expected and actual outputs don't match) accounted for a notable 25% of errors, showing a nuanced aspect of functional correctness in real-world code. Although R_2E simplifies this to equivalence tests, assertions often need more granularity than simply checking for equality. For example, checking for class attributes, columns in a dataframe, etc., requires a deeper understanding of code and repository context. Lastly, `EnvironmentErrors` (21%), like OS and File system errors, indicate challenges with test environment configuration.

Refinement of Specifications

Natural language docstrings in GitHub repositories might be ambiguous or under-specified to be used for code generation. Here, we propose an automated approach to refine the natural language docstring of a given function by asking the model to refine the docstring in a self-instruct-like fashion [39]. Distinctly, however, we provide the model with additional context in the form of the original docstring, test harness class, argument types, and serialized input-output arguments available via the test harness. The prompt explicitly instructs the model to describe the format and types of expected inputs and outputs without adding verbosity or detailed programming logic. We note that while we cannot evaluate the quality of refined specifications, we perform rigorous manual evaluations and filter problems with poor or ambiguous specifications.

3.4 R2E-Eval Benchmark

In Section 3.3, we showed that R_2E enables a scalable framework for building execution-based test environments for programming agents. R_2E takes a function from a codebase and converts it into a tuple $\mathcal{I} = \mathcal{D}\ \mathcal{R}\ \mathcal{T}$, where $\mathcal{D}$ is a refined docstring for a function, $\mathcal{R}$ is the remainder of the codebase, and $\mathcal{T}$ is the generated test harness.

We instantiate this framework to construct R_2E-Eval, the first large-scale dataset of *real-world* code-generation problems with functional-correctness tests. Table 3.2 compares R_2E-Eval against several popular benchmarks used to evaluate the code-generation capabilities of LLMs. Prior work like HumanEval [3] and ODEX [54] support execution-based metrics but for isolated simple problems with no real-world repository setting. Recent work on repository-level code generation like CrossCodeEval [86], RepoBench [56], and RepoEval [87] uses repository context, but either foregoes execution-based evaluation or depends heavily on

[2]Zhang et al. [87] did not release associated tests

Dataset	Exec?	Repo?	Auto?	LOC	#Tests
HumanEval [3]				6 26	6 6
ODEX [54]				2 05	1 9
CrossCodeEval [86]				1 0	-
RepoBench [56]				1 0	-
RepoEval-Func [87]				10 8	-[2]
R_2E-Eval				10 5	11 5

Table 3.2: Comparing R_2E-Eval with other NL-to-code generation benchmarks, in terms of test execution-based support (Exec?), use of repository context (Repo?), and the number of lines in the ground truth function (LOC). R_2E-Eval is the only executable benchmark, has repository context, and is automated, enabling scalability. Additionally, our benchmark contains more tests (harnesses) per function with diverse input types and quality assurance.

Feature	Value
# Problems (# Repos)	246 (137)
Avg. # lines (# tokens)	10 5 (127 2)
Avg. # tests (coverage)	11 5 (92 2)
Avg. # dependencies	3 7
# Unique APIs	70
# Unique Arg Types	118

Table 3.3: Statistics for problem instances in our R_2E-Eval.

human-written tests, which are seldom available at scale on GitHub. R_2E-Eval is the only executable benchmark that has repository context and is automated, enabling scalability. Following, we describe the construction of R_2E-Eval and analysis.

Benchmark Construction

Dataset Quality

We emphasise heavily on the quality of problems in this work. Quality, here, means how well the function, docstring, and test cases are written. To ensure this, we only consider functions with high branch coverage. Our final benchmark problems have an average of 11 5 test cases with 92 2% average branch coverage. An additional round of manual inspection helps us select high-quality problems. Notably, in the manual inspection, we avoid very long or complex functions that are hard to specify precisely using docstrings (like functions with many peculiar corner cases).

Dataset Composition

We also consider the diversity and interestingness of the problems in the benchmark. We identify several properties of code that calibrate interestingness, such as the number of dependencies, argument types, lines, libraries used, etc.

Table 3.3 showcases statistics of our benchmark. Our manual analysis also shows that R_2E-Eval problems are diverse in terms of the domains they cover: pythonic operations (`list`, `str`, `dict` manipulations), data manipulation (`JSON`, files, `pandas`, `numpy`), algorithm and protocol implementations (`networkx`, statistics), domain-specific problems (programming languages, networks, quantum computing, formal verification, numerical computing), and more.

We also ensure that the benchmark is diverse in terms of the number of distinct repositories, preventing bias towards a codebase or domain. Overall this process leads to a curated set of 246 problems from 137 repositories in R_2E-Eval.

Each problem instance $\mathcal{I}$ can be used to evaluate a code-generation system by providing docstring $\mathcal{D}$ to the system and evaluating its response (in the context of the repository $\mathcal{R}$) against the generated test harness $\mathcal{T}$.

3.5 Code Generation Experiments

We conduct experiments to understand three important problems about LLM performance on real-world coding. (i) How well can current LLMs solve real-world code generation tasks statically? (ii) What are the typical LLM failure modes? (iii) How do programming agent paradigms (like self-repair) perform against static programming?

Our results show that the SOTA LLM model (GPT-4) can only achieve $\sim 50\%$ performance in R_2E-Eval, despite high accuracy on HumanEval. Throughout the analysis, we find that LLMs struggle at *understanding* interfaces to existing functions and reasoning about complex objects. Finally, we compare static coding approaches (e.g., COT) with the proposed interactive programming paradigm, showing gains in the evaluated interactive setting.

Metric and sampling. We use Pass@1 to evaluate functional correctness, computed by generating 5 candidate completions for each problem instance and computing the fraction that passes against the test harness. Sampling uses top-$p = 0\ 95$ and temperature $T = 0\ 2$.

Prompting. The code-generation prompt asks the model to return a single fenced code block that completes the target function, given the refined docstring $\mathcal{D}$ and a retrieved slice of the repository $\mathcal{R}$. For instruction-tuned models (GPT-4, GPT-3.5-turbo) we use a chat-format system + user message. For the CodeLLaMa base models we use the same content rendered as a code-completion prefix. The exact templates follow Olausson et al. [16] and are reproduced in the R2E appendix of Jain et al. [85].

Models. We evaluate two closed instruction-tuned models – GPT-4 (`gpt-4-1106`) and GPT-3.5-TURBO (`gpt-3.5-turbo-1106-16k`) – and three open base models from the CODELLAMA-Python family at 7B, 13B, and 34B parameters.

Contamination. GPT-4 and GPT-3.5-TURBO have a cut-off date of 2021 and are therefore not contaminated on our benchmark since we curate our problems from repositories created after August 2022 (see Section 3.3).

Static Code Generation

First, we study direct code generation on the R_2E-EVAL dataset, i.e., using code generation without interaction. Owing to the test harnesses generation approach, we perform functional correctness evaluations for the generated code. This contrasts with prior works [56, 86] that rely on execution-free exact-match metrics to evaluate code completion in the repository setting, which can be unreliable and restrict the scope of the evaluation.

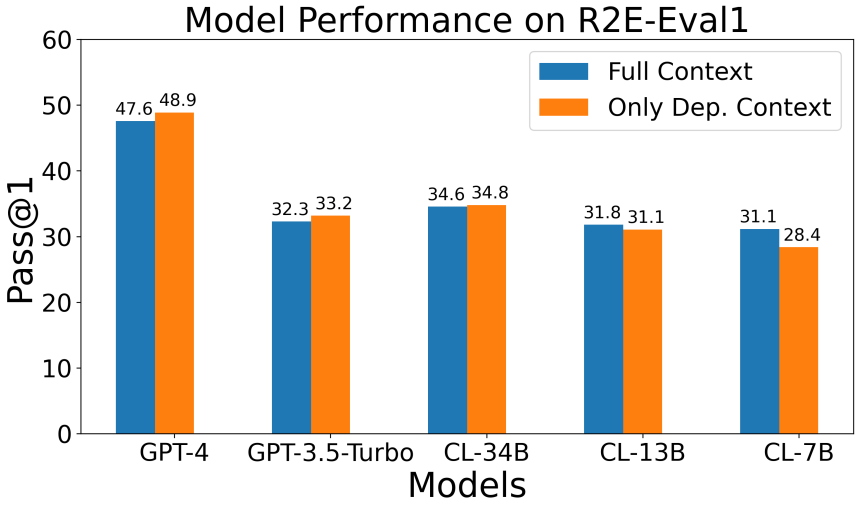


Figure 3.3: Functional correctness (PASS@1) of the GPT and CODELLAMA families on R_2E-EVAL. All models score well below their reported HUMANEVAL numbers, and GPT-4 is the only model to reach the 50% mark. The two bars per model compare *dependency* (oracle-sliced helper definitions only) and *full* (up to 6 000 tokens of surrounding file and imports) retrieval; aggregate PASS@1 is within ±1% but the per-instance correlation is only 0 48.

Figure 3.3 compares the performance of various models on our benchmark using the PASS@1. We find that the performance of various models is relatively lower than other benchmarks like HUMANEVAL. This is expected since our benchmark represents more challenging real-world problems collected from GITHUB, which require *understanding* existing context from the

repository before generating the code. We find that GPT-4 performs significantly better than all other models with a PASS@1 close to 50% whereas other models only achieve PASS@1 in the vicinity of 30%.

Effect of retrieval. We study the effect of function-definition retrieval vs. function-usage retrieval using dependency slicing (Section 3.2) on the ground-truth function. Specifically, dependency-only-context only provides the necessary definitions, while the full context setting adds the remainder of the file and other files until a 6000 token limit. Figure 3.3 compares the two settings.

The two retrieval methods perform similarly, achieving ±1% of each other's performance across most models. On a closer look, we find non-overlapping problems with a Pearson correlation coefficient of 0 48. We find that dependency-only-context vs full-context provides an interesting trade-off. On the one hand, dependencies provide a more focused view of relevant function implementations to the model. At the same time, function usage (present in full context) is often reused and enables models to *copy it directly*. We provide a more detailed discussion and examples of this trade-off in the supplementary material of Jain et al. [85].

Effect of COT. We study better-prompting strategies and look at both zero-shot and two-shot COT prompts that sketch a *plan* for the function implementation before generating the code. We study this for the instruct GPT-3.5-TURBO and GPT-4 models but find that COT-style prompting does not improve performance over direct prompting (Table 3.4).

	Base	COT-0-shot	COT-2-shot
GPT-3.5-TURBO	48.9	45.8	–
GPT-4	33.2	33.0	28.8

Table 3.4: Effect of chain-of-thought prompting on PASS@1 for a subset of R_2E-EVAL. Neither zero-shot nor two-shot COT improves over direct prompting; two-shot COT costs GPT-4 roughly four points.

Self-Repair Agent

So far, we described model evaluations on our benchmark using the direct code generation approach. However, testing harnesses and access to the interpreter allow us to evaluate programming agents that can interact with the interpreter and get feedback. Specifically, we instantiate a self-repair agent that uses the test harness.

We study that when provided with feedback from (oracle) testing harnesses (present in our benchmark instances), can LLMs correct their own mistakes? We sample 56 and 48

instances from our benchmark for GPT-4 and GPT-3.5-TURBO on which the models do not generate a correct solution. We consider the incorrect programs generated by the models as the initial programs and then provide the models with error feedback using the harness iteratively for 5 iterations.

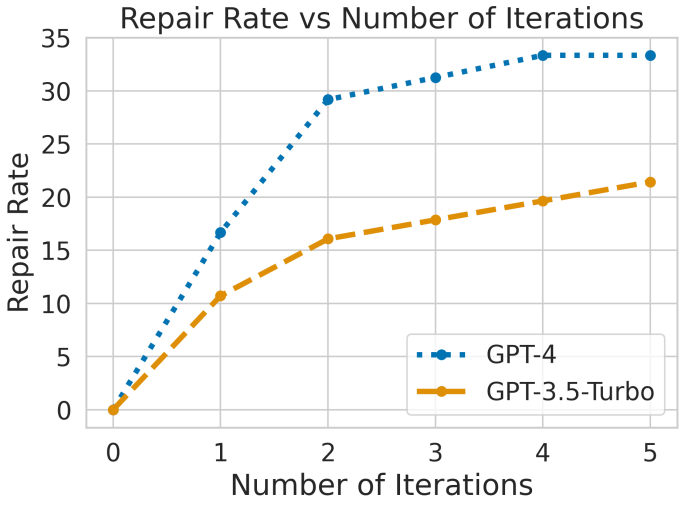


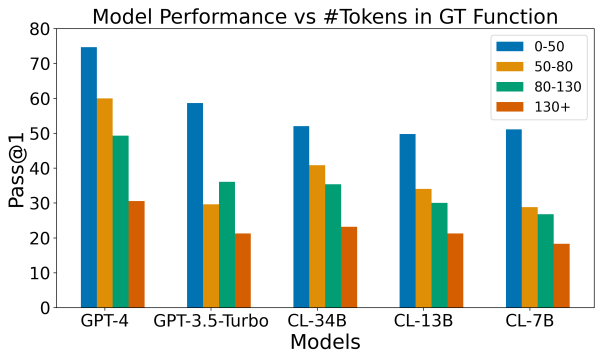
Figure 3.4: Self-repair rate as a function of iteration for GPT-4 and GPT-3.5-TURBO, measured on the subset of problems they fail at PASS@1. After five rounds, GPT-4 recovers 33% of its failures and GPT-3.5-TURBO recovers 20%. Because we subsample failing instances, iteration 0 is 0% by construction.

Figure 3.4 shows the self-repair rate of the models on our benchmark as a function of the number of iterations. First note that since we subsample only the failing instances where models do not generate correct solutions, the 0-th iteration score is 0% for both models. Next, we find that GPT-4 attains a maximum self-repair rate of 33% while GPT-3.5-TURBO only attains a maximum self-repair rate of 20%. This highlights that using execution, interpreter, and test cases, programming *agents* can improve code generation. Note that while advanced prompting techniques do not improve performance (Table 3.4), using an interpreter enables programming agents to achieve higher self-repair rates.

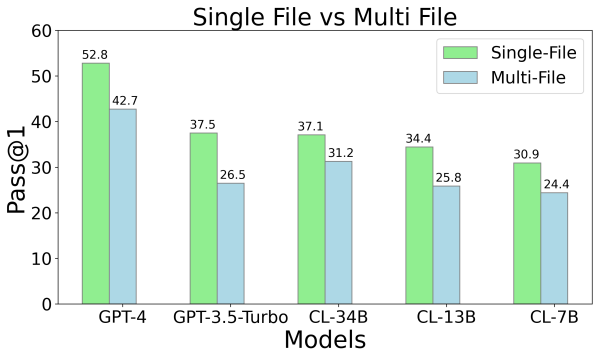
3.6 Qualitative Analysis

Performance with problem-complexity. We measure the complexity of a problem instance using (1) the number of tokens in the ground-truth implementation, (2) the number of dependencies used by the ground-truth implementation [3]. We find that both these measures are (inversely) correlated with the PASS@1 of the models. In Figure 3.5, we plot the PASS@1

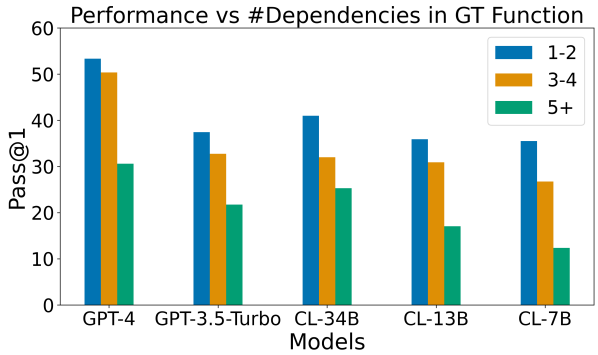
[3]counted using the number of unique functions or global variables used in the function body.



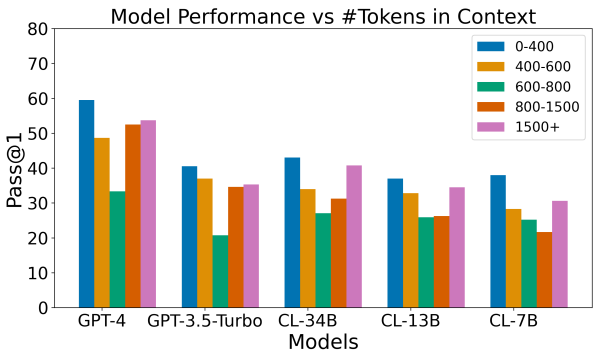
(a) Ground-truth tokens



(b) File usage



(c) Dependency count



(d) Retrieved context tokens

Figure 3.5: PASS@1 as a function of problem complexity. Longer targets, more dependencies, and multi-file usage correlate with lower accuracy.

of the models against the number of dependencies and the number of tokens used in the ground-truth implementation showing a downward trend.

Single File vs Multi-File Context. We compare how models perform on problems that require only a single file to be generated against problems that require multiple files to be generated. Model performance is significantly better on single-file problems than multi-file problems (Figure 3.5). This suggests that a.) models struggle with multi-file contexts compared to single-file contexts and b.) problems in the multi-file category are more complex than single-file problems in our benchmark, also observed in practice.

Do not *understand* the interface to provided functions. We find that when provided with complex functions in the context, LLMs do not understand the right input-output behavior of such functions and pass in wrong inputs or expect wrong outputs. Thus, even strong LLMs like GPT-4 make mistakes when provided with complex functions in the context.

```python
import torch

def _product_attr(vision, text, alter):
    """
    Args:
        vision: N x D
        text: M x D
        alter: N x M, to replace results in some cases
    Returns: N x M.
    """
    vision = vision.unsqueeze(1)
    text = text.unsqueeze(0)
    num_attr = text.sum(-1)
    queried_attr = vision * text
    queried_attr = queried_attr.masked_fill(text == 0, 1)
    queried_attr = torch.float_power(queried_attr.prod(dim=2),
        1 / torch.max(num_attr, torch.ones_like(num_attr))).float()
    no_attr_queries = num_attr.squeeze(0) == 0
    queried_attr[:, no_attr_queries] = alter[:, no_attr_queries]
    return queried_attr

def obj_with_attributes(input_embeddings, query_embeddings,
                        n_obj, n_part, n_attr):
    vision = input_embeddings[:, :n_obj]
    text = query_embeddings[:, n_obj:n_obj + n_attr]
    alter = input_embeddings[:, n_obj + n_attr:]
    queried_attr = _product_attr(vision, text, alter)
    obj_score = (input_embeddings[:, :n_obj]
                 * query_embeddings[:, :n_obj]).sum(dim=1, keepdim=True)
    scores = torch.sqrt(obj_score * queried_attr)
    return scores
```

```
RuntimeError: The size of tensor a (5) must match the size of
tensor b (2) at non-singleton dimension 2
```

Figure 3.6: GPT-4 misreads the shape convention of `_product_attr` when completing `obj_with_attributes` and supplies tensors whose non-singleton dimensions disagree, producing a runtime shape mismatch.

See Figure 3.6 for reference. This motivates that if provided access to execution context, programming agents can *understand* such interfaces and perform better.

Repeat vs Reuse Code. Abstractions are an integral part of writing good code. LLMs, however, tend to duplicate code instead of using existing context. Specifically, when provided with some existing function in the context, models re-implement the same functionality instead of directly using it. Figure 3.7 provides an example. This aligns with findings on how copilot affects code quality [88].

```python
def standardize_and_get_inchi(mol, options='', log_level=None,
                              warnings_as_errors=False) -> str:
    """Return InChI after standardising molecule and inferring stereo."""
    mol = deepcopy(mol); mol = assert_sanity(mol)
    mol = remove_isotopic_info(mol)
    has_pose = mol.GetNumConformers() > 0
    if has_pose: RemoveStereochemistry(mol)
    mol = RemoveHs(mol)
    try: mol = neutralize_atoms(mol)
    except AtomValenceException:
        mol = Uncharger().uncharge(mol)
    mol = add_stereo_hydrogens(mol)
    if has_pose: AssignStereochemistryFrom3D(mol, replaceExistingTags=True)
    with CaptureLogger():
        inchi = MolToInchi(mol, options=options, logLevel=log_level,
                           treatWarningAsError=warnings_as_errors)
    return inchi

def check_identity(mol_pred, mol_true, inchi_options=''):
    """Check if two molecules share the same InChI representation."""
    ## CodeLlama-13B-Python completion (excerpt): inlines the whole
    ## standardization pipeline twice instead of calling
    ## standardize_and_get_inchi.
    mol_pred = deepcopy(mol_pred); mol_true = deepcopy(mol_true)
    mol_pred = assert_sanity(mol_pred); mol_true = assert_sanity(mol_true)
    mol_pred = remove_isotopic_info(mol_pred)
    mol_true = remove_isotopic_info(mol_true)
    # ... twelve more duplicated lines per molecule ...
    inchi_pred = standardize_and_get_inchi(mol_pred, options=inchi_options)
    inchi_true = standardize_and_get_inchi(mol_true, options=inchi_options)
    results = _compare_inchis(inchi_true, inchi_pred)
    return {'results': results}
```

Figure 3.7: CodeLLaMa-13B completes `check_identity` by inlining the entire standardization pipeline twice instead of calling the in-context helper `standardize_and_get_inchi`. The duplicated block is elided with an ellipsis for space; the original completion is verbatim from the `rdkit`-based `check_identity` problem in R_2E-Eval.

3.7 Related Work

Code Generation Benchmarks. Code generation is primarily evaluated using functional correctness and has been explored in multiple domains. HumanEval [3] and MBPP [4] study code generation on isolated single-function problems. APPS [37] and Code-Contests [5] benchmarks are primarily used for evaluating algorithmic code-generation capabilities. DS-1000 [50], Arcade [51], NumpyEval [52], and PandasEval [53] study data-science API code generation. More recently, Wang et al. [54] proposed ODEX that evaluates coding on

APIs with human-written input-output examples. These works evaluate code-generation capabilities in isolated settings devoid of surrounding context or dependencies from other files. In contrast, R_2E coding problems are curated directly from GitHub thus more similar to real-world setups. InterCode and WebArena provide general environments for domain-specific interactive programming and web tasks respectively. We provide a framework and environments for interactive general-purpose programming tasks extracted from GitHub.

For the repository setting, prior works have primarily focused on execution-free evaluation metrics like exact-match and BLEU due to the absence of test harnesses. CoNaLA [89] curated a large dataset from StackOverflow with paired natural language and program snippets. Shrivastava et al. [57, 90] study different context-selection methods for prompting and training LLMs for repository-level code generation. RepoEval [58], RepoBench [56], and CrossCodeEval [86] study repository-level code *completion*. However, these works only evaluate short context code-generation capabilities without execution or functional-correctness restriction to short completions. In contrast, we synthesise function-level test harnesses using our novel test-generation approach and use them for performing functional-correctness checks on repository code. Recently, Jimenez et al. [60] proposed SWE-Bench to evaluate whether LLMs can solve GitHub issues. However, they assume test case availability from pull requests preventing scalable collection of problems. Our test-harness synthesis in contrast allows collecting problems from a diverse set of repositories (137 repositories vs 12 repositories). Finally, Du et al. [91] proposed ClassEval, manually curated for evaluating LLMs.

Other code-related tasks. Beyond code generation, tasks like self-repair [16, 33, 66–68], test generation [73, 74], execution [4, 34, 92], and optimisation [17] have been studied. These enable various *agentic* setups as CodeT [70], Key et al. [93], Parsel [94], FunSearch [95], Reflexion [32], LEVER [96], CodePlan [97], AlphaCodium [13], ReAct [98], and ToT [99].

3.8 Discussion and Limitations

Limitations. Natural language is inherently ambiguous and docstrings might not specify the corner cases properly. We tried to mitigate this effect with our specification-refinement approach along with manual filtering. Future work should study this ambiguity in more detail and also look into better interaction mechanisms. Next, we use observational equivalence to check whether the model-generated candidates are correct over a set of inputs. We use branch coverage as a metric for evaluating tests but it is still a softer check. Future work can apply mutation testing and oversampling to provide further confidence in generated tests.

Conclusion. We propose R_2E, a scalable framework to convert GitHub repositories to programming-agent test environments. R_2E-Eval, constructed via this framework, can evaluate both static and interactive code-generation systems, offering valuable insights into model behaviours and the need for better programming workflows. Prior work has applied rejection sampling and reinforcement learning to improve coding capabilities of LLMs [64, 100, 101]. We believe R_2E can enable such attempts for real-world programs.

The equivalence-harness design introduced here — static-analysis-sliced context, LLM-generated test program, self-equivalence validation, coverage-based quality control — is adapted by Chapter 4 for performance optimisation and by Chapter 5 for full SWE-agent training environments.

The next chapter turns from evaluating repo-level correctness to evaluating repo-level performance, where the executable specification acquires a quantitative component — a reference-timed workload on top of the correctness harness — and where the evaluation changes from *matching* a reference implementation to *exceeding* a reference runtime.

Chapter 4

GSO: Evaluating Software Optimization Agents

Correctness is a binary specification, but production software engineering is rarely binary. Codebases like `vLLM`, `OpenCV`, `VERL`, `pandas`, and `llama.cpp` have recurring runtime and throughput requirements. The commits that improve their performance are carefully engineered optimizations that cross from `Py` into `Cython`, `C`, `C++`, and occasionally `Rust`, sometimes using hand-written `SIMD` intrinsics. Therefore, we should evaluate the SWE-Agents in this study not just on whether they can fix a bug, but whether they can produce a patch that is behaviorally equivalent and at least as fast as a human expert's optimization. This chapter introduces GSO, a benchmark of 102 such tasks across 10 popular open-source codebases spanning 5 languages (Python, `Cython`, `C`, `C++`, and `Rust`), where 63% of the tasks demand edits outside pure Python. Each task pairs correctness tests with a performance harness and is graded by the $\text{Opt}_p@K$ metric: a candidate patch counts as a success only if it is functionally equivalent on all tests and achieves at least a fraction p of the speedup the human commit delivered. At the main-paper threshold $p = 0\,95$, every evaluated agent scores below 5% on Opt@1, and Opt@10 stays under 20% even for the strongest model. This capability gap remains wide despite test-time scaling and persists sharply at the Python/non-Python boundary.

4.1 Introduction

High-performance software is critical for modern computing systems, from data-analytics frameworks to machine-learning infrastructure. Developing such systems demands specialized expertise in algorithmic optimization, hardware-aware programming, performance analysis, and reasoning across multiple layers of the software stack. The complexity of these tasks is evident in production-critical systems like `vLLM` [36], `OpenCV` [102], and `VERL` [103], where teams dedicate substantial efforts to iterative and continuous maintenance over long development cycles. Simultaneously, SWE-Agents are gaining rapid traction in software development,

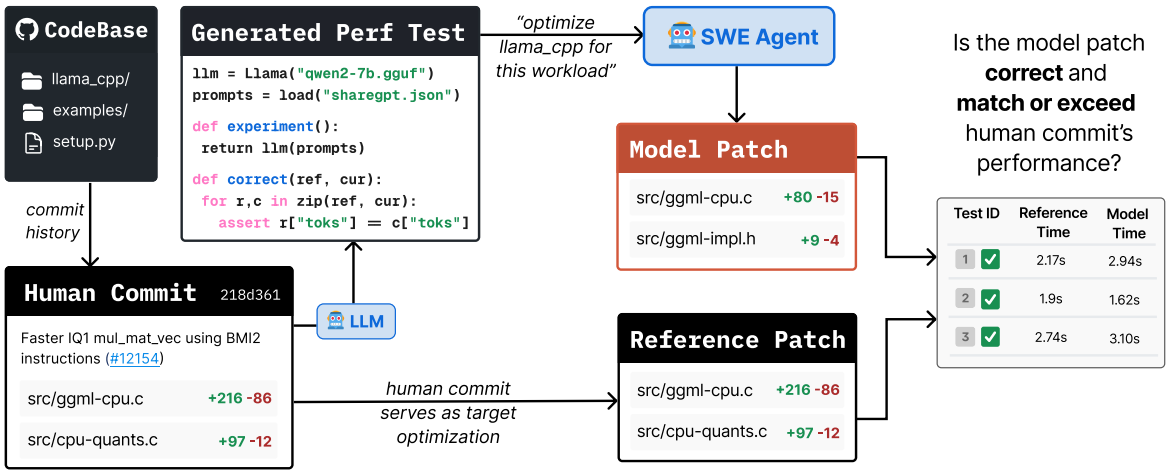


Figure 4.1: **An example GSO task.** We develop an automated pipeline that generates performance tests and analyses repository commit histories to identify real-world code-optimization tasks. Each task consists of a codebase, performance tests, and the expert developer commit that serves as the performance target for the optimization problem. LLM-based SWE-Agents are tasked with generating optimization patches using the performance test as a *precise specification* for the optimization problem. We evaluate the patches for both correctness and runtime efficiency, measuring whether they match or exceed the human expert optimization performance while ensuring equivalence.

demonstrating remarkable results on simple bug-fixing tasks [104]. This has also spurred excitement in adapting LLMs to aid in automating research tasks themselves, for example improving deep-learning kernels [105]. In this work, we study the question: *"Can LLM agents aid in the development of high-performance software?"* To answer this, we introduce GSO, a benchmark for evaluating SWE-Agents on challenging software-optimization tasks.

To create GSO, we develop an automated pipeline that generates performance tests and runs them across a repository's commit history to identify substantial optimizations discovered by expert developers. After careful manual curation, we extract 102 challenging tasks across 10 codebases, spanning diverse domains and languages including PYTHON, `C`, and `SIMD`. Each task consists of a *codebase*, *performance tests* exercising real-world workloads, and a target *optimization* from expert developer commits. SWE-Agents receive a performance test as task specification and must produce an optimization patch that improves runtime efficiency while maintaining correctness. We evaluate these patches using our OPT@K metric, providing reliable assessment in a machine-agnostic manner. Rather than naively measuring machine-dependent speedups, we assess whether model-generated patches can *consistently* match or exceed the performance of human expert optimizations.

Our benchmark evaluates the capabilities needed for high-impact optimization work, tracking usefulness for real-world high-performance software development. Particularly, problems in GSO evaluate challenging systems-engineering tasks, including optimising `Pandas`

operations, `Pillow` image or video processing operations (like GIF animation), and `llama-cpp` model inference runtimes.

Code optimization uniquely bridges algorithmic reasoning and systems engineering, providing a challenging yet well-specified evaluation domain for LLM-based programming agents. Unlike bug-fixing SWE benchmarks that rely on potentially ambiguous natural-language specifications [106], performance tests natively provide precise specifications for both correctness and efficiency. Our tasks require substantial code changes, with gold patches containing 4–15× more lines edited than previous benchmarks (Figure 4.2, middle panel). We evaluate leading LLMs on GSO using the state-of-the-art OpenHands agent framework [107] (Section 4.5). Our evaluation reveals that most agents struggle with the benchmark, achieving less than 5% success rate measured by Opt@1, with test-time compute also providing only modest improvements (Opt@10 remaining around 15%).

To understand why SWE-Agents struggle with GSO, we perform a qualitative analysis of agent behaviour and failure modes (Section 4.7). First, agents struggle with low-level languages, often avoiding them entirely or introducing fatal errors. Second, agents resort to superficial optimizations ("lazy optimizations") like compiler-flag manipulation or input-specific fast-path insertion, often making bizarre non-idiomatic code changes. Third, localization remains challenging: agents frequently misdiagnose the root cause of performance issues, leading to ineffective optimization attempts.

The key contributions of this work are:

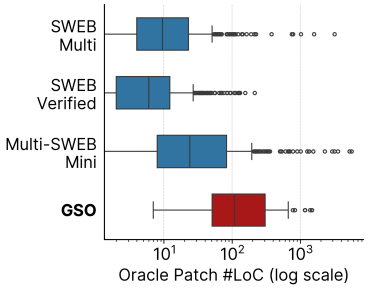
1. An automated pipeline leveraging test generation and execution information for generating software-optimization tasks from real-world codebases, resulting in the GSO benchmark.
2. Evaluation of leading SWE-Agents on GSO, revealing a substantial performance gap in systems-engineering tasks.
3. Qualitative analysis of agent behaviour and failure modes with directions for future research.

Given the substantial performance gap, we believe considerable progress in reasoning capabilities and SWE-Agents will be required to close the gap and hope GSO serves as a valuable resource for future LLM-based programming-agent research.

4.2 GSO

Global Software Optimization (GSO) is a benchmark for evaluating SWE-Agent capabilities for aiding in high-performance software development. Each task consists of an initial codebase snapshot, performance tests measuring runtime and correctness, a build script for environment setup, and a reference human commit establishing the target performance threshold. The goal is to generate a patch that improves the performance of the codebase while maintaining functional correctness.

Benchmark	Repo Level	Evaluate Runtime	Multi-Lingual	Precise Specification
HumanEval				
EvalPerf				
ECCO				
LiveCodeBench				
KernelBench				
SWE-Bench-Verified				
SWEB-Multi				
MultiSWE-Mini				
GSO (Ours)				



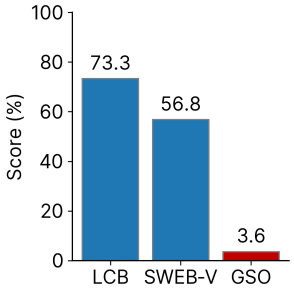


Figure 4.2: **Benchmark Feature Comparison and Performance Gap.** Left: Depicting how GSO improves over existing benchmarks across key dimensions. Middle: Distribution of oracle LoC changes across benchmarks, showing GSO solutions require over 4-15x larger edits than existing benchmarks. Right: Performance comparison of O4-Mini across LCB (algorithmic puzzles), SWE-Bench-Verified (repository-level bug-fixes), and GSO depicting the performance gap on optimization tasks.

Task Formulation

Input. The agent receives the initial codebase, build script, and a performance test serving as input and is tasked with correctly improving the runtime on the given workload in a generalizable manner.

Output. The agent produces a unified patch that implements the required performance improvements.

Evaluation. We apply the generated patch and execute all associated performance tests. Success requires that the patch (1) applies cleanly, (2) passes all correctness checks, and (3) matches or exceeds the target human commit's performance.

Distinctive Features of GSO

Precise task specification. GitHub issues provide ambiguous specifications, especially for complex software engineering tasks [106]. GSO employs performance tests as specifications that provide executable optimization targets, enabling rigorous evaluation.

Unifying algorithmic coding with real-world SWE. Code LLM research is divided across isolated but algorithmically-focused benchmarks, and simple bug-fixing based SWE benchmarks. GSO combines algorithmic challenges with real-world software tasks.

Diverse tasks spanning system boundaries. 63% of tasks demand non-Python modifications across five programming languages, reflecting production environments where performance-critical components leverage systems languages beneath high-level interfaces (Figure 4.2-right).

Challenging tasks via strong human targets. Each task centers on human-authored commits averaging 108 lines, establishing demanding optimization targets requiring sophisticated code comprehension and algorithmic reasoning.

Unbounded performance measurement. Software optimization inherently enables unbounded performance improvements through identification of previously unexplored bottlenecks. Speedups thus serve as a critical secondary metric for quantifying exceptional performance beyond human optimization targets. Since task-specific factors can skew raw speedup metrics, we establish OPT@K as our primary metric while providing comprehensive speedup analysis.

Evading contamination. Contamination represents a fundamental concern for agent benchmarks, particularly with real-world codebases potentially present in pretraining data. Our speedup metric provides a continuous signal that systematically detects potential contamination between human and model patches. We posit that models substantially outperforming human-written patches provide evidence against simple memorization, though they do not rule out contamination.

4.3 Benchmark Construction

Unlike prior benchmarks that rely on manually written issues and test cases, we develop an automated pipeline to construct GSO tasks from GITHUB repositories. Our key insight is that software optimization problems can be identified by executing tests across commit boundaries and measuring performance improvements with minimal human curation. Therefore, we use LLMS to identify performance-related commits, generate performance tests, and execute them to identify optimization tasks. Particularly, we use the following two-stage pipeline:

Stage I: Identifying Performance Improving Commits. We scan popular open-source GITHUB repositories using an LLM-based judge with code-change heuristics to identify performance-related commits. For each candidate, we extract context including relevant files, commit messages, linked issues, pull requests, and endpoints exercising the affected code. This efficient filtering process handles large commit volumes while gathering the rich context needed for test generation.

Stage II: Generating and Executing Performance Tests. We generate performance tests via execution-based rejection sampling using an LLM prompted with the commit context. Tests exercise the codebase with real-world workloads, e.g., generating completions from `qwen-7b` for the `ShareGPT` dataset using LLAMA.CPP. They measure runtime, and verify equivalence between the pre- and post-commit codebase states via assertions on the outputs. We retain commits showing significant performance improvements across multiple test cases.

Final Curation. We perform a careful manual review of the automatically collected candidates to ensure the benchmark's quality and diversity. We remove instances with weak tests or reproducibility issues, selecting problems spanning various optimization techniques, difficulty levels, and application domains. During this phase, we also identified and removed tasks susceptible to reward hacking, such as those where trivial `@lru_cache` or `@memoize` decorators matched the human speedup, or where repeated calls within the test script invited simple output caching.

4.4 Designing $\textsc{Opt}_p@K$ Metric

Evaluating code optimization presents unique aggregation challenges absent in traditional code generation benchmarks. Common aggregate metrics do not address two critical issues: (1) different tasks have varying baseline performance levels, making cross-task comparison and aggregation difficult, and (2) within tasks, tests with disparate speedup magnitudes can considerably skew aggregate metrics.

Robust Speedup Calculation. Prior work aggregates per-test speedups using geometric mean, but this approach is vulnerable to outliers. A model achieving speedups of [`0 1 1000`] across two tests yields a geometric mean of `10`, despite degrading performance on one test. In Section 4.7, we show that agents indeed perform such optimizations and thus can exploit the geometric mean. Drawing from systems optimization literature [108], we compute speedup using the harmonic mean of individual test speedups which is more robust to extreme positive outliers. Let $s_i = \frac{T(C_1\ i)}{T(C_2\ i)}$ denote the speedup on test i, where C_1 and C_2 represent two codebase states and $T(C\ i)$ denotes runtime on test i. We then define the overall speedup as the harmonic mean:

$$S(C_1\ C_2) = \frac{n}{\sum_{i=1}^{n} \frac{1}{s_i}} = \frac{n}{\sum_{i=1}^{n} \frac{T(C_2\ i)}{T(C_1\ i)}}$$

Relative Performance Evaluation. To enable cross-task comparison, we evaluate model patches against human-authored optimization targets rather than absolute speedups against the original codebase. For each task, we measure whether the model achieves performance comparable to expert developers. Thus, we measure the speedup against the human target as $S(C_h\ C_a)$, where C_h is the codebase state from the human target optimization and C_a is the codebase after applying the model's patch. For each task, we define success using both performance and correctness criteria:

$$\textsc{Opt}_p = \begin{cases} \text{true} & \text{if } S(C_h\ C_a) \geq p \text{ and correct}(C_a) \\ \text{false} & \text{otherwise} \end{cases}$$

The first criterion ensures that the model's patch achieves at least p fraction of the human speedup. The second criterion (correct(C_a)) ensures functional equivalence through test assertions.

Final Metric Definition. We compute $\textsc{Opt}_p@K$ as the fraction of tasks where at least one successful solution exists among K attempts:

$$\textsc{Opt}_p@K = \frac{1}{N} \sum_{i=1}^{N} \mathbf{1}(\ k \quad [K] : \textsc{Opt}_p)$$

We estimate confidence intervals following established methods for pass@K metrics [3, 109]. Our $\textsc{Opt}_p@K$ metric provides machine-independent assessment by comparing against human baselines rather than absolute speedups. While raw speedups vary significantly across machines, the relative evaluation ensures consistent assessment across different hardware configurations. Finally, we denote $\textsc{Opt}@K$ as the $\textsc{Opt}_{0\,95}@K$ metric that uses a 95% threshold for evaluating success against the human target.

Within the context of this thesis, this relative, continuous metric serves as the quantitative executable specification for optimization tasks, extending the binary correctness predicates of earlier chapters.

4.5 Evaluation Setup

Machine Configuration. We use Docker to containerize the task environment for each task in GSO. The initial codebase is cloned and installed into a local environment in the container before providing it to the agent. All tasks are run on a single Google Cloud `n2-standard-64` VM (64 vCPUs, 256 GB memory). While raw speedups may vary across machines, we empirically find that measuring $\textsc{Opt}@K$ is resilient to machine variations, provided each task gets sufficient resources.

Agent Scaffold. We use OpenHands [107] (`CodeActAgent-v0 35 0`) as our common agent scaffold for all models and experiments. The scaffold provides access to a file-editor tool and a bash terminal tool to the agent to perform code changes and execute commands. To support lengthy and frequent codebase rebuilds (in the case of `C` or `C++` code changes), we configure the agent with a 3-hour time limit per task and a 20-minute timeout per step. Our task-specific prompt instructs the agent to optimize the runtime of the specification performance test and also contains the build and test commands. The supplementary material of Shetty et al. [110] includes the complete agent prompts and details.

Models. We evaluate GPT-4o, O3-Mini, O4-Mini, Claude-3.5-v2 (referred to as `claude-3.6`), Claude-3.7, and Claude-4.0. Our experiments focus on two settings: Opt@1 (Section 4.6) and inference-time scaling (Section 4.6). For Opt@1, we sample 3 rollouts (trajectories) at temperature $T = 0\ 1$. For inference-time scaling, we limit our evaluations to O4-Mini and Claude-3.5-v2 due to API rate limits and high cost, and sample rollouts at temperature $T = 0\ 8$ to encourage trajectory diversity.

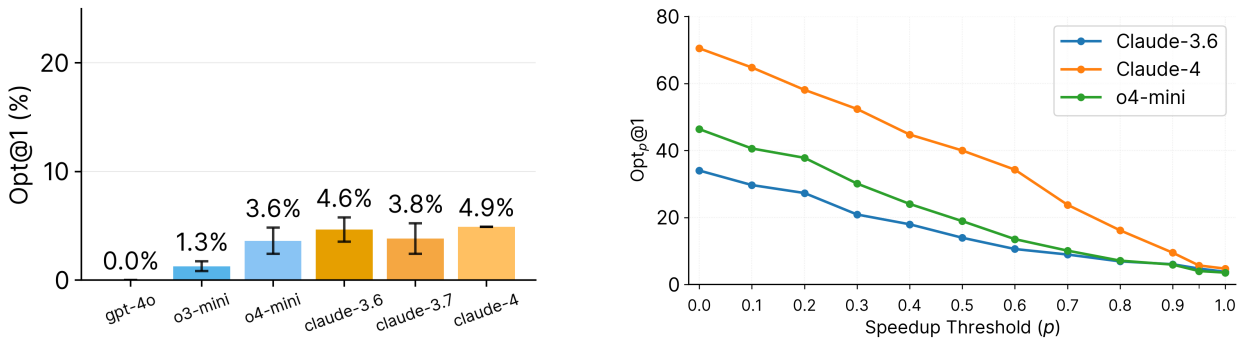


Figure 4.3: **OPT@1 performance.** (a) Left: OPT@1 (speedup threshold p set to 0 95) across models, with all models achieving less than 5% success. (b) Right: OPT$_p$@1 indicating the portion of problems where model patches match p fraction of the human commit's performance. We find that the strongest performing models remain strong throughout, with success rates reducing as it becomes more challenging to match human-level performance.

4.6 Results

OPT@1

Figure 4.3-left shows consistently poor OPT@1 performance across agents based on all models, confirming software optimization as a significant challenge for current SWE-Agents. Even the best performing model, CLAUDE-4.0, achieves less than 5% success, while GPT-4O fails completely at 0 0%. These results suggest that success on SWE-Bench-like benchmarks does not necessarily transfer to more challenging real-world tasks like software optimization, which require both algorithmic reasoning and engineering expertise.

We next vary p in OPT$_p$@1 (Figure 4.3-right). Recall that OPT$_p$@1 evaluates whether the agent's patch is able to match p fraction of the human commit's performance. Thus $p = 0$ evaluates whether the agent's patch is correct, regardless of its performance, while $p = 1$ evaluates whether the agent's patch matches the human commit's performance, increasing in difficulty. We find that OPT$_0$@1 performance shows considerably more variation, with CLAUDE-4.0 achieving 70% OPT$_0$@1 while O4-MINI achieves 45%. We also find that the trend stays with the strongest performing model but the gap compresses as p increases, indicating challenges in matching human-level performance.

Scaling Inference-time Compute

Drawing inspiration from Olausson et al. [111], we examine two dimensions of test-time compute scaling: (1) sampling multiple trajectories and picking the best (referred to as parallel compute), and (2) allowing more steps per trajectory (referred to as serial compute).

Scaling serial vs parallel compute. In Figure 4.4-left, we analyse step-count scaling

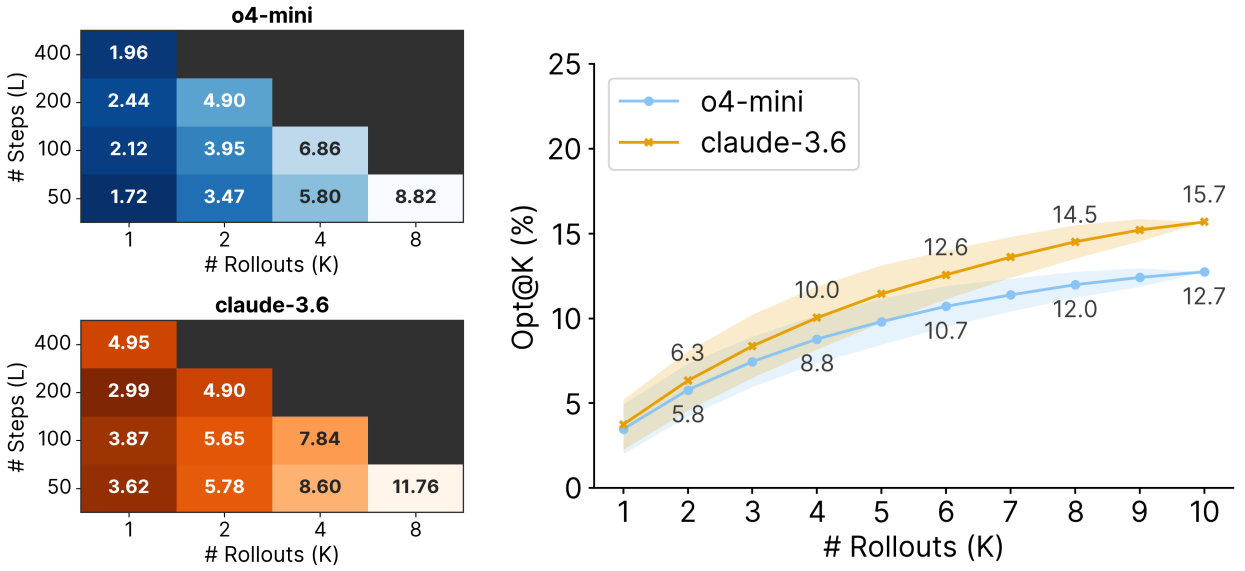


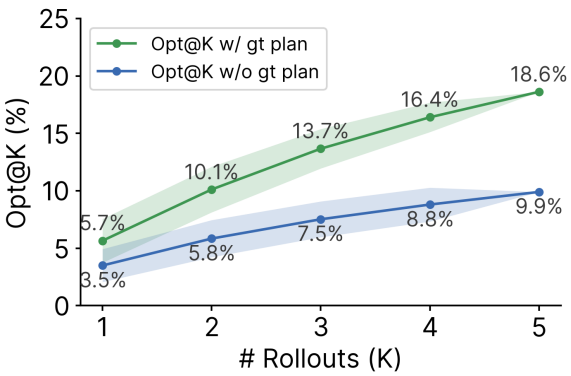
Figure 4.4: **Scaling test-time compute for O4-Mini and Claude-3.5-v2.** (a) Left: Opt@K performance as a function of inference steps (L) and parallel rollouts (K), showing parallel compute scales more efficiently than serial compute. (b) Right: Opt@K performance with increasing rollouts, improving to over 10% with diminishing returns beyond eight rollouts.

from 50 to 400 with different numbers of rollouts between 1 and 8. Results show parallel compute scales more efficiently than serial compute. With only 50 steps, 8 rollouts yields higher performance (8 82 for O4-Mini and 11 76 for Claude-3.5-v2) than 400 steps with a single rollout (1 96 for O4-Mini and 4 95 for Claude-3.5-v2). This indicates increased sample diversity across trajectories can effectively compensate for reduced step counts, providing insights for optimal inference-time compute allocation.

Low Opt@10 performance. Building on these findings, we further examine performance with extended parallel compute. Figure 4.4-right demonstrates that both models gain performance with additional rollouts, with Opt@K increasing from under 4% to over 12% with 8 rollouts. Despite these improvements, Opt@10 performance remains modest (under 20%) for both models with diminishing returns, indicating remaining difficulty for current SWE-Agents.

Performance with Ground-Truth Plans

Beyond engineering, solving GSO requires identifying bottlenecks and planning optimization strategies over a long horizon. Inspired by prior work on "backtranslation"-guided reasoning [112–115], we assess the impact of guided rea-



soning by prompting O4-Mini with descriptive backtranslated plans of ground-truth optimizations. We provide O4-Mini with the ground-truth diff and sample 5 plans describing the optimization strategy and specific file-localized changes. The supplementary material of Shetty et al. [110] details the prompt and example plans.

We observe that prompting agents with backtranslated plans improves performance, suggesting that high-level plans aid in matching human-level performance. However, Opt@1 only reaches 5 7% and Opt@5 improves by just 9% with these plans. So while strategic planning and reasoning helps, implementing low-level system changes remains challenging for current models.

4.7 Qualitative Analysis of Agent Behaviour

We use an LLM-aided pipeline (details in the supplementary material of Shetty et al. [110]) to qualitatively analyse agent behaviour and failure modes. We categorize the failures as (1) challenges with low-level code, (2) compute-management issues, and (3) localization errors.

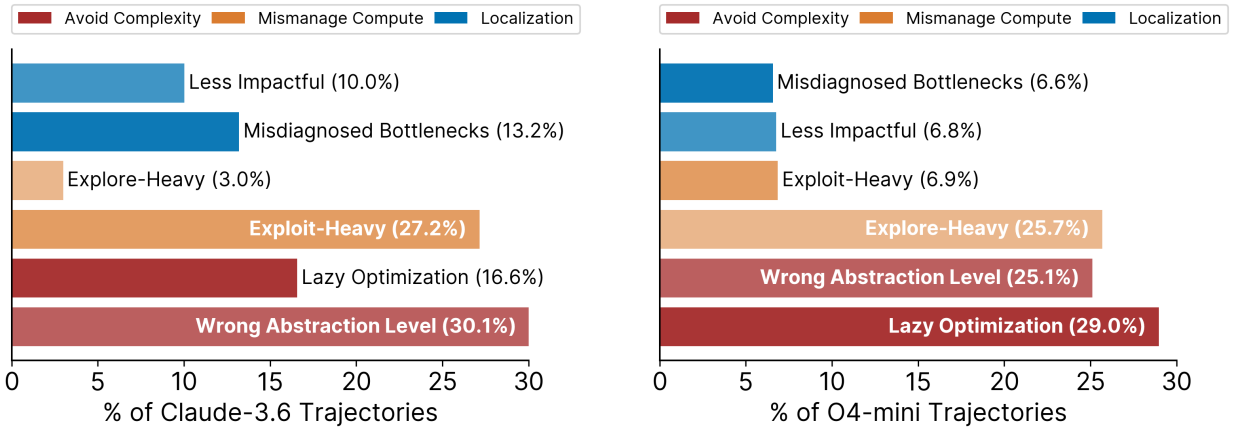


Figure 4.6: **Qualitative analysis of agents.** Model failures are classified into three high-level categories: (1) Localization: misidentifying code regions or opportunities for optimization, (2) Mismanage Compute: mismanaging exploration-exploitation trade-offs, and (3) Avoid Complexity: challenges with low-level code changes. **Left:** Claude-3.5-v2 shows an exploit-heavy behaviour, making massive code changes with less exploration of the codebase. It also attempts deeper changes but fails to localize bottlenecks and changes to the right abstraction level. **Right:** O4-Mini in contrast is explore-heavy, avoids low-level code, and makes superficial optimizations like spurious compiler-flag modifications.

Agents Struggle with Low-Level Code Changes

Poor performance on low-level problems. We identify sharp declines in agent performance as language complexity increases. Models perform best with high-level languages, with O4-Mini achieving 21% on Python tasks. Performance drops from 21% to 4% when `Cython`, `C`, and `C++`, etc. are involved.

Modifications at the wrong abstraction level. Production codebases have a hierarchy of abstraction levels, from high-level APIs to low-level implementations, with each layer encapsulating complexity beneath it. Our analysis reveals that operating at inappropriate abstraction levels contributes to 25 − 30% of agent failures. However, interestingly, models exhibit opposite but equally problematic approaches. Figure 4.7 shows that O4-Mini avoids making changes to the `C`/`C++` files 40% of the time even when it was necessary based on the human optimization commit. Claude-3.5-v2 on the other hand surprisingly makes unnecessary low-level `C` changes (9 2%) when even the human optimization commit was Python-only.

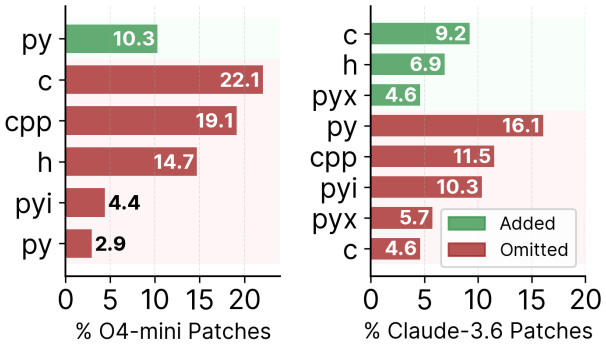


Figure 4.7: File extensions modified in model patches, indicating additions or omissions relative to the reference human commit.

In one case, O4-Mini attempted to optimize NumPy's `np subtract at` function. NumPy conceptually implements this in a layer below the Python API called `ufunc` (universal function) written in `C`. While the model scrolled through these `C` files, it decided to not make changes there and instead tried to override it with a Python function, completely avoiding the required deeper change.

Fundamental errors in low-level programming. Beyond selecting incorrect abstraction levels, agents also struggle with fundamental low-level programming concepts. In one case, Claude-3.5-v2 incorrectly modified Pillow's SIMD pointer arithmetic, causing segmentation faults.

Agents Favour Superficial Optimizations

Minimal-change optimization attempts. Agents consistently favour trivial code changes to meet performance targets rather than investigating and implementing more substantial improvements. O4-Mini exhibits this behaviour in nearly 30% of trajectories (Figure 4.6), with patch sizes significantly smaller than human-written optimizations. In fact, in over 60%

of incorrect trajectories, the agent made $\leq 15\%$ of the edits compared to the corresponding human developer commit (see supplementary material).

Spurious compiler-flag changes. In one case, CLAUDE-3.5-V2 attempted to optimize Pillow's SIMD implementation by simply adding `-O3` compiler flags. This approach is ineffective since the Pillow project already uses optimized builds by default. This pattern appears across many agent trajectories, suggesting that agents often fail to account for real-world project configurations.

Input-specific fast paths. Agents frequently implement narrow optimizations targeting only the specific input patterns present in a given performance test. In one case, O4-MINI created a specialized fast path for NumPy's `ljust` API that only handled "matching-shaped" input arrays. Our test suite identifies these narrow optimizations as failures due to their poor generalization properties.

Non-idiomatic overrides in `__init__.py`. A recurring pattern in O4-MINI trajectories is modifying `__init__ py` files to override functions instead of making core improvements. These overrides typically implement input-specific optimizations in a non-idiomatic manner, as shown below:

```
# __init__.py
_orig_strftime = _PeriodCls.strftime
def _fast_strftime(self, fmt):
    if fmt is None and getattr(self, "freqstr", None) == "M":
        return f"{y:04d}-{m:02d}" # Fast path for default monthly formatting
    return _orig_strftime(self, fmt)
```

See the supplementary material for further examples and analysis of this behavioural pattern.

Agents Mismanage Compute

Underutilize available compute. First, we find that agents often underutilize their available compute budget. We observe this quantitatively in our inference-time-scaling experiments (Section 4.6), where we increased the number of available agent steps. Even with larger budgets of 200+ steps, 75% of trajectories terminate before 100 steps. This again is consistent with the superficial-optimization behavior discussed earlier and highlights the need for better agent scaffolding and model improvements to optimally use compute.

Imbalance in exploration and exploitation. Figure 4.6 reveals a dichotomy in exploration-exploitation behaviours. O4-MINI trajectories are rated as explore-heavy, meaning they spend most of their steps examining the codebase without converging on actionable optimizations. On the other hand, CLAUDE-3.5-V2 trajectories are rated as exploit-heavy, meaning they commit to solutions with insufficient exploration of alternatives, and eagerly make many code changes. This also indicates a promising research direction: improving agent performance by leveraging the strengths of the two models.

Agents Misdiagnose Optimizations

Misidentify bottlenecks and solutions. Agents misdiagnose performance bottlenecks, implementing ineffective optimizations. In one case, CLAUDE-3.5-V2 attempted to parallelize NumPy's `char count` API, ignoring Python's GIL and process-startup overhead, resulting in worse performance. After multiple failures, the model concluded: "*For this specific use case, numpy's string operations are already highly optimized, stick with the original implementation.*"

Analyzing Model Successes

Section 4.6 shows that with increasing test-time compute, SWE-Agents can solve a small fraction of the tasks. Here, we analyze the characteristics of the tasks that SWE-Agents can solve. We find that agent solutions vary significantly in sophistication, ranging from simple but effective changes to genuinely impressive algorithmic improvements.

Some successful optimizations are less impressive when compared to what humans achieved on the same problems. In one case, O4-MINI added a fast path for writing data when network streams are idle, avoiding unnecessary buffering. But the human developer completely redesigned the entire buffering system with a much more sophisticated approach. In one case, CLAUDE-3.5-V2 optimized database-style lookups using bit-combining. The human solution was more comprehensive, upgrading the underlying search algorithms across the entire codebase. In one case, O4-MINI improved sorting by working directly with integer codes instead of string values. However, the human approach was cleaner, refactoring shared utilities that benefited multiple sorting operations.

However, agents can also implement sophisticated optimizations that outperform human solutions. O4-MINI completely rewrote image file parsing to read only essential metadata instead of decompressing entire frames, reducing algorithmic complexity from $O(n^2)$ to $O(n)$ (see supplementary material). The human developer only made a simple check, while the agent delivered an algorithmically stronger approach. CLAUDE-3.5-V2 eliminated memory waste by calculating exact allocation sizes upfront instead of repeatedly resizing arrays. The human solution still used dynamic resizing, just with better growth patterns, while the agent eliminated resizing entirely.

4.8 Related Work

Code LLM Benchmarks. Initial code generation benchmarks like HUMANEVAL [3, 24] and MBPP [4] focused on isolated small programs with simple specifications. These benchmarks have since evolved to evaluate LLMs across multiple languages (MULTIPL-E [47]), Data Science (DS-1000 [50], ARCADE [51]), API usage (JIGSAW [53], ODEX [54], BIGCODEBENCH [116]), and more complex algorithmic tasks in competitive programming (LIVECODEBENCH [30], APPS [117], CODE-CONTESTS [5], XCODEEVAL [61], CODESCOPE [118]). However, these benchmarks remain focused on isolated puzzle-solving tasks rather focusing on only code correctness and not performance optimization.

Performance Evaluation. Various works have introduced benchmarks to evaluate the performance capabilities of LLMs. EvalPerf [119] and EffiBench [120] assess runtime efficiency of code generated from natural language specifications on HumanEval and LeetCode tasks. In contrast, PIE [17], ECCO [121], and NoFunEval [29] focus on code optimization capabilities, where models improve existing programs while maintaining functional equivalence. These benchmarks employ different approaches to reliably measure program runtimes. PIE simulates hardware-level runtime for C++ programs while EvalPerf employs hardware counters for precise performance measurement. While providing reliability, these approaches unfortunately do not scale to larger codebases considered in our work. Other works [122, 123] utilize LeetCode's execution environment to evaluate LLM-generated code performance adding unwarranted dependence on external services. ECCO, similar to our approach, leverages cloud computing environments to ensure consistent benchmarking.

Repo-Level SWE-Agent Benchmarks. SWE-Bench [104] evaluates issue resolution in open-source repositories. Extensions include multi-modal capabilities [124] and multi-lingual capabilities [125, 126]. Specialized benchmarks address test generation [127, 128] and bug-localization [129]. Zhao et al. [130] proposed Commit-0 for library generation from scratch while R_2E [85] and RepoST [131] propose frameworks for function level code generation. These benchmarks however emphasize functional correctness rather than performance optimization considered in our work.

Recently, using LLMs to generate code has been receiving considerable attention, with hopes of automating AI research and development. Particularly, KernelBench [105] and METR-KernelEngineering [132] are two benchmarks that evaluate the performance of LLMs in generating performant code for kernel engineering. While they focus on a specific domain of kernel engineering, we explore software optimization capabilities of LLMs across domains.

Where Chapter 2 evaluates algorithmic correctness and Chapter 3 evaluates repository-level correctness, this chapter evaluates repository-level performance – completing Part I of the thesis by adding the quantitative dimension that binary specifications cannot measure.

4.9 Limitations

Benchmark Size. Our benchmark contains 102 software optimization tasks, which may introduce variance in results due to its limited size. Nevertheless, each task represents a challenging real-world optimization problem, making successful completion a strong indicator of model capabilities for high-performance software development. We will consider expanding the benchmark based on community feedback identifying additional representative tasks.

Hacky Optimizations. Reward hacking is a recurring issue in software agent benchmarks [133], with agents circumventing test cases in unintended ways [134]. As noted in Section 4.7, models already attempt to overfit tests and produce non-idiomatic code. Our

precise task specifications and test suite currently detect such issues, but monitoring these behaviors remains critical for future work and we recommend community efforts to develop mitigation approaches.

Evaluation Beyond Speedup. Our work focuses on improving runtime performance of the code but practical software development also requires other metrics such as memory usage, maintainability, and idiomaticity. For example, optimization often requires trade-offs between different metrics which are not captured by our speedup metric. Unfortunately, automated evaluation of these properties is an open problem, and we hope to tackle these challenges in future works.

Contamination. The current low performance reduces the likelihood that contamination materially affects these results, but does not rule it out despite our tasks being collected from GitHub repositories. Additionally, as discussed in Section 4.2, our continuous speedup metric helps detect contamination, as agent solutions that exceed human performance provide evidence against simple memorization.

4.10 Chapter Summary

We present GSO, a benchmark for evaluating LLMs in aiding the development of high-performance software. Our quantitative results demonstrate that current LLMs fall short in this domain and our qualitative analysis identifies various failure modes. We hope GSO can serve as a valuable resource for future works in this direction in building more capable SWE-Agents, including improvements to both the model and the agent scaffold.

This chapter closes Part I — evaluation — and the thesis now turns to training: how executable specifications power scalable data generation, supervised fine-tuning, and reinforcement learning for programming agents, starting with Chapter 5.

Part II

Training

Chapter 5

R2E-Gym: Procedural Synthetic Environments for Scaling SWE Agents

SWE agents perform below proprietary systems on SWE-Bench-Verified by roughly twenty points. The main bottleneck is the lack of executable training environments: each task in a usable gym requires a repository frozen at a buggy commit, a reproducible build, a fail-to-pass unit test that witnesses the bug, and a natural-language issue the agent can act on. Pan et al. [135] show exactly how tight that constraint is: 66 894 raw commits from the SWE-Bench training repositories yield only 2 438 executable tasks, because every surviving task demands a human-written issue and a human-written test linking the issue to its fix. GitHub contains many commits, but prior pipelines can directly use only those with suitable issue and test specifications. This chapter introduces SynGen, a framework that reduces this dependence on human-written specifications by using executable tests distilled from GitHub commits to ground an LLM's synthesis of new problem statements. This approach yields about 3 3× as many executable environments as SWE-Bench-Gym. When used for supervised fine-tuning, the resulting R2E-Agent agent achieves strong performance on SWE-Bench-Verified, demonstrating that synthetic issues paired with real execution environments provide an empirically effective source of supervised fine-tuning data for agent training.

5.1 Introduction

Autonomous SWE (software engineering), aiming to solve real-world software engineering problems such as GitHub issues, has made significant progress in recent times [124, 136]. While LLM-based SWE-Agents have demonstrated remarkable improvements, state-of-the-art performance is largely driven by proprietary models [137, 138]—with open models lagging behind [139].

Addressing this gap requires scalable curation of high-quality execution environments to train open-weight models. While several benchmarks for evaluating SWE-Agents on

GitHub issues exist [60, 130], scalable curation of high-quality training environments remains a challenging problem. For instance, while the training split from SWE-Bench [60] contains output patches, it lacks executable environments. Pan et al. [135] collect executable test environments, but rely on human-written issues and test cases, restricting sample size.

In this work, we introduce R2E-Gym, the largest procedurally curated environment for training real-world SWE-Agents—consisting of more than 8 1K problems, with executable gym environments, unit tests, and natural-language task descriptions (Section 5.2).

Synthetic Data Enables More Scalable Training. We propose SynGen—a novel synthetic data curation recipe that enables collection of a large number of executable training environments, reducing reliance on human-written pull requests (PRs) or unit tests. We show that instead of using human-written PRs, good-quality execution environments can directly be curated from *commits* through backtranslation [12, 140] and test collection or generation (Section 5.2). Compared to PR-based data collection [135], this approach enables more scalable data curation and agent training, resulting in a Pass@1 performance of 34 4% on the challenging SWE-Bench-Verified benchmark.

The key contributions of this chapter are:

1. We introduce R2E-Gym, the largest procedurally curated environment for training real-world SWE-Agents, increasing the number of executable environments by over 3× compared to human-curated baselines.
2. We release an open-weights 32B sft agent that achieves 34 4% Pass@1 on SWE-Bench-Verified—a +13 8-point lift over a matched-base-model SWE-Bench-Gym baseline, outperforming that baseline in the reported comparison.

5.2 SynGen: Procedural Synthetic Data Generation

SWE task collection methods [60] rely on human-written issues and unit tests for problem statements and evaluation functions. However, this presents a challenge for scaling data curation as size is limited by human-written PRs. To overcome this limitation, we propose SynGen—a synthetic data curation recipe using backtranslation and test generation. We procedurally generate environments using only commits from GitHub repositories, reducing reliance on both human-written issues and test cases.

Repository and Commit Curation

We use SEART GitHub Search[1] to identify Python repositories with a large number of commits. Next, we extract commit history and associated code changes for each repository. We filter relevant commits using a combination of rule-based and LLM-based heuristics, identifying *interesting* code changes. For each relevant commit, we next collect build scripts by semi-manually searching across dependency pins.

[1] https://seart-ghs.si.usi.ch/

Dataset (split)	Repo?	Executable?	# Instances
APPS [117]			10 000
R2E [85]			246
SWE-Bench-Train [60]			19 008
SWE-Gym Raw [135]			66 894
SWE-Bench-Test [60]			2 294
SWE-Gym [135]			2 438
R2E-Gym-Subset **(Ours)**			**4 578**
R2E-Gym **(Ours)**			**8 135**

Table 5.1: **Dataset statistics.** Comparing statistics across different datasets curating executable training environments for SWE-Agents training. R2E-Gym refers to our full dataset, and R2E-Gym-Subset refers to a filtered subset of tasks with non-overlapping repositories with SWE-Bench.

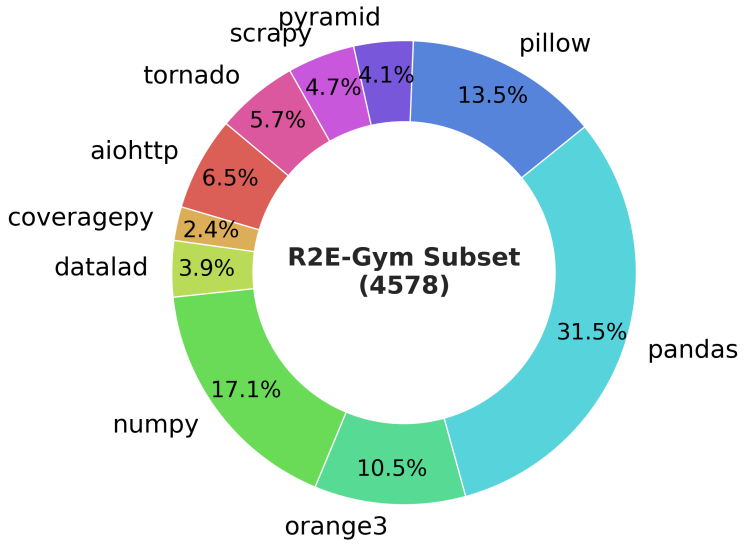


Figure 5.1: **Repo distribution** for R2E-Gym-Subset (no overlap with SWE-Bench) used for training (Section 5.3).

Executable Environment Construction

Installing historical commits from GitHub repositories presents significant challenges due to evolving dependency requirements and API changes. We use a Docker-based approach with a search-based dependency resolution strategy to create reproducible environments for each commit. Our installation process follows these steps:

1. Extract dependency information from `requirements.txt`, `setup.py`, etc.
2. Iteratively identify potential version conflicts and compatibility issues.
3. Generate multiple candidate dependency configurations.
4. Test each configuration until a working environment is found.

This process is semi-manual and challenging to scale, and we aim to rely more on LLMs in the future. This approach allows us to create working environments for historical commits, enabling execution-based validation of our dataset.

Test Validation and Generation

Following Jimenez et al. [60], we use the existing test cases in the curated commits to identify Fail Pass (F2P) test cases, i.e. test cases that fail in the original buggy commit and pass in the fixed commit. In cases where the curated commits do not have associated tests—limiting the ability to use them for training environments—we supplement such commits with automatically generated Fail Pass test cases. We use an Agentless-like reproduction test-generation approach; a key difference is that we use the ground-truth patch as context when generating the tests.

Backtranslation: Non-reliance on GitHub Issues

Using the above steps, we collect a large number of commits, associated build environments, and F2P (Fail Pass) test cases. Now, we need to collect the problem statements associated with the commits. Prior works [60, 135] use human-written GitHub issues as problem statements. This inevitably cannot use the entire commit history since human-written issues are not available for all commits. Here, following Wei et al. [12] and Li et al. [140], we propose a backtranslation approach to collect the problem statements associated with the commits.

However, direct backtranslation often produces generic problem statements that omit behavior-relevant details from the code changes. Instead, we identify that human-written issues often contain failing tests and execution traces as part of bug reports. We use this observation to collect high-quality problem statements by using the F2P test cases as part of the backtranslation prompt. Similar to existing works [85, 116], we find that using test-execution information allows generating precise and directed problem statements.

The issue-generation process follows these steps:

1. Extract failing test functions from the execution results.

2. Analyse test outputs to identify error messages and expected behaviours.
3. Provide the LLM with commit message, code patch, and test-execution results.
4. Guide the LLM to generate a concise, informative issue that describes the bug without revealing the solution.

For each commit, we extract and utilise specific components: commit metadata (hash and commit message); code patches, separated into non-test file changes (showing what was fixed) and test-file changes (showing how to verify the fix); test execution results from both the old (failing) and new (passing) commits; test functions that demonstrate the bug; and assertion failures extracted and formatted from the old commit to show error details.

We carefully design our prompting strategy to ensure the generated issues resemble human-written ones, focusing on clarity, naturalness, and providing sufficient information for understanding the bug. Figure 5.2 shows a representative output drawn from a `PIL` thumbnail-optimization commit.

We collect over 8 1K problem statements using this approach (referred to as R2E-Gym). We decontaminate this set by removing repositories overlapping with SWE-Bench test-set repositories, obtaining 4578 problems (referred to as R2E-Gym-Subset) and use that across all experiments unless specified otherwise. Notably, using our SynGen approach, we can collect over 2 5× more problems than relying on data collection from GitHub issues (Figure 5.3).

5.3 Training SWE-Agents using R2E-Gym Environments

After constructing the R2E-Gym environments, we turn them into training data for an open-weight SWE-Agent through a minimal scaffold and a standard SFT recipe.

Agent Scaffolding. We design a minimal scaffold on top of OpenHands [136] to experiment with agents for diverse SWE tasks. It uses a traditional ReAct framework [98] without any specialized workflow; equipping the LLM with only a bash terminal, file editor, and search tool. No internet or browser access is provided to the agent. Specifically, the policy has access to four tools—`file_editor` for viewing and editing files (supporting `create`, `view`, `str_replace`, `insert`, and `undo` operations); `search_tool` for locating terms within files or directories; `execute_bash` for running non-interactive bash commands; and `submit` (alias `finish`) to terminate the trajectory and emit the final patch.

Trajectory Collection and SFT Training. We next collect SFT trajectories using R2E-Gym environments. To avoid contamination, we only use a subset of R2E-Gym consisting of repos with no overlap with the SWE-Bench dataset. The resulting subset (R2E-Gym-Subset) consists of 4578 executable environments across 10 repositories (Figure 5.1). For each task environment, we use Claude-3.5-v2 with our agent scaffold and collect the successful agent trajectories. Through this process, we collect 3321 trajectories from 2048 unique task environments. We then use these trajectories to train our agent via supervised fine-tuning on agent thoughts and actions. For training, we use LLaMA-Factory [141] and Qwen-Coder models (7B, 14B, 32B) as our base models.

````
**Title:** Calling `load()` Before `draft()` Causes `draft()`
to Fail for JPEG Images

**Description:**
When generating a thumbnail for a JPEG image using the
`thumbnail()` method, the method calls `load()` before
`draft()`. This sequence results in the `draft()` method
returning `None`, which prevents the thumbnail from being
properly optimized.

**Example Code:**
```python
from PIL import Image

with Image.open("Tests/images/hopper.jpg") as im:
    im.thumbnail((64, 64))
```

**Expected Behavior:**
The `thumbnail()` method should utilize the `draft()`
method to optimize the image size before loading, ensuring
that the thumbnail is resized correctly and efficiently.

**Actual Behavior:**
The `draft()` method returns `None` because `load()` is
invoked before it. This prevents the thumbnail from being
optimized, potentially leading to incorrect thumbnail sizes
or unnecessary memory usage.
````

Figure 5.2: Example synthetic issue generated by execution-grounded back-translation from a `PIL` thumbnail-optimization commit.

5.4 Results and Analysis

Comparison to open-weight SWE-Agents across Model Scales. We report Pass@1 of R2E-Gym-trained models on the SWE-Bench-Verified and SWE-Bench-Lite benchmarks in Table 5.2. We also report comparisons with recently proposed SWE-Gym [135], which is closest to our work. As seen in Table 5.2, we find that our approach enables better scaling for training SWE-Agents across all model sizes. For instance, on SWE-Bench-Verified, for the same base-model type and scale, our 32B model improves Pass@1 by 13 8 percentage points, from 20 6% (SWE-Gym) to 34 4%.

Scaling with Number of Trajectories. We investigate the relationship between training sample size (number of trajectories) and agent performance in Figure 5.4. We evaluate 14B and 32B models trained with trajectory counts ranging from 100 to 3 200. Our findings indicate that performance improves with increasing trajectory count, though

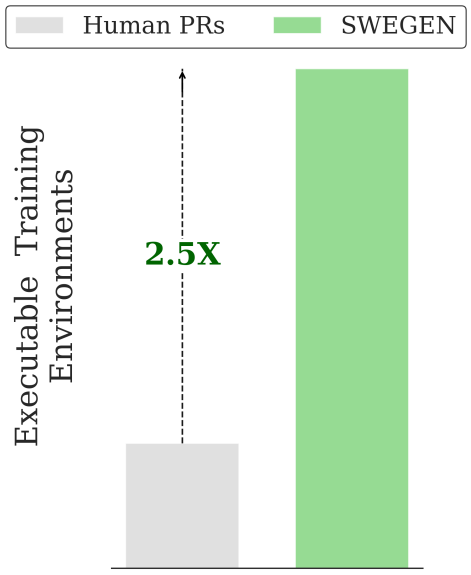


Figure 5.3: Executable environments per commit pool. Under the issue-and-test constraint of prior work, the same GitHub commit histories yield 2 438 tasks (SWE-Gym); under SynGen's synthetic-test and execution-grounded-issue construction, they yield 8 135—a $\sim$2 5$\times$ expansion.

Table 5.2: Resolve Rate (%) Comparison on SWE-Bench-Verified and SWE-Bench-Lite benchmarks. We observe that synthetic data curation (SynGen) allows our approach to scale better across different model sizes. All experiments use Qwen-Coder as base models.

Model	SWE-Bench-Lite				SWE-Bench-Verified			
Size	**Base-model**	**SWE-Gym**	**Ours**	**Δ**	**Base-model**	**SWE-Gym**	**Ours**	**Δ**
7B	1 0 (±1 0)	10 0 (±2 4)	**11 0** (±0 8)	+1.0	1 8 (±1 3)	10 6 (±2 1)	**19 0** (±1 0)	+8.4
14B	2 7 (±1 9)	12 7 (±2 3)	**20 67** (±0 7)	+7.97	4 0 (±1 6)	16 4 (±2 0)	**26 8** (±1 4)	+10.4
32B	3 0 (±1 4)	15 3 (±2 5)	**23 77** (±0 8)	+8.47	7 0 (±1 3)	20 6 (±2 1)	**34 4** (±1 2)	+13.8

with diminishing returns for both models. Notably, the 14B model begins to saturate at approximately 800 samples, while the 32B model still shows improvements, likely due to its larger capacity. These results extend the findings of Pan et al. [135], who studied dataset scaling up to $\sim$500 samples. Our analysis demonstrates that while performance does improve with increasing sample size, the rate of improvement diminishes or even stabilizes for smaller models.

Real vs Synthetic Problem Statements. The R2E-Gym approach enables us to generate problem statements without relying on human-written descriptions and test cases,

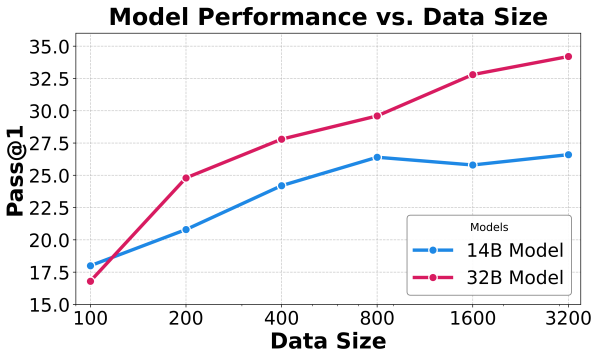


Figure 5.4: **Pass@1 scaling curve with increasing number of training samples.** Performance improvement with more training samples, enabled by SynGen.

Ablation	Config	Pass@1 (%)
Adding Thoughts	With	34 4
	Without	30 4
Real vs. Synthetic	Real	28 0
	Synthetic	27 8

Table 5.3: **Top.** Using thoughts in ReAct agent trajectories leads to significant performance improvements. **Bottom.** Using SynGen synthetic generated issues and test cases achieves similar performance as real-world issues (400 trajectories for both real & synthetic) while providing better scalability during data collection.

offering greater scalability. We compare the performance of models trained on real GitHub issues versus our synthetic problem statements (collecting 400 trajectories from both sets). Remarkably, models trained on synthetic data achieve nearly identical performance (27 8% Pass@1) to those trained on real data (28 0%). This finding shows that procedurally generated environments achieved similar performance to real-issue training in this 400-trajectory comparison while providing scalability.

Explicit Thought Traces are Important. During SFT we use both the agent's thought processes and actions as training targets. Models trained with thought demonstrations achieve better performance compared to those trained without (34 4% vs 30 4% in Table 5.3). This suggests that exposing the model to step-by-step reasoning processes is helpful for reliable problem-solving in complex environments.

5.5 Chapter Summary

R2E-Gym scales executable SWE training environments by synthesizing problem statements and tests procedurally, and fine-tuning Qwen-Coder-32B on R2E-Gym-Subset yields 34 4% Pass@1 on SWE-Bench-Verified—a +13 8-point lift over SWE-Bench-Gym at the same base-model scale, with the 32B model continuing to improve through 3 200 trajectories.

Several caveats qualify these results. The Docker-based dependency search is still partly manual; automating it end-to-end with an LLM-driven installer could allow larger-scale environment construction. The pipeline depends on Claude-3.5-v2 as trajectory teacher. Decontamination is repo-level: we exclude repositories that overlap with SWE-Bench.

The R2E-Gym infrastructure carries through the rest of the thesis: Chapter 7 uses the same F2P specifications as a sparse 0 1 RL reward, and Chapter 8 uses them as two

complementary verifier signals at inference time. The next chapter, however, changes the role of the specification—from training signal to behaviour-preservation constraint—and explores how LLMs can clean training data without sacrificing correctness; see Chapter 6.

Chapter 6

Code Quality: Tests as Behaviour-Preservation Constraints for Training-Data Cleaning

Large[1] language models trained on competition-programming corpora are optimized for functional correctness, while readability and structure are not directly emphasized. The training targets inherited from APPS and CODE-CONTESTS are contest artifacts (single-letter identifiers, flattened control flow, no helper functions, no comments), and a model that imitates these targets may reproduce the terse style common in contest solutions. This chapter asks whether program style and structure can affect downstream code-generation capability.

We operationalise this view in Language-Model Data Transformations (LMDT): an instruction-tuned LLM is prompted to rename variables, modularise scripts into named helpers, and prepend natural-language plans. Crucially, the executable specification acts as a *behaviour-preservation constraint*: after each rewrite, an oracle re-executes the dataset's tests and rejects any changes that break correctness. This ensures that the stylistic improvements do not alter the underlying program semantics.

6.1 Introduction

Natural language to code generation has witnessed considerable advances in recent years with the advent of large language models (LLMs). These advances primarily arise from training on large web-scale data and are measured based on the functional correctness of the programs. Thus, other aspects like readability, structuring, and styling and how they affect training and data quality are largely ignored.

[1]This chapter is based on joint work with Tianjun Zhang, Wei-Lin Chiang, Joseph E. Gonzalez, Koushik Sen, and Ion Stoica, published as *LLM-Assisted Code Cleaning for Training Accurate Code Generators* at ICLR 2024 [64].

In this chapter, we show that using programs following good programming practices and allowing for more readability leads to improved code generation performance compared to using programs that do not follow these practices. We use these insights to build a novel automated code data-cleaning pipeline that transforms programs while maintaining functional correctness using input-output examples. In contrast to prior works that curate high-quality datasets by directly generating new data using LLMs, here we translate existing datasets into their parallel cleaned versions while identifying attributes that actually improve data quality.

We use LLMs to perform the transformations used in our data-cleaning approach. We demonstrate that instruction-tuned models can take a user-identified attribute of data quality as a natural language instruction and perform the transformation accurately. Our approach leverages the disparity in difficulty between generating a solution and editing an existing one. We perform our data-cleaning transformations in three iterations: 1) renaming variables, 2) modularizing complex code into subfunctions, and 3) adding planning annotations.

Figure 6.1 provides an overview of our approach. Notice that the variable renaming step at the top adjusts the variable names to be contextually relevant (e.g. `a` to `root_u` and `d` to `graph`). The modularization step (depicted on the right) identifies and decomposes the original program into several smaller subfunctions such as `find_root`, `merge_trees`, `build_graph`, etc. It then implements these subroutines and assembles the modular program. Finally, our planning step (depicted at the bottom) constructs a plan by summarizing functions in a top-down fashion (starting from the `main`).

We evaluate our approach in the domain of algorithmic code generation using two well-known benchmarks, APPS [117] and Code-Contests [5]. We transform the corresponding programs in the training sets and obtain parallel datasets from our cleaning approach, utilizing input-output examples to maintain functional equivalence.

We fine-tune the CodeLlama-7B model on the various collected datasets. Our findings reveal that the model fine-tuned on our modularized dataset outperforms the model fine-tuned on the functionally equivalent original dataset by up to 30% relative on APPS-Introductory. Furthermore, in comparison to existing baselines, our fine-tuned models outperform the larger AlphaCode [5] models on Code-Contests.

6.2 The LMDT Framework and Code Transformations

In this section, we present our general data transformation approach and then instantiate it for performing code data cleaning.

Transformations for Data Cleaning

Given a dataset $\mathcal{D}$ consisting of N instances d_i, to achieve a desired data cleaning specification, the user provides a data-cleaning instruction $\mathcal{I}$, which highlights an attribute that needs to

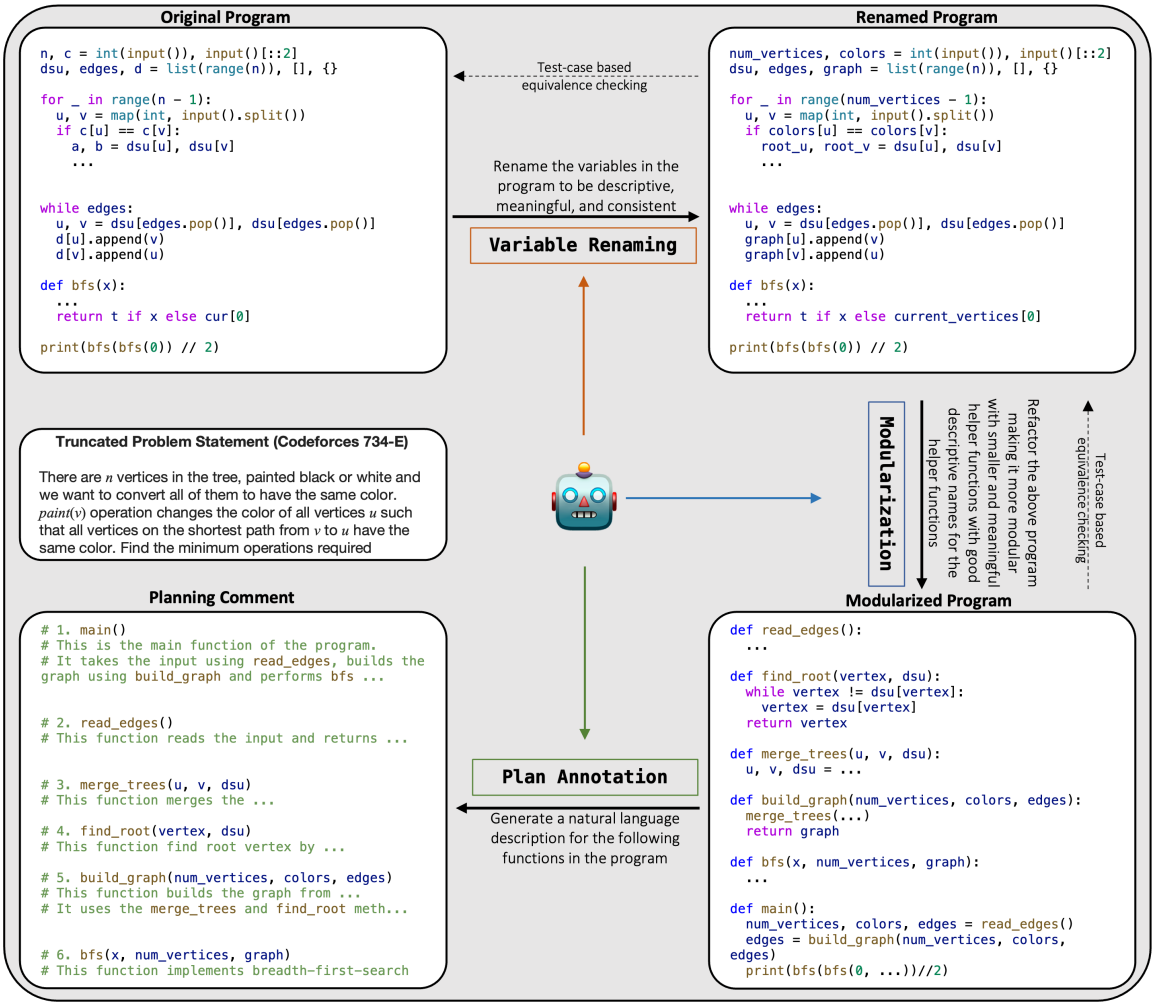


Figure 6.1: The three-way transformation of a single CODE-CONTESTS program. The top-left panel shows the original scraped solution. Renaming rewrites cryptic identifiers (top-right); modularization extracts helper functions and a `main()` entry point (bottom-right); and the planning step prepends a top-down natural-language summary as a comment (bottom-left). The middle-left panel is the truncated problem statement.

be modified. We also use an oracle equivalence checker $\mathcal{O}$ which ensures that the transformed data instance d_i is consistent with the original input based on some desired metric.

We use a pre-trained language model $\mathcal{M}$ to generate the transformed instance d_i by prompting the model with the transformation instruction $\mathcal{I}$ and the original answer d_i. Finally, we apply our oracle equivalence checker $\mathcal{O}$ to ensure consistency with the original data. If $\mathcal{O}(d_i \; d_i) = 0$, i.e., the oracle reports a failure, we reject the generated output and retry the example within a sampling budget.

While our transformation approach does not provide formal guarantees about the quality

Dataset	Notation	Applied on	Transformation instruction ($\mathcal{I}$)
Base	$\mathcal{D}_{\text{original}}$	—	—
Rename	$\mathcal{D}_{\text{rename}}$	$\mathcal{D}_{\text{original}}$	*Rename the variables in the program to be descriptive, meaningful, and consistent.*
Modularize	$\mathcal{D}_{\text{modular}}$	$\mathcal{D}_{\text{rename}}$	*Refactor the above program making it more modular with smaller meaningful helper functions with good descriptive names.*
Plan	$\mathcal{D}_{\text{planning}}$	$\mathcal{D}_{\text{modular}}$	*Generate a natural language description for the following functions in the program.*

Table 6.1: The three code transformations composing our cleaning pipeline. Each step applies to the previous step's output under a single natural-language instruction; the oracle $\mathcal{O}$ — the dataset's own input–output tests — is held fixed across steps.

of the performed transformation, we empirically observe that instruction-tuned LLMs can perform various unstructured data cleaning steps quite effectively once the oracle is available to reject failures.

Three Code Transformations

We apply our transformations-based data cleaning approach to programming data. Coding requires both low-level programming and high-level reasoning or planning skills. Therefore, we propose a three-step cleaning pipeline that improves the readability and program structuring targeting the low-level coding skills and inserts natural-language based plans data targeting the high-level reasoning skills. Our steps are detailed below.

1. **Rename variables** ($\mathcal{D}_{\text{rename}}$, applied on $\mathcal{D}_{\text{original}}$). This step renames the variables in the program, making them descriptive and easier to follow.

2. **Modularize functions** ($\mathcal{D}_{\text{modular}}$, applied on $\mathcal{D}_{\text{rename}}$). Problem decomposition has been identified as a key approach for improving the reasoning capabilities of models. We identify program decompositions and transform the program by extracting their functionality into smaller helper functions.

3. **Plan annotations** ($\mathcal{D}_{\text{planning}}$, applied on $\mathcal{D}_{\text{modular}}$). This step summarizes the helper functions in the already modularized program and prepends it to the programs in the form of a natural language plan. These natural language descriptions are analogous to prompting approaches that are used for solving reasoning problems like chain-of-thought prompting.

Additionally, while performing these transformations, we use the test cases provided in the dataset to construct our oracle equivalence checker $\mathcal{O}$. It ensures that our transformed programs maintain functional equivalence to the original program.

6.3 Experimental Setup

In this section, we detail our experimental setup and implementation, outlining the benchmarks, metrics, and fine-tuning details.

Benchmarks and Data Transformations

We use two standard algorithmic code generation benchmarks, APPS [117] and CODE-CONTESTS [5]. The benchmarks provide a collection of problem statements described in natural language and corresponding test cases.

For APPS, we only consider problems sourced from a subset of the competition websites based on the number of test cases provided. For CODE-CONTESTS, we only use a subset of the training split that includes `python` solutions satisfying the provided test cases. Additionally, since the training set provides over a hundred solutions per problem, we perform LSH-based near-deduplication on the solutions and limit them to a maximum of 25 solutions per problem. Table 6.2 provides further details about our final datasets.

Subset	Split	# Problems	Median Tests	# Solutions
APPS-INTRODUCTORY	train	42	1	736
	test	702	10	–
APPS-INTERVIEW	train	1,247	1	18,394
	test	2,699	19	–
APPS-COMPETITION	train	361	9	5,060
	test	309	39	–
CODE-CONTESTS	train	7,132	200	98,582
	test	165	200	–

Table 6.2: Problem counts, median number of public tests per problem, and number of reference solutions retained for training across the four subsets after curation.

We apply our data transformation approach on the APPS and CODE-CONTESTS datasets. Unless specified otherwise, we use GPT-3.5-TURBO as our default language model to perform the transformations with a default temperature of 0 3. In case of failure, we retry up to 5 iterations. We obtain three parallel datasets at the end of our cleaning process: $\mathcal{D}_{\text{rename}}$, $\mathcal{D}_{\text{modular}}$, and $\mathcal{D}_{\text{planning}}$.

We also simulate a simple direct synthetic data generation approach. Specifically, we generate solutions for the training problems using the GPT-3.5-TURBO model with two-shot

prompt examples selected from our $\mathcal{D}_{\text{modular}}$ dataset. We filter the solutions for correctness based on the ground truth test cases to ensure we are not training on incorrect programs. Since it resembles a distillation-like setup, we refer to this dataset as $\mathcal{D}_{\text{distill}}$.

Metrics and Fine-Tuning Details

We assess the code generation performance of the models using the PASS@K metric, which evaluates the functional correctness of generated programs. We vary K in 1 10 25 for the APPS dataset and 1 10 100 for the CODE-CONTESTS benchmark.

We evaluate the quality of the transformed datasets by measuring how they impact the test benchmark accuracy. We study both in-context learning and fine-tuning. We use the CODELLAMA-7B model [142] in all our experiments and perform batched inference through vLLM [36].

For in-context learning, we select two question-answer pairs from the $\mathcal{D}_{\text{original}}$ and $\mathcal{D}_{\text{modular}}$ training sets as our in-context learning examples. For a fair comparison, we use the same problem and corresponding solutions from the two datasets as examples.

For fine-tuning, we perform full fine-tuning over the base CODELLAMA-7B model on the different datasets. We train the models for two epochs on the APPS dataset and one epoch on the CODE-CONTESTS dataset using a learning rate of 5e−5 and an effective batch size of 256 assembled via gradient accumulation on four A6000 GPUs.

6.4 Main Results

We evaluate each cleaned dataset by comparing CODELLAMA-7B trained on $\mathcal{D}_{\text{original}}$ against the same model trained on our refactored datasets, using PASS@K with the sampling budgets defined in Section 6.3. Modularization gives the largest gains at PASS@1 and improves several higher-budget metrics, with relative gains varying by benchmark and metric, while renaming alone and planning-annotation fine-tuning contribute smaller gains.

APPS

Table 6.3 reports PASS@K on APPS-INTRODUCTORY and APPS-INTERVIEW for in-context learning (ICL) and fine-tuning, across all of our cleaned variants.

Under ICL, swapping the two-shot exemplars from $\mathcal{D}_{\text{original}}$ to $\mathcal{D}_{\text{modular}}$ moves PASS@1 on APPS-INTRODUCTORY from 14 2 to 17 5, a 23% relative gain. This indicates that more readable, better-structured coding is helpful to the model in solving more problems.

Fine-tuning shows a larger version of the same trend. On APPS-INTRODUCTORY, training on $\mathcal{D}_{\text{modular}}$ improves PASS@1 from 18 7 to 22 7, a 21% relative increase over $\mathcal{D}_{\text{original}}$. PASS@10 rises from 34 4 36 9, and PASS@25 moves from 40 2 to 42 6. $\mathcal{D}_{\text{planning}}$ does slightly better on the higher-budget metrics (PASS@25 43 8 vs 42 6) but underperforms $\mathcal{D}_{\text{modular}}$ at PASS@1. $\mathcal{D}_{\text{rename}}$ improves over the original dataset but remains below $\mathcal{D}_{\text{modular}}$, highlighting that

	APPS-Introductory			APPS-Interview		
Setting	Pass@1	Pass@10	Pass@25	Pass@1	Pass@10	Pass@25
In-context learning						
CodeLlama-7B + $\mathcal{D}_{\text{original}}$	14 2	29 2	38 4	1 8	7 3	10 4
CodeLlama-7B + $\mathcal{D}_{\text{modular}}$	**17 5**	30 1	39 7	2 2	8 6	12 3
Fine-tuning						
CodeLlama-7B + $\mathcal{D}_{\text{original}}$	18 7	34 4	40 2	3 4	9 7	13 6
CodeLlama-7B + $\mathcal{D}_{\text{modular}}$	**22 7**	**36 9**	42 6	**4 2**	**11 0**	**15 0**
CodeLlama-7B + $\mathcal{D}_{\text{planning}}$	22 1	**37 1**	**43 8**	3 7	10 5	14 8
CodeLlama-7B + $\mathcal{D}_{\text{rename}}$	19 2	36 6	42 9	4 0	10 7	14 6
CodeLlama-7B + $\mathcal{D}_{\text{distill}}$	21 1	35 3	40 5	4 1	10 8	14 5
Closed-source reference						
Code-Davinci-002	22 1	50 2	58 7	4 1	16 8	23 8

Table 6.3: **Main results on APPS.** We fine-tune CodeLlama-7B on each cleaned dataset and additionally evaluate two-shot in-context learning with parallel exemplars from $\mathcal{D}_{\text{original}}$ and $\mathcal{D}_{\text{modular}}$. Our cleaned datasets yield over 20% relative improvement in Pass@1 on APPS-Introductory. Code-Davinci-002 numbers are from Chen et al. [70].

functional decomposition, not just readability, is a key aspect of improving performance. $\mathcal{D}_{\text{distill}}$ performs between $\mathcal{D}_{\text{rename}}$ and $\mathcal{D}_{\text{modular}}$, suggesting an advantage for cleaning over the generation baseline.

At Pass@1, CodeLlama-7B fine-tuned on $\mathcal{D}_{\text{modular}}$ reaches 22 7, matching Code-Davinci-002 (22 1) on this split. The same pattern holds on APPS-Interview, where fine-tuning on $\mathcal{D}_{\text{modular}}$ improves Pass@1 from 3 4 to 4 2 (+23% relative), making modularization the strongest recipe at Pass@1 in these APPS comparisons.

Code-Contests

Code-Contests is a harder corpus, so absolute pass rates are lower and we evaluate at Pass@10, Pass@25, and Pass@100 following prior work [5]. Table 6.4 reports the main numbers.

Fine-tuning on $\mathcal{D}_{\text{modular}}$ improves Pass@10 from 5 0 to 6 1 (+22% relative), Pass@25 from 6 4 to 8 3 (+30% relative), and Pass@100 from 10 9 to 12 4 (+14% relative). The two-shot ICL variant sees Pass@100 jump from 7 2 to 9 3 (+29% relative). $\mathcal{D}_{\text{planning}}$ and $\mathcal{D}_{\text{rename}}$ again trail $\mathcal{D}_{\text{modular}}$, suggesting that renaming and planning in isolation are not enough once the algorithmic difficulty rises.

CodeLlama-7B trained on $\mathcal{D}_{\text{modular}}$ beats both AlphaCode-9B and AlphaCode-41B at Pass@10, Pass@25, and Pass@100. It also beats Code-Davinci-002 at Pass@10 and

Setting	Pass@10	Pass@25	Pass@100
In-context learning			
CodeLlama-7B + $\mathcal{D}_{\text{original}}$	5 1	6 5	7 2
CodeLlama-7B + $\mathcal{D}_{\text{modular}}$	4 9	6 6	**9 3**
Fine-tuning			
CodeLlama-7B + $\mathcal{D}_{\text{original}}$	5 0	6 4	10 9
CodeLlama-7B + $\mathcal{D}_{\text{modular}}$	**6 1**	**8 3**	**12 4**
CodeLlama-7B + $\mathcal{D}_{\text{planning}}$	5 3	7 0	10 8
CodeLlama-7B + $\mathcal{D}_{\text{rename}}$	4 7	6 3	10 5
Closed or larger models			
AlphaCode-9B [5]	5 0	7 0	10 0
AlphaCode-41B [5]	5 0	7 0	10 0
Code-Davinci-002 [62]	3 0	–	7 5
GPT-3.5-turbo [143]	–	–	18 2
+ Brainstorm [143]	–	–	29 3

Table 6.4: **Main results on Code-Contests.** CodeLlama-7B fine-tuned on $\mathcal{D}_{\text{modular}}$ outperforms AlphaCode-9B and AlphaCode-41B on all Pass@K.

Pass@100 on this benchmark. It remains behind GPT-3.5-turbo (12 4 vs 18 2 at Pass@100), which is consistent with the scale differences.

6.5 Ablations and Discussion

We decompose the main results into two key ablations that isolate where the quality gain comes from.

Modularization dominates renaming. Training on $\mathcal{D}_{\text{rename}}$ alone already beats training on $\mathcal{D}_{\text{original}}$—improving Pass@1 from 17 2 to 19 1 on APPS-Introductory—but the gain is roughly a third of what full modularization delivers (22 7). Variable renaming alone leaves the surrounding control flow unchanged. Functional decomposition is therefore the strongest factor among the cleaning steps tested here.

Cleaning transformations vs distillation. $\mathcal{D}_{\text{modular}}$ and $\mathcal{D}_{\text{distill}}$ share the same GPT-3.5-turbo teacher, so their comparison cleanly isolates rewriting versus generating from scratch. Fine-tuning on $\mathcal{D}_{\text{modular}}$ dominates $\mathcal{D}_{\text{distill}}$ on APPS across all Pass@K. The gap is foreshadowed at data-collection time: refactoring accepts 94% of inputs under the oracle gate, whereas distillation produces a correct solution for roughly half of the problems. In this

setup, editing existing correct solutions yields a much higher oracle-accepted rate than direct generation, so cleaning yields both more data and more faithful data than distillation.

6.6 Related Work

Data quality for LLMs. A recent line of work argues that small, carefully curated instruction corpora rival much larger noisy ones [144–146]. A complementary thread builds quality into pretraining by generating "textbook-grade" synthetic code from strong teachers [147, 148]. LMDT sits between these two stances: we neither generate fresh training pairs nor filter to a smaller subset, but transform in place and gate each transform with the dataset's own tests.

Synthetic data from LLMs. Distilling from a stronger teacher has driven much of the recent instruction-tuning literature [39, 149–151] and has been specialised to reasoning [152, 153], summarisation [154], tool use [20], and mathematics [155, 156]. LMDT shares the oracle-verified spirit but targets transformations of existing, behaviour-correct solutions rather than free-form generation, which has a higher acceptance rate than the distillation baseline (94%).

Algorithmic code generation. We adopt the algorithmic-programming evaluation anchored by APPS [117] and Code-Contests [5], with AlphaCode [5] as the canonical open baseline. Around these benchmarks, a family of inference-time techniques has grown: lookahead-search decoding [157], test-based re-ranking [62, 70], and planning DSLs [94]. The closest in spirit is CodeChain [158], which promotes modular decomposition through iterative prompting at inference; LMDT instead introduces the same decomposition bias through the training data.

Broader connections. LMDT also touches repository-level retrieval [57, 58, 97], data-science and API benchmarks [20, 159–161], reinforcement learning with execution feedback [101, 162, 163], self-repair [32, 33], and prompt optimisation [164].

6.7 Chapter Summary

Traditionally, data quality has been linked to functional correctness, ignoring the rich stylistic aspects differing across programs. In this chapter, we demonstrated that aspects like readability and program structuring impact the performance of the trained model on downstream tasks and thus also contribute to data quality.

We proposed LMDT, a novel data-cleaning pipeline demonstrating that LLMs can be used to transform existing datasets to improve their quality based on user instructions and an oracle

equivalence checker. Treating every cleaning step—rename, modularize, plan—as a behaviour-preserving refactor gated by the dataset's own I/O tests checks functional equivalence on the available I/O tests, though it does not guarantee semantic equivalence beyond those tests. On Code-Contests, this check accepts 94% of the corpus. Training CodeLlama-7B on the cleaned data improves Pass@1 by roughly 30% relative on APPS-Introductory and matches or exceeds the original-dataset baseline with about 15% of the cleaned examples.

In the next chapter, we return to the main argument and show how the executable specifications synthesized in R2E-Gym (Chapter 5) can be used directly as a sparse RL reward, training an open-weight SWE agent end-to-end without any supervised-fine-tuning warm-start.

Chapter 7

DeepSWE: Tests as a Sparse RL Reward for End-to-End Agent Training

This chapter is based on *DeepSWE: Training a Fully Open-sourced, State-of-the-Art Coding Agent by Scaling RL*, a Together AI blog post published in July 2025 [165]. The work was a joint Agentica–Together AI collaboration; the author's contributions, with Jaskirat Singh, centered on the R2E-Gym training environment, agent scaffold, SWE-agent SFT data, verifier training, and the hybrid test-time scaling used again in Chapter 8.

This chapter investigates whether a SWE agent can be trained end-to-end from a base LLM using only reinforcement learning on realistic GitHub issues. Prior open-weights SWE agents at this scale (R2EGym-Agent, Skywork-SWE, OpenHands-LM, SWE-Agent-LM, Devstral-Small) require supervised fine-tuning from a stronger closed teacher before applying RL. DeepSWE asks whether a capable base model (Qwen3-32B) can acquire the full SWE-agent loop (localize, edit, run tests, iterate, submit) directly from a verifiable terminal reward, without being shown a single trajectory from a frontier model. In this setup, the executable specification collapses to its simplest form: the Pass2Pass and Fail2Pass test suite becomes a scalar 0 1 reward. This identical test suite is the *only* signal the agent sees throughout RL (no per-step rewards, no teacher, no demonstrations), making the specification the actual training objective. Starting from Qwen3-32B and 4 500 R2E-Gym tasks, DeepSWE-Preview reaches 42 2% Pass@1 averaged over sixteen runs and 71 0% Pass@16 on SWE-Bench-Verified. With hybrid test-time scaling (Chapter 8) it reaches 59 0% Hybrid-Best@16, setting a new open-weights state of the art at release.

7.1 Training Setup

Reinforcement learning treats an agent as an autonomous entity that issues actions against an environment and receives feedback in the form of new observations and a scalar reward. The environments span Atari games, continuous robotic control, database management, protein design, and, of interest here, software development inside real codebases. An LLM-based

agent is a specific instantiation of that abstraction: its policy conditions on the running trajectory of prior observations and actions, and each action is a function or tool call whose side-effects produce the next observation. The cumulative rollout is a ChatML-style trace of assistant messages (thought + tool call) interleaved with tool outputs, terminated by an environment signal or by the agent's own `finish` call [165].

Autonomous SWE instantiates this abstraction with a programming environment: the observation space is a terminal plus a project filesystem, and a single pull request is one RL task. The agent is handed a natural-language issue and a codebase frozen at a specific commit, and must navigate, edit, build, and test until it believes the issue is resolved. Its action space mirrors the tools a human developer uses in an IDE: bash execution over a shell, a search primitive over a directory or file, and a file-viewer/editor that supports view, create, string-replace, insert, and undo; an optional `finish`/`submit` tool ends the trajectory [166]. The reward is computed by running the project's automated test suite on the modified code; the outcome reward is 1 when the submitted patch passes the selected `Pass2Pass` and `Fail2Pass` tests within a time budget and 0 otherwise—a direct scalarization of the executable specification that frames this thesis.

Pure RL over this environment is the interesting claim. Prior open-weights SWE agents—R2EGym-Agent, Skywork-SWE, OpenHands-LM, SWE-Agent-LM, and Devstral-Small—all warm-start from supervised fine-tuning on trajectories distilled from proprietary teachers, typically Sonnet-3 5/3 7/4, and some add RL on top [124, 135, 136, 139, 166]. DeepSWE removes the SFT stage entirely, starting from Qwen3-32B [167] and training end-to-end with reward alone. This echoes the `DeepSeek-R1-Zero` hypothesis that a sufficiently capable base model can acquire a new domain's skill directly from a verifier [168], and doubles as a reproducibility argument: frontier-agent capability is decoupled from access to any single closed model.

Scaling RL from single-step reasoning [168, 169] to long-horizon multi-turn agents exposes three concrete obstacles. First, trajectory length and multi-turn credit assignment: each rollout can span dozens of tool calls, so the single-step GRPO objective must be extended to mask environment observations and user messages from the policy loss, following RAGEN, Verl [103], ROLL, ART, and Sky-RL. Second, reward sparsity and collapse: a 0 1 terminal reward distributed over a $\sim$40-step trajectory is a famously sparse gradient signal, and—more insidiously—an agent can stumble into passing tests by accident, reinforcing noisy post-fix thrashing; the blog identifies this as the primary driver of reward collapse during training. Third, environment-infrastructure scaling: spinning up hundreds of real Dockerized build environments per RL iteration is a distributed-systems problem rather than a learning problem, and the one that ultimately caps throughput. These three challenges shape the training setup in Section 7.1 and motivate the algorithmic choices in Section 7.2.

Environment

The training environment is a thin wrapper around the R2E-Gym gym (Chapter 5): a pool of executable SWE tasks, a four-tool action space, and a sparse outcome reward. We keep

every component intentionally minimal — the goal is to put the policy in direct contact with real repositories and real tests, and let the reward do the shaping.

Dataset. We draw 4 500 training tasks from a subset of R2E-Gym, released as `R2E-Gym-Subset`. To prevent training-time contamination, we filter out every task whose source repository overlaps with SWE-Bench-Verified — notably `sympy` — so that the training and evaluation pools are repository-disjoint by construction. Each task maps to its own Docker image, carrying the pinned toolchain, project dependencies, and both the Fail-to-Pass (F2P) and Pass-to-Pass (P2P) test selections used to score a submitted patch.

Action space. The policy sees the same four tools used by the R2E-Agent scaffold in Section 5.3, carried over unchanged into RL:

- `execute_bash` — run an LLM-generated bash command and return its `stdout` and `stderr`.
- `search` — return every occurrence of an LLM-supplied query within a named directory or file.
- `file_editor` — view, create, `str_replace`, `insert`, or `undo` edits against a specific file.
- `finish`/`submit` — signal that the policy believes the pull request is resolved, terminating trajectory generation.

The state returned to the policy after each action is the tool's Python `stdout`/`stderr` stream; there is no parsed observation or structured environment state. Reusing the Chapter 5 scaffold unchanged keeps the interface fixed between SFT and RL, so the chapter can focus on the training signal rather than a changed API.

Reward. The reward function is a sparse Outcome Reward Model (ORM). For a submitted patch,

$$r \;=\; [\,\textsc{all}\,(\text{F2P} \quad \text{P2P})\ \text{tests pass within 5 minutes}]$$

so r 0 1 : a trajectory receives reward only if the final patch passes every selected test within a five-minute wall-clock budget. The cap is the one substantive difference from the official SWE-Bench-Verified evaluator, which allows thirty minutes; the shorter window trades a handful of spuriously-timed-out rollouts for a roughly 6× reduction in per-trajectory environment cost, which is what makes 512-wide rollout batches (Section 7.1) affordable. The reward is the executable-specification object at its most compressed: a single scalar that is the Boolean AND over all F2P and P2P tests passing within the time budget, with no partial credit, no shaping, and no intermediate supervision.

Policy and rollout configuration. We initialize from QWEN3-32B with thinking mode enabled; the non-thinking variant fails to improve under the same recipe (Section 7.4). Each RL iteration collects 8 trajectories per prompt over a 64-prompt batch, for 512 live containers per iteration. Trajectories are evaluated at 64K context and capped at 100 environment steps — sized so that the long-horizon behaviours we care about (reproduction, patch, regression-test check) fit inside a single QWEN3-32B forward window. The full training campaign runs for six days on 64 H100 GPUs, with environment orchestration handled by the Kubernetes layer described next.

Kubernetes-Scale Environment Orchestration

At final-run scale, R2E-GYM is also a distributed-systems artifact. Each GRPO++ iteration launched 512 Docker containers in parallel — batch size 64 times 8 rollouts per prompt — and concurrent experiments pushed thousands of containers into flight. A single-host `dockerd` became the bottleneck: the Docker API server overloaded and the daemon eventually crashed. The practical requirement was therefore not only more GPU compute, but a rollout layer that could schedule and isolate hundreds of real repository environments per iteration.

The fix was a Kubernetes wrapper around R2E-GYM. Instead of sending every rollout to one Docker daemon, the orchestrator schedules pods across worker nodes, each with roughly 200 CPU cores and more than 6 TB of local NVMe SSD. The RLLM trainer still sees the same environment API; the implementation detail changes from a local container id to a scheduled pod. This keeps the learning loop separated from the placement and failure-management logic of the environment fleet.

Two engineering details make the wrapper usable at this scale. First, SWE-BENCH-VERIFIED and R2E-GYM images are preloaded on each node's local disk, avoiding repeated Docker Hub pulls in the rollout hot path. Second, the Kubernetes Cluster Autoscaler provisions nodes when pods remain unschedulable and reclaims nodes that stay under-utilized for roughly twenty minutes. Together, image locality and elastic scheduling let the cluster scale past 1 000 CPU cores while keeping the sparse-reward RL loop limited by agent work rather than container startup.

7.2 GRPO++ and Compact Filtering

Extending GRPO from single-step reasoning to multi-turn SWE agents requires masking the policy loss to assistant-generated tokens: user messages and environment observations in the trajectory are context, not actions to reinforce. GRPO++ starts from this masked multi-turn objective and adds the stability modifications in Table 7.1.

GRPO++ combines ideas from DAPO [170], DR.GRPO [169], and LOOP/RLOO [171], plus two modifications from DEEPSWE: compact filtering and removing the entropy bonus. We use the component list, rather than an inferred equation, as the algorithmic record for

this chapter because the public release does not include a full formal loss or hyperparameter table.

Component	Source	Purpose
Clip High	DAPO	Raise the upper bound of the PPO/GRPO surrogate-loss clip; encourage exploration and stabilize token-level entropy.
No KL Loss	DAPO	Remove the KL(π π_{ref}) term so the policy can leave the base model's trust region — essential when training from scratch without SFT.
No Reward Std	Dr.GRPO	Drop per-group reward-std normalization to eliminate the difficulty bias that up-weights gradients on easy vs. hard problems.
Length Norm.	Dr.GRPO	Divide the surrogate loss by the max context length rather than per-sample length; eliminates the length bias that inflates loss on incorrect (often longer) completions.
Leave-One-Out	LOOP/RLOO	Drop one sample per group when computing the advantage baseline; unbiased variance reduction of the policy-gradient estimator.
Compact Filtering	**Novel (ours)**	Mask the loss on trajectories that hit max context, max steps, or rollout timeout (20 min); prevents reward collapse (Section 7.2).
No Entropy Loss	**Novel (ours)**	Remove the explicit entropy-bonus term; over long horizons the bonus drives entropy to blow up and collapses training. If the base model's token-level entropy stays in [0 3 1 0], the bonus is unnecessary.

Table 7.1: The seven components of GRPO++. The first five are drawn from prior work; the last two — compact filtering and the removal of the entropy bonus — are contributions of this work. Compact filtering is analyzed in depth in Section 7.2.

The public ablations most relevant to the final agent are the compact-filtering curves in the next subsection; the release does not publish per-component SWE-Bench-Verified deltas for the full GRPO++ recipe.

Compact Filtering

Compact filtering is the most important long-horizon adjustment in GRPO++. DAPO introduced *overlong filtering* for single-step settings: completions that hit the maximum context length are masked from the loss. DeepSWE generalizes this rule to multi-turn agents,

where a trajectory can also terminate by hitting the maximum number of environment steps or by timing out during generation or environment execution.

The rule is simple: mask the loss for any trajectory that reaches max context, max environment steps, or a twenty-minute timeout. Only trajectories that explicitly call `finish`/`submit` within the resource budget contribute gradient. This matters because sparse 0 1 rewards can otherwise reinforce trajectories that happen to contain a correct patch but continue with noisy post-fix behavior. Compact filtering makes the positive signal correspond to both solving the task and deciding to stop.

The public release gives two pieces of evidence. In Figure 7.1, the 14B ablation without compact filtering collapses, while the filtered run continues improving. In Figure 7.2, response length decreases while the number of environment steps increases, suggesting shorter, more tool-interleaved reasoning after the filter is enabled.

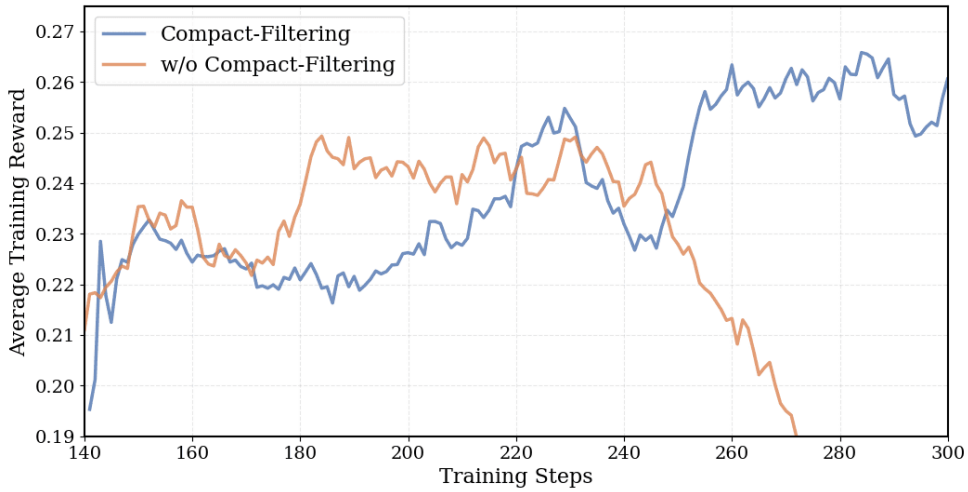


Figure 7.1: Compact-filtering ablation. Without the filter, correct-by-luck rollouts reinforce post-submission thrashing and the reward curve collapses; with the filter only deliberately terminated trajectories contribute gradient, and training continues to improve. Reproduced from Luo et al. [165].

The compact-filtering ablation is on the 14B sibling rather than DeepSWE-Preview itself, and it reports training reward rather than a direct SWE-Bench-Verified score. We therefore treat compact filtering as the documented stability lesson, not as a numerically isolated contribution to the final 42 2% Pass@1.

7.3 Results and Test-Time Scaling

Five numbers summarise DeepSWE-Preview's headline result. On the 500-problem SWE-Bench-Verified benchmark, evaluated through the official R2E-Gym codebase at 64K

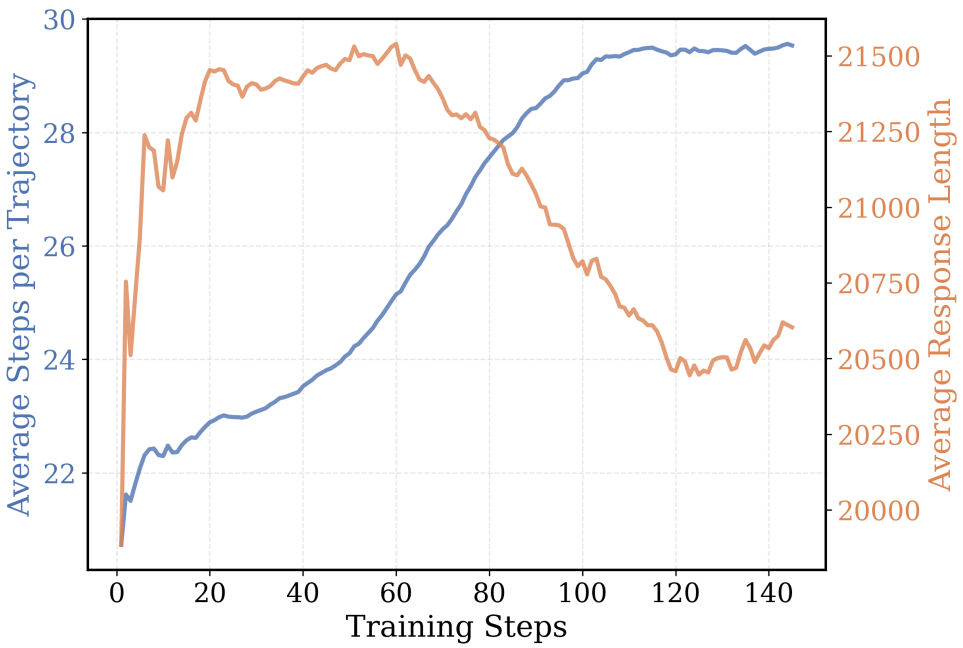


Figure 7.2: Behavioral signature of compact filtering. Once random post-submission tokens stop being rewarded, average per-step response length declines while the average number of environment steps per trajectory grows, indicating that reasoning is redistributed across more, shorter, tool-interleaved steps. Reproduced from Luo et al. [165].

context and 100 max environment steps and scored with the official SWE-Bench harness, DeepSWE-Preview reaches 42 2% Pass@1 averaged over sixteen runs, 71 0% Pass@16, 57 9% hybrid Best@8, and 59 0% hybrid Best@16. The 59 0% figure is a new open-weights state of the art at release, and the gap from 42 2% to 71 0% fixes the inference-time budget that Chapter 8's hybrid verifier then closes. Figure 7.3 shows the training-reward curve that produces the Pass@1 number: two hundred GRPO++ iterations move SWE-Bench-Verified Pass@1 from 23% to 42%, a twenty-point lift from the base Qwen3-32B's zero-shot agent performance, with no SFT cold-start and no proprietary-teacher trajectories in the loop.

Table 7.2 situates the agent against contemporary open-weight SWE systems. In single-rollout mode, DeepSWE-Preview reaches 42 2% Pass@1, below Devstral-Small but above the other listed open-weight agents. With hybrid test-time scaling, it reaches 57 9% Best@8 and 59 0% Best@16, substantially above the prior open-weight TTS number reported for Skywork-SWE plus an execution-free verifier (47 0%).

Two claims follow. First, DeepSWE-Preview is the first open-weight SWE agent to clear 40% Pass@1 on SWE-Bench-Verified without SFT distillation from a proprietary teacher: training with only the sparse 0 1 executable-specification reward outperforms prior approaches that use similar or more training data *and* SFT trajectories from stronger closed models. Second, the 71 0% Pass@16 ceiling bounds the additional lift a perfect verifier could

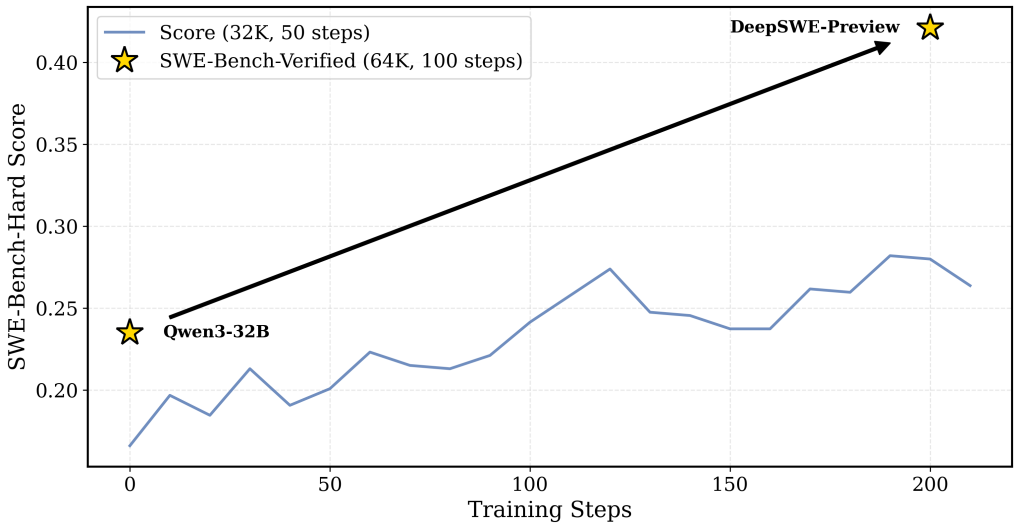


Figure 7.3: Validation score during GRPO++ training. Two hundred RL steps move the agent from roughly 23% to 42% Pass@1 on SWE-Bench-Verified, starting from Qwen3-32B with no SFT cold-start. Reproduced from Luo et al. [165].

Model	Scaffold	Type	SWE-Bench-Verified (%)
DeepSWE-Preview (32B)	R2E-Gym	Agent + Hybrid Best@16	**59 0**
DeepSWE-Preview (32B)	R2E-Gym	Agent + Hybrid Best@8	57 9
DeepSWE-Preview (32B)	R2E-Gym	Agent (Pass@1)	42 2
Devstral-Small (24B)	OpenHands-LM	Agent	46 6
OpenHands-LM (32B)	OpenHands-LM	Agent (Iterative)	37 2
SWE-Agent-LM (32B)	SWE-Agent-LM	Agent	40 2
R2EGym-Agent (32B)	R2E-Gym	Agent	34 4
Skywork-SWE (32B)	OpenHands-LM	Agent	38 0
Skywork-SWE (32B)	OpenHands-LM	Agent + Exec-Free Best@8	47 0
SkyRL-Agent (14B)	OpenHands-LM	Agent	21 6

Table 7.2: SWE-Bench-Verified Pass@1 (and Best@K where applicable) for contemporary open-weight agents. DeepSWE-Preview + hybrid test-time scaling reaches 59 0%, the best open-weight result at release; the single-rollout 42 2% is itself the first open-weight, pure-RL number to clear 40% without SFT distillation from a proprietary teacher. Numbers reproduced from [165].

extract from the same agent at K=16 — a 28 8-point gap that Chapter 8's hybrid verifier closes to within about twelve points (59 0% vs. 71 0%) using test-time scaling alone.

Test-Time Scaling

Test-time compute for DeepSWE-Preview scales along two orthogonal axes: the number of *tokens* the agent is allowed to spend inside a single trajectory, and the number of independent *rollouts* the agent produces before a verifier selects one. The two axes behave very differently on SWE-Bench-Verified: token scaling is nearly flat past 32K, whereas rollout scaling with a good verifier adds more than sixteen points over Pass@1. We report the applied outcomes here and defer verifier mechanics to Chapter 8.

Token Scaling

Sweeping the maximum context length from 16K to 128K tokens, DeepSWE-Preview rises from roughly 40% at 16K to 43 2% Pass@1 at 128K, but the marginal gain beyond 32K is at most 2%, and the same flat shape holds for every baseline we evaluate. As the blog puts it, "for SWE-related tasks, scaling the number of output tokens does not seem to be effective." This stands in sharp contrast to LiveCodeBench, where DeepCoder-14B-Preview scaled from 57 8% to 60 6% by doubling its context from 32K to 64K. The difference reflects what SWE agents are bottlenecked by: not how long they think at a step, but what they *do* at each step—which files they open, which commands they run, and when they commit to submit.

Rollout Scaling and Verifiers

Given N candidate trajectories, the agent must pick the one that actually solves the task. Three verifier families appear in the literature. An *execution-free* verifier is an LLM trained to score each trajectory or patch as correct or incorrect; DeepSWE-Verifier is trained for two epochs on correct/incorrect patches drawn from DeepSWE-Preview's own rollouts. An *execution-based* verifier instead uses a separate LLM to generate a diverse set of tests—including edge cases and regression tests—and picks the trajectory that passes the most. A *hybrid* verifier combines both signals, following the R2E-Gym design of Chapter 8.

Applied to DeepSWE-Preview on SWE-Bench-Verified (Figure 7.4), either verifier alone adds more than ten percentage points over the 42 2% Pass@1 baseline, confirming that verification—not generation—is where the headroom lies. The hybrid configuration is strictly better at every K and reaches 59 0% Hybrid-Best@16, a new open-weights state of the art at the time of release and a +16 8-point lift over Pass@1. The practical sweet spot is $K = 8$: hybrid Best@8 already reaches 57 9%, so the extra eight rollouts buy only 1 1 additional points. Full treatment of hybrid-verifier mechanics is in Chapter 8; here we report only the TTS outcome as applied to DeepSWE-Preview.

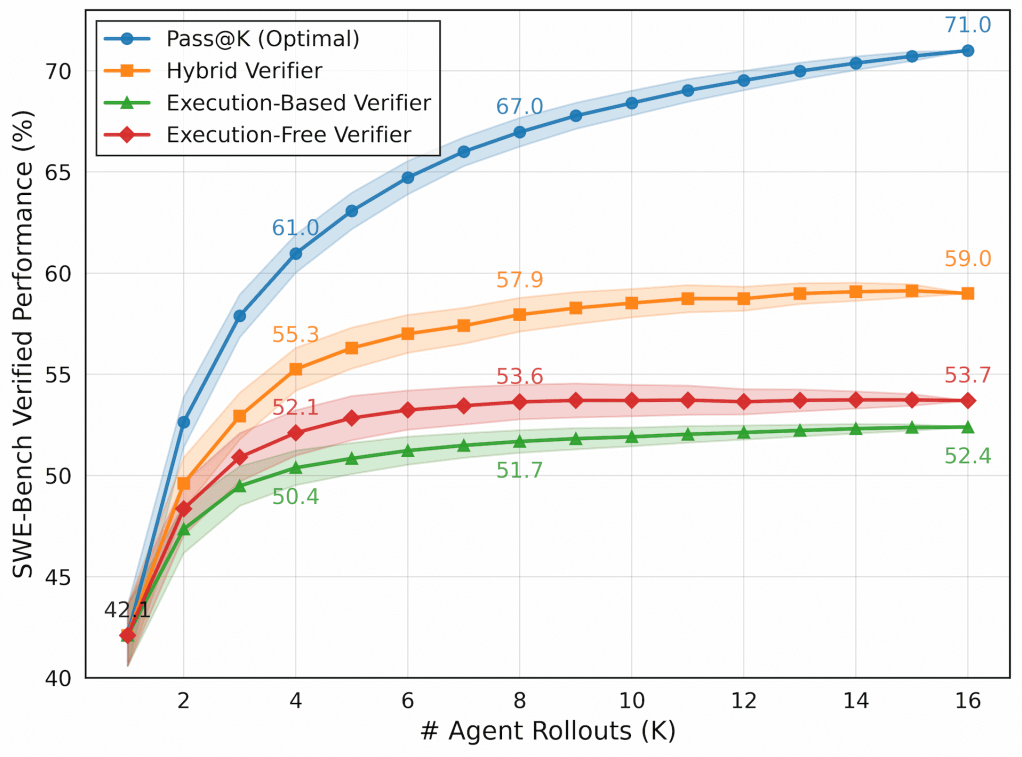


Figure 7.4: Rollout scaling with different verifier families on SWE-Bench-Verified. Hybrid TTS (Chapter 8) dominates each single-axis verifier at every K and reaches 59 0% Hybrid-Best@16, twelve points above the prior open-weights SOTA. Reproduced from Luo et al. [165].

Emergent Behaviors

The blog reports two qualitative behaviors that are useful as color. First, DeepSWE-Preview sometimes reasons through edge cases and searches for repository regression tests before submission. Second, it appears to allocate more thinking tokens to localization and fix design than to bookkeeping actions such as running a script or searching a file. These anecdotes are consistent with the main thesis claim: a sparse all-tests-pass reward can shape the agent's workflow, not just its final patch.

7.4 Negative Results and Limitations

Three training variants did not pan out, and they help delimit the pure-RL claim.

First, an *SFT cold-start from a proprietary teacher* did not help. We fine-tuned four variants of Qwen3-32B on Claude-Sonnet 3.7/4 trajectories, in thinking and non-thinking

modes, and then ran GRPO++ on top. All four variants stopped improving after roughly 100 RL iterations, and the SFT checkpoints were slightly weaker than SWE-Agent-LM.

Second, alternative training datasets were weaker. Running the same scaffold and reward on SWE-Smith and SWE-Gym produced limited gains and a high `solve-none` rate. The blog attributes the advantage of R2E-Gym to curriculum: its task pool provides enough solvable problems for group-relative RL to produce signal.

Third, non-thinking mode underperformed. Running GRPO++ over Qwen3-32B with the thinking budget disabled yielded only marginal gains. These negatives suggest that the result depends on both a capable reasoning base model and a training distribution with enough reward density; they do not establish that the optimizer alone is sufficient.

Chapter Summary

DeepSWE-Preview shows that a SWE agent can be trained end-to-end with reinforcement learning, using only the executable-specification test suite as a scalar 0 1 reward, and reach 42 2% Pass@1, 71 0% Pass@16, and 59 0% hybrid Best@16 on SWE-Bench-Verified.

In the language of this thesis, this chapter completes the training arc by collapsing the executable specification to its simplest form: the scalar output of "all tests pass within the time budget." Where Chapter 5 used the same Pass2Pass and Fail2Pass tests as a commit filter, a back-translation seed, and an SFT reward, this chapter uses them as the only signal the agent sees throughout RL.

The limitations are equally important. The result depends on the R2E-Gym task distribution and on thinking-mode Qwen3-32B; other datasets and non-thinking mode produced limited gains. Because the public source is a blog post rather than a conference paper, this chapter does not claim an exact GRPO++ loss, a full RL hyperparameter table, per-component SWE-Bench-Verified ablations, per-repo error breakdowns, variance over the 16 runs, or solve-none curves for the failed datasets.

Part III now turns from training to agent design: how executable specifications power inference-time scaling (Chapter 8) and long-horizon task decomposition (Chapter 9).

Part III

Agent Design

Chapter 8

Hybrid Verifiers: Complementary Spec Signals for Inference-Time Scaling

The[1] R2E-Gym agent of Chapter 5 achieves Pass@1 = 34.4% on SWE-Bench-Verified. If we let the agent sample 26 trajectories and grade the pool using the gold fail-to-pass tests, performance rises to 64.4%. The gap between Pass@1 and oracle Pass@26 shows that many sampled pools contain a correct patch, but selecting it requires a verifier without access to the gold tests. This chapter introduces two complementary verifier families: an *execution-based* verifier that synthesizes reproduction tests and scores patches by pass count, and an *execution-free* verifier that scores candidates using a fine-tuned trajectory judge. While each saturates individually at 42–43%, their additive hybrid reaches 51.0% on SWE-Bench-Verified, setting a new open-weights state of the art at the time of publication. The hybrid works because the two axes fail differently: execution-based verifiers suffer from low distinguishability (few generated tests actually separate correct from incorrect patches), while execution-free verifiers are biased by stylistic features of the agent's thoughts. Because their failure modes differ, combining the two signals improves selection performance.

8.1 Inference-Time Scaling via Verifier-Based Aggregation

This chapter studies how verifiers can turn a fixed compute budget—multiple sampled candidate patches per task—into a reliable performance gain on SWE-Bench-Verified. The framework is the same one used by test-based re-rankers in algorithmic coding [70, 172]; the subject of study is what a verifier should actually *look at* in the SWE setting to maximize test-time performance.

[1] As noted in Chapter 5, this chapter is based on joint work with **Jaskirat Singh**, Manish Shetty, Liang Zheng, Koushik Sen, and Ion Stoica, published as *R2E-Gym: Procedural Environment Generation and Hybrid Verifiers for Scaling Open-Weights SWE Agents* at COLM 2025 [166]. Where Chapter 5 covered synthetic environment construction and SFT, this chapter focuses on the inference-time hybrid-verifier contributions.

The Pass@K gap. The R2E-Gym-32B editing agent of Chapter 5 reaches Pass@1 = 34 4% on SWE-Bench-Verified. Sampling K independent trajectories at temperature $T \geq 0\ 8$ lifts the oracle Pass@26—the fraction of problems for which at least one of 26 candidates is correct—to 64 4% [166]. Using this gap requires selecting the correct patch without access to gold tests.

The aggregator abstraction. Given N candidate trajectories $\mathcal{T}_1 \quad \mathcal{T}_N$ with patches $\mathcal{P}_1 \quad \mathcal{P}_N$ for a problem described by $\mathcal{D}$, a verifier V assigns scores $s_1 \quad s_N$ and returns the top-ranked patch:

$$k \ = \arg\max_{k \ [N]} V(\mathcal{D} \ \mathcal{T}_k \ \mathcal{P}_k) \qquad \mathcal{P} = \mathcal{P}_k \tag{8.1}$$

Verifier quality is measured by the Best@K metric, the fraction of problems for which $\mathcal{P}$ actually passes the held-out F2P tests. By construction, Best@K is upper-bounded by Pass@K: oracle ranking would reach the upper bound; any real V leaves some gap.

Sampling configuration. Following Pan et al. [135] and Xia et al. [172], we fix a single budget for every experiment in this chapter: one trajectory at $T = 0$ plus 25 trajectories at $T \quad 0\ 8 \ 0\ 9$ from the R2E-Gym-32B agent, on the 500 problems of SWE-Bench-Verified. That yields $K = 26$ candidate patches per problem, and all subsequent Best@26 numbers are reported on this fixed pool unless stated otherwise. Holding the candidate pool fixed makes verifier ablations directly comparable: the only degree of freedom is the scoring function V.

Section 8.2 introduces the two verifier families (execution-based and execution-free) we study; Section 8.3 diagnoses their failure modes; Section 8.4 proposes the hybrid rule that combines them; and Sections 8.5 and 8.6 report results and design ablations.

8.2 Two Verifier Families

Given the aggregation scaffolding of Section 8.1 — a task description $\mathcal{D}$, K sampled trajectories $\mathcal{T}_k \ _{k=1}^{K}$, and candidate patches $\mathcal{P}_k \ _{k=1}^{K}$ — we study two families of scoring functions s_k that differ in how they interrogate each candidate. *Execution-based* (EB) verifiers run tests against the patched repository and score by pass counts; *execution-free* (EF) verifiers consume the full ReAct trace plus the patch and emit a learned scalar. The two families arise naturally from a R2E-Gym environment: the same F2P tests that define correctness for training (Section 5.2) can either be *regenerated* at inference — a freshly-synthesized executable specification — or used *offline* to label trajectories for a learned trajectory judge. One infrastructure, two complementary signals.

Execution-Based Verifier: Self-Generated Tests

Architecture. Our execution-based verifier is built around a dedicated *testing-agent* that, given the issue description and repository state, writes a single Python test script containing

$M = 10$ diverse tests covering corner cases, multiple input types, and boundary behaviours. Unlike Agentless-style zero-shot prompting [172], the testing-agent is an agent: it explores the repository, inspects existing test conventions, iterates when a candidate test fails to execute, and only emits tests that are runnable end-to-end. Figure 8.1 depicts the data-flow: issue and code in, trajectory in the middle, test patch out.

Figure 8.1: **Execution-based verifier pipeline.** A trained testing-agent ingests the issue and the current repository state and emits a single `test_issue.py` script with $M = 10$ diverse reproduction tests. These tests are then run against each candidate patch to compute the EB score.

Base model and training. We use Qwen-Coder-32B as the base model and perform full supervised fine-tuning on 2 203 test-generation trajectories distilled from Claude-3.5-v2. The mix includes both positive and negative trajectories with only minimal rejection sampling: we deliberately keep imperfect trajectories so the student learns to recover from execution errors rather than to mimic idealized behaviour.

Django starter. A recurring failure mode in preliminary runs was Django-heavy repositories, where tests must configure settings, register models, and run migrations before any assertions can execute. We therefore inject a fixed in-context starter lifted from the Django test suite into every testing-agent prompt. This single addition accounts for roughly 2% of resolved problems and is the only piece of repository-specific scaffolding we found necessary at inference time.

Scoring. For each candidate patch $\mathcal{P}_k$, the execution-based score is the number of generated tests that pass, *gated* on a regression-test filter borrowed from Agentless [172]. Writing RS_k for the patch's regression score (fraction of existing repository tests it still passes) and $\mathrm{Pass}(\mathcal{P}_k \ \mathrm{Test}_i)$ 0 1 for a single generated test, the score is

$$s_k^{EB} = \begin{cases} \sum_{i=1}^{M} \mathrm{Pass}(\mathcal{P}_k \ \mathrm{Test}_i) & \text{if } \mathrm{RS}_k = \max_{j \ [1\ K]} \mathrm{RS}_j \\ 0 & \text{otherwise} \end{cases} \tag{8.2}$$

The regression filter zeros out any patch that breaks existing behaviour; among the survivors, the generated-test count tiebreaks. Two properties follow from this structure. First, s_k^{EB} $\mathbb{Z}_{\geq 0}$ is integer-valued, so many patches can tie — a weakness we quantify in Section 8.3. Second, every component of the score is derived from *executing a real Python program* against a real environment: the generated test suite is a concrete, runnable artifact, not a prose judgement.

Executable-spec interpretation. This point matters for the thesis framing. The testing-agent's output is an executable specification synthesized *at inference time* for the specific issue in front of it: a fresh $\mathcal{S}$ in the language of Section 1.2, produced on demand, and consumed immediately by the aggregator. Everywhere else in this dissertation, executable specifications supply training or evaluation signal; the EB verifier is the most explicit case of specs acting at inference. The remaining weaknesses of the EB signal — low distinguishability, occasional toxicity — are therefore the weaknesses of inference-time spec synthesis itself, and we dissect them in Section 8.3.

Execution-Free Verifier: Learned Trajectory Judge

Architecture. Our execution-free verifier is a Qwen-Coder-14B model fine-tuned to predict a single `YES`/`NO` token given the task description $\mathcal{D}$, the full ReAct trajectory $\mathcal{T}_k = a_1\ o_1 \quad a_n\ o_n$, and the final patch $\mathcal{P}_k$. Unlike the EB verifier, it never executes code — it reads the trajectory and may rely on trajectory features (which the analysis in Section 8.3 turns into a specific concern). Figure 8.2 summarises the pipeline.

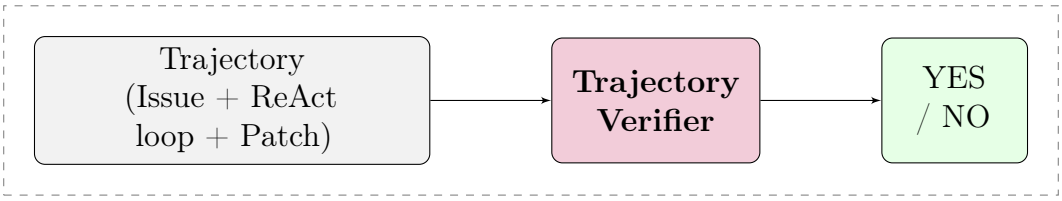


Figure 8.2: **Execution-free verifier pipeline.** A Qwen-Coder-14B model reads the issue, the full ReAct loop, and the final patch, and predicts a single `YES` / `NO` token whose normalized probability is the EF score.

Scoring. Following the outcome-supervised reward-model formulation of Cobbe et al. [173] and the verifier recipe of Pan et al. [135], we define the EF score as the normalized log-probability of the `YES` token against its `NO` alternative:

$$s_k^{EF} = \frac{P(\texttt{YES} \quad \mathcal{D}\ \mathcal{T}_k\ \mathcal{P}_k)}{P(\texttt{YES} \quad \mathcal{D}\ \mathcal{T}_k\ \mathcal{P}_k) + P(\texttt{NO} \quad \mathcal{D}\ \mathcal{T}_k\ \mathcal{P}_k)} \qquad [0\ \ 1] \tag{8.3}$$

The two probabilities are read off from the final-token logits at inference. Unlike s_k^{EB}, s_k^{EF} lives on a *continuous* scale — a feature that matters at inference time, because ties on the integer EB score can be broken by the fractional EF score. That combinatorial observation is what motivates the hybrid in Section 8.4.

Training data and hyperparameters. We train on 5 700 trajectories balanced between correct and incorrect. The seed pool mixes offline trajectories from Claude-3.5-v2 with on-policy rollouts from the R2E-Gym-trained 32B editor; following Pan et al. [135], the on-policy mix is important for final verifier quality. Crucially, the correct/incorrect labels

come from *running* each trajectory's patch against the R2E-Gym environment's ground-truth F2P tests — so the EF verifier's training signal is itself a direct downstream product of executable specifications, even though the verifier never executes anything at inference. We fine-tune with LoRA at rank 64 using LLaMA-Factory [141]. We reuse the prompt template from Pan et al. [135], which asks the model to emit `YES` for correct trajectories and `NO` for incorrect ones at the final position.

Executable Specs as Two Signals

The two verifiers look like different architectures but they are grounded in the same object. The EB verifier runs a Python test suite that the testing-agent has just synthesized — an executable specification produced at inference for this issue. The EF verifier does not execute anything at inference, but its training labels were produced by executing R2E-Gym's F2P specs against thousands of trajectories; the verifier has internalized those execution outcomes into a learned continuous score. Both signals therefore flow from *one infrastructure* — a SynGen environment with F2P tests — and produce *two signals* with different distributional properties: integer-valued and grounded on one side, continuous and discriminative on the other. The rest of this chapter studies the characteristic failure modes of each signal (Section 8.3), combines them into a single hybrid score (Section 8.4), and shows that the combination improves over prior open-weight verifier results on `SWE-Bench Verified` (Section 8.5).

8.3 In-Depth Analysis: Strengths and Weaknesses

Both verifiers elicit substantial inference-time gains, and both saturate at similar observed performance levels. On SWE-Bench-Verified, the execution-based score saturates at Best@K $\approx$ 43 7% and the execution-free score at Best@K $\approx$ 42 8%, more than twenty percentage points below the oracle Pass@26 = 64 4% value. The two verifier axes fail in structurally different ways: EB suffers from limited distinguishability, while EF can be biased by agent thoughts over the final output patch.

Limited Distinguishability of Generated Tests

The execution-based score $s_k^{EB} =$ $_i$ Pass($\mathcal{P}_k$ Test$_i$) is a sum of integer pass counts. Two patches that pass the same set of generated tests tie, and the EB verifier has no further resolution to separate them. A test is distinguishing if it separates the best correct patch from the best incorrect one.

We compute the distinguishability rate per problem and plot the problem-level density across SWE-Bench-Verified in Figure 8.3 (left). The result shows that for the majority of problems, fewer than 20% of generated tests provide discriminative signal. The finding is robust across three qualitatively different test generators—our trained Qwen-Coder-

32B testing agent, a prompted CLAUDE-3.5-V2 baseline, and the zero-shot `Agentless-1.5` reproduction tests [172].

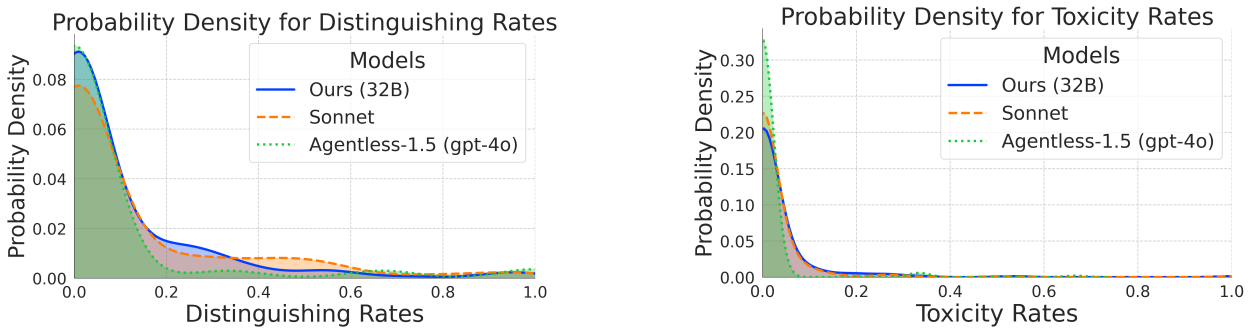


Figure 8.3: **Analyzing limitations of execution-based verifiers. Left:** Problem Probability Distributions for distinguishability rates depicting weak discrimination capabilities of tests. We observe that for the majority of problems, less than 20% of tests provide discriminative signal, constraining the re-ranking ability of test-based agent. **Right:** Distributions for toxicity rates showing (rare) generation of toxic tests. We find that execution-based verifiers are also vulnerable to (rare) generation of toxic tests (tests that pass incorrect patches but fail correct patches); which can undermine the reliability of execution-based verifiers.

Why are so few generated tests distinguishing? Figure 8.4 depicts that most generated tests either do not reproduce the bug (high Pass Pass values) or do not pass ground truth patches (high Fail Fail values) primarily due to bugs or exceptions in the generated test cases.

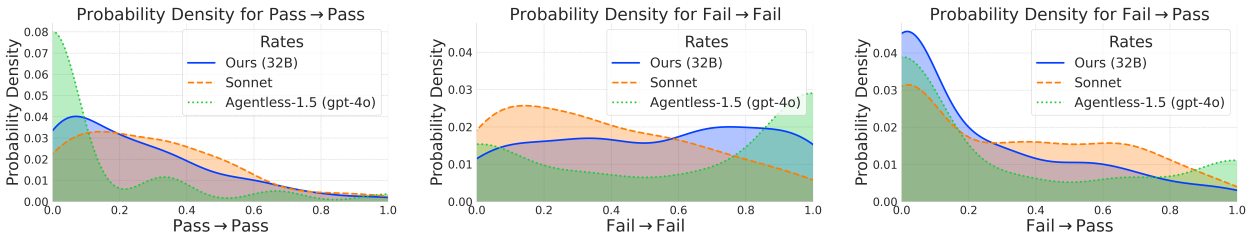


Figure 8.4: Problem Probability Distributions for Pass Pass, Fail Fail, and Fail Pass generated test fractions for various approaches. We identify a large fraction of generated tests either do not reproduce the bug (left) or do not even pass the correct solution (middle).

Test Toxicity

Following the framing of Chen et al. [70], we call a generated test *toxic* if it passes more incorrect patches than correct ones. If it enters the EB sum, it counts toward an incorrect patch's score and can push that patch past a correct one in the final ranking.

Figure 8.3 (right) shows the problem-level density of toxicity rates. In aggregate, toxicity is rare—most problems produce zero toxic tests—but on a small but non-trivial subset of SWE-Bench-Verified problems the toxicity rate reaches roughly 10%. That is enough to mis-rank patches on those problems, motivating a Top_n gate driven by a different signal in Section 8.4.

Execution-Free Verifiers can rely on heuristics

The execution-free verifier produces a continuous score s_k^{EF} [0 1] and therefore does not share EB's resolution problem. However, we find that execution-free verifiers can be biased by agent thoughts over the final output patch.

Quantitative ablation. We train multiple execution-free verifiers excluding different trajectory components. Figure 8.5 (a) reports the full-input reference, a *patch-only* variant that receives only the final unified diff, and a *trajectory-minus-thoughts* variant that retains actions and observations but strips the agent's chain-of-thought tokens. The final Best@26 drops from 42 8% to 37 6% when we remove the trajectory from the verifier input (i.e., only use the final patches). This means that while the patch alone is responsible for determining correctness, execution-free verifiers rely on other heuristics, such as agent thoughts, to make predictions.

Qualitative attention analysis. To further investigate this phenomenon, we perform an attention analysis to visualize parts of the input trajectory which are most relevant while predicting the output success. Figure 8.5 (b) illustrates the top two windows receiving the highest attention scores, demonstrating that verifiers disproportionately attend to agent thoughts. This can be misleading since the verifier can use the sentiment signals in these thoughts as proxies for correctness rather than evaluating the technical merits of the solution (i.e. the output patch).

The Two Axes Are Complementary

Putting the three diagnoses together supports the interpretation that the two verifier families provide complementary information. EB provides direct signal for patch correctness through execution but suffers from limited distinguishability. EF offers better distinguishability between patches through a continuous reward score but can be biased to pay more attention to heuristics (e.g., agent thoughts) over the final output patch. A hybrid verifier that leverages

Method	Acc. (%)	Best@26 (%)
Final Patch + Traj.	**71.82**	**42.8**
Patch Only	68.01	37.6
Traj. - Thoughts	68.77	41.4

(a) **Impact of Patch & Thoughts** on execution-free verifier. Patch alone reduces performance, indicating that model relies on other heuristics (e.g., agent thoughts) for reranking; which can be misleading (see part-b: right).

1. Successfully reproduced the issue
2. Implemented a fix [...]
4. Ensured edge cases are handled
5. Maintained backward compatibility [...]

<function=finish>submit</function> [...]

Great! The fix works. Let's see what we did to fix the issue:
1. We identified that the original code was failing because it was
, trying to use the '. inverse ()' method directly on
, permutations, which [...]

(b) Top two attention windows while predicting `YES` for an incorrect trajectory. We find that focusing on heuristics (agent thoughts) can mislead the verifier.

Figure 8.5: **Quantitative and qualitative analysis on limitations of execution-free verifiers.** We perform two experiments: a) Quantitative ablations on the impact of output patch on verifier performance; showing that execution-based verifiers rely on other heuristics (e.g., agent thoughts) over the final patch. b) Qualitative visualization analyzing top $k = 2$ sliding windows with highest mean attention score while predicting output token `YES` for an *incorrect* agent trajectory (`sympy__sympy-24443`: SWE-Bench [104]). Focusing on heuristics (*e.g.*, agent thoughts) can be misleading, and the verifier predicts the trajectory as correct.

the strengths of both approaches therefore has a principled reason to outperform either axis alone. That is the hybrid rule we introduce in Section 8.4.

8.4 Hybrid Verifier Formulation

The diagnosis in Section 8.3 motivates a simple hybrid scoring rule. Execution-based (EB) scores provide direct signal for patch correctness through execution but suffer from limited distinguishability. Execution-free (EF) scores offer better distinguishability between patches through a continuous score but can be biased to pay more attention to heuristics over the final output patch. Combining the signals should leverage the strengths of each approach.

We therefore define the hybrid score

$$s_k^H = \mathrm{Top}_n\big(s_k^{EF}\big) + s_k^{EB} \qquad \mathrm{Top}_n\big(s_k^{EF}\big) = \begin{cases} s_k^{EF} & \text{if } s_k^{EF} \text{ is among the top } n \text{ EF scores} \\ - & \text{otherwise} \end{cases} \tag{8.4}$$

and rank candidate patches by s_k^H. Three operative choices: additive combination, Top-n gating on EF, and regression filtering after Top-n. Each addresses a specific failure mode of the constituent verifiers.

Additive, no learnable weights. We add the two scores directly. EB lives in $\mathbb{Z}_{\geq 0}$ as a pass count over M=10 generated tests; EF lives in [0 1] as a normalized token probability. The scales are not accidental: EB's integer units produce the coarse ordering, and EF's fractional tail acts as a tie-breaker among the very common EB ties.

Top-n filter on EF first. Only candidates whose EF score sits in the top-n tier remain eligible; everything else gets $-$ and drops out. EF restricts the candidate set to patches assigned high plausibility by the EF verifier, and EB then executes its tests on that shortlist. The effect may reduce exposure to test toxicity: toxic tests can only rank shortlisted patches, and shortlisted patches, which have high EF scores, are already more likely to be correct.

Regression filter after Top-n. Following Xia et al. [172], we apply the regression-test filter *after* the Top-n cut. Applying it within the Top-n pool keeps non-zero scores available while still eliminating patches that break existing behaviour.

Each piece answers a specific failure mode diagnosed in Section 8.3: EB supplies the execution-derived correctness signal, EF supplies the continuous distinguisher for EB ties, and Top-n restricts the hybrid verifier to only consider the top verifier-ranked patches. Results in Section 8.5 confirm that the hybrid continues to improve as K grows while EB and EF saturate.

8.5 Results

This chapter reports three main performance values. Single-axis EB saturates at Best@26 $\approx$ 43 7%. Single-axis EF saturates at Best@26 $\approx$ 42 8%. The hybrid score of Equation (8.4) reaches Best@26 = 51 0% on SWE-Bench-Verified — a new open-weights state of the art. We report the result in two views: a comparison against open and proprietary baselines, and the Best@K scaling curve that shows how complementarity changes with K.

Headline Performance on SWE-Bench-Verified

Table 8.1 situates R2E-Gym-32B against contemporary open-weight and proprietary systems on the 500-problem SWE-Bench-Verified benchmark. At 32B scale with hybrid test-time scaling, R2E-Gym reaches 51 0%. The proposed approach significantly outperforms other open-weight alternatives. Among other generalist-agent methods, SWE-Gym [135] achieves

a Best@16 performance of 32 0%. Similarly, concurrent work [174] achieved 41 0% using RL and Best@500 (using Agentless). In contrast, despite mainly relying on supervised fine-tuning for training, our proposed approach achieves a Pass@1 of 34 4% with Best@26 performance of 51 0% — achieving strong performance improvements through more scalable data curation and better test-time scaling.

Table 8.1: Pass@1 and Best@K on SWE-Bench-Verified across proprietary and open-weight systems. R2E-Gym-32B with the hybrid verifier reaches 51 0% Best@26, a new open-weights state of the art at the time of publication. Adapted from Jain et al. [166].

Method	Model	Type	Verified
Proprietary Models			
Agentless-1 5 [172]	GPT-4o	Pipeline	34 0
Agentless [172]	o1	Pipeline	48 0
Claude + Tools	Claude-3 6-Sonnet	Agent	49 0
Agentless-1 5 [172]	Claude-3 6-Sonnet	Pipeline	50 8
OpenHands [136]	Claude-3 6-Sonnet	Agent	53 0
Claude + Tools	Claude-3 7-Sonnet	Agent	62 3
Claude + Tools (Best@Any)	Claude-3 7-Sonnet	Agent	70 3
Open-weight Models			
SWE-SynInfer [175]	Lingma-SWE-GPT-72B	Agent	30 2
SWE-Fixer [139]	SWE-Fixer-72B	Pipeline	30 2
SWE-Gym (Best@16, Verifier) [135]	SWE-Gym-32B	Agent	32 0
SWE-RL (Best@500, Tests) [174]	SWE-RL-70B	Pipeline	41 0
Agentless [172]	DeepSeek-R1	Pipeline	49 2
R2E-Gym (Pass@1)	R2E-Gym-32B	Agent	**34 4**
R2E-Gym (Best@16, Hybrid)	R2E-Gym-32B	Agent	**49 4**
R2E-Gym (Best@26, Hybrid)	R2E-Gym-32B	Agent	**51 0**

Best@K Scaling Curves

Figure 8.6 plots Best@K as a function of editing-agent rollouts on the fixed 26-candidate pool. Both execution-based and execution-free verifiers demonstrate substantial performance improvements with increased number of rollouts. However, Best@K rate quickly saturates for both methods, converging similarly to 43 7% and 42 8% respectively. The hybrid curve sits strictly above both from K=2 onward, hits 49 4% at K=16, and reaches 51 0% at K=26.

This shape is the chapter's quantitative claim compressed into one picture: the complementarity persists as K increases. The hybrid grows while both of its ingredients saturate, because the two signals make different errors.

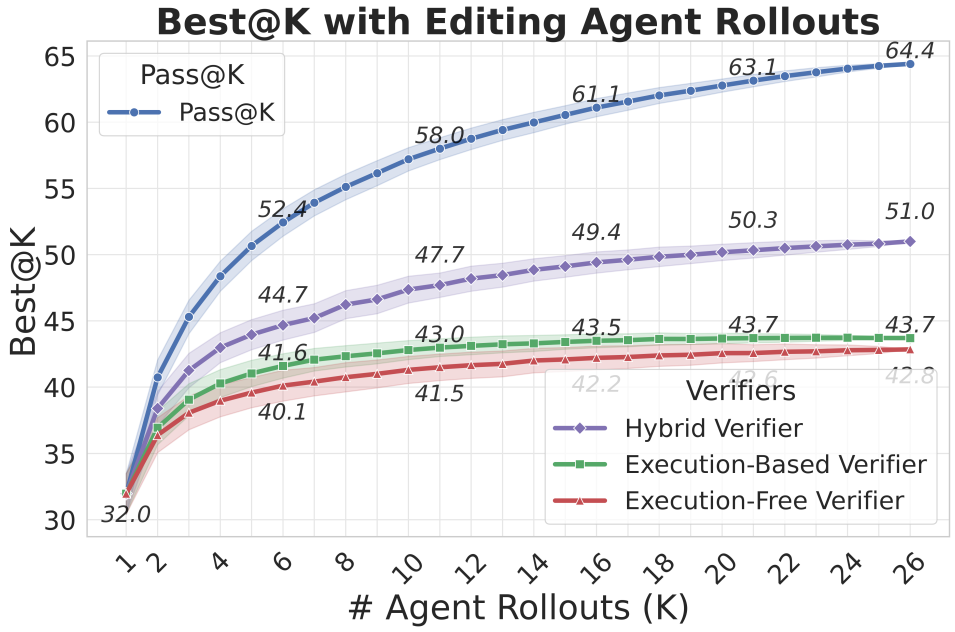


Figure 8.6: BEST@K versus editing-agent rollouts K on SWE-BENCH-VERIFIED. Execution-based and execution-free verifiers saturate below 44% by $K \approx 10$; the hybrid verifier of Equation (8.4) stays strictly above both axes and reaches 51 0%. Adapted from Jain et al. [166].

8.6 Ablation Studies on Hybrid Verification Design

We close the chapter with five controlled ablations on the hybrid design of Section 8.4, each holding every other factor fixed and varying a single design choice. Figure 8.7 collects the main numbers; the paragraphs below discuss each design choice in turn. Throughout, $K = 26$ is the editor rollout budget and the evaluation benchmark is SWE-BENCH-VERIFIED.

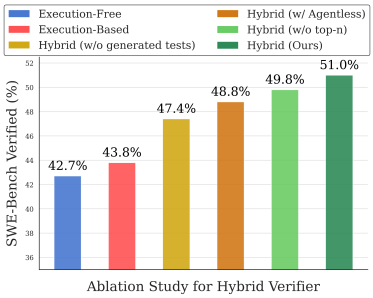


Figure 8.7: **Ablation Study on Hybrid Verifier.** We find three key insights: 1) While both execution-based and execution-free verifiers saturate around 42-43%, the hybrid approach yields significantly higher test-time gains (51.0%). 2) Regression tests alone are insufficient for hybrid scaling | achieving only 47.4% aggregation performance. 3) Agentic vs Agentless: training a specialized testing agent is important improving the performance from 48.8% to 51.0%.

Variation with Test-Agent Rollouts. As in Section 8.3, execution-based test generation can suffer from a lack of distinguishing tests. One approach to address this is to sample more test-agent rollouts. We quantify this effect in Figure 8.8. We observe that increasing the number of test-agent rollouts consistently helps improve performance with our hybrid approach.

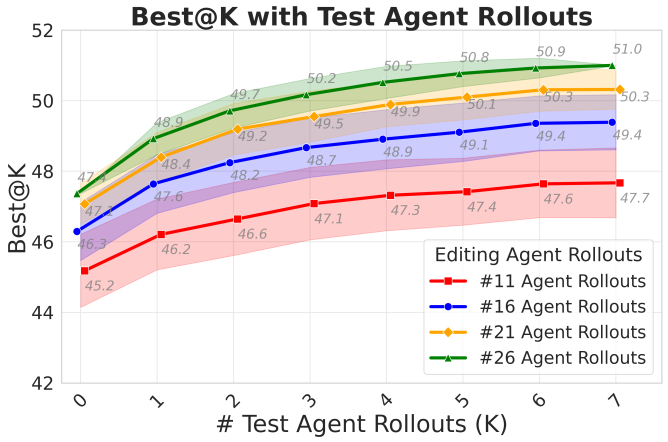


Figure 8.8: Best@26 as a function of test-agent rollouts, with editor rollouts held at $K = 26$. More test rollouts monotonically improve the hybrid score.

Compute-Efficient Rollouts. Interestingly, we find that sampling more test-agent rollouts can provide more compute-optimized inference-scaling over naively sampling more editing-agent rollouts. For instance, increasing the number of editing-agent rollouts from 16 to 21 improves the Best@K performance from 47 6% to 48 4%. In contrast, simply sampling 5 more test-rollouts can yield better gains (Best@K 49 3%). Note that test-agent rollouts are also usually considerably cheaper than editing-agent rollouts.

Regression Tests Alone are Insufficient. Our execution-based verification framework integrates both regression and generated reproduction tests. Figure 8.7 isolates the impact of regression tests alone on the final performance. While regression tests alone improve performance from 42 9% to 47 4%, using generated tests further enhances performance to 51 0%, demonstrating that both test types provide essential and complimentary signals.

Agentic vs Agentless Tests. A distinguishing feature of our approach is to train a specialized agent for test-generation instead of the zero-shot approach from Xia et al. [172]. To evaluate this design choice, we conducted a controlled comparison using official Agentless tests from their released artifact [176] within our hybrid verification framework. Figure 8.7 demonstrates that while Agentless tests provide meaningful performance improvements, our

agent-generated tests yield superior results (51 0% versus 48 8%), validating our agent-based approach to test generation.

Role of $\text{Top}_\mathbf{n}$. We evaluate the impact of the Top_n filtering mechanism introduced in Equation (8.4). Removing the gate—letting the execution-based score rank every patch—gives 49 8% BEST@26; restoring the gate at $n = K$ 2 raises it to 51 0%. This improvement likely stems from mitigating the impact of toxic tests by restricting their application to higher-quality patches (identified via execution-free reward scores), thereby enhancing the reliability of the verification process.

8.7 Related Work

SWE-coding verifiers. Verifier design for real-world GITHUB issue resolution has crystallised around two lines of work that this chapter directly builds on. *Agentless* and *Agentless-1.5* [172] pair zero-shot reproduction-test generation with a regression-test filter that rejects patches breaking existing behaviour. Committee-style approaches aggregate votes across multiple agents and LLM judges to improve diversity [177]. Closest to our execution-free half is the SWE-Gym trajectory verifier of Pan et al. [135], a Qwen-14B judge fine-tuned on correct/incorrect trajectories.

General-code verifiers. Outside the SWE setting, test-based re-rankers have a long lineage on isolated-function benchmarks: CODET [70]; ALPHACODIUM [13]; Speak-Code [93]; and ALGO [62]. A parallel thread trains neural re-rankers that score candidates by learned features of the program or its execution traces [87, 96, 178]. The third line is explicitly sceptical of LLM-as-judge: Gu et al. [179] show that large models are poor at judging code correctness from surface features alone.

Novelty positioning. Prior work studies execution-based and execution-free verifiers in isolation, and typically on algorithmic puzzles rather than on repository-scale issue resolution. This chapter presents an empirical dual-axis analysis that makes the limited distinguishability of test-based scoring and the bias of trajectory judging visible and measurable on the same candidate pool, together with an additive hybrid with a Top_n gate that converts their complementarity into strong test-time performance. Both halves are made practical by the SYNGEN substrate of Chapter 5: procedurally generated F2P environments provide execution-based scoring at inference, and they also provide the ground-truth trajectory labels that train the execution-free judge.

8.8 Chapter Summary

Self-generated reproduction tests and a learned trajectory judge are two signals with complementary failure modes, and an additive hybrid with a Top-n gate converts that complementarity into a performance of 51 0% on SWE-Bench-Verified.

In the language of this thesis, self-generated tests are executable specifications synthesized on demand at inference. The hybrid verifier exploits the complementarity of two spec-grounded signals: one literal, running the testing-agent's synthesized Python tests against each candidate patch, and one learned, a trajectory judge trained against fail-to-pass gym labels. This chapter demonstrates that executable-specification reasoning applies at inference time as well: the same infrastructure that supplied the SFT filter in Chapter 5 and the RL reward in Chapter 7 also supplies the candidate-verifier data used for inference-time aggregation.

A key limitation bounds the execution-based axis: it bottlenecks on distinguishability, with fewer than 20% of generated tests separating correct from incorrect patches on most problems.

The final chapter extends the executable-specification framing from inference aggregation to long-horizon decomposition: per-step equivalence tests and dynamic-analysis annotations that guide a multi-thousand-line `C`-to-`Rust` translation; see Chapter 9.

Chapter 9

Syzygy: Per-Step Equivalence Tests and Dynamic-Analysis Specs for Long-Horizon Translation

Legacy[1] C remains widely used in operating systems, codecs, cryptographic libraries, and network stacks, but it is a major source of memory-safety vulnerabilities [181]. Rust offers comparable performance while preventing many classes of memory-unsafe behavior, including use-after-free, data races, and unchecked out-of-bounds access. This chapter asks whether translating C to safe Rust can be automated at the scale of a real library, such as Google's Zopfli compression library [182]. Long-horizon translation cannot be graded by an end-to-end test alone. It requires two new forms of executable specification: *(i)* per-step function-level equivalence tests mined at run time from top-level executions, and *(ii)* structural dynamic-analysis annotations (aliasing, nullability, allocation size, underlying element type). The first grades progress one function at a time; the second tells the LLM *which* safe-Rust type each C pointer must take before any equivalence test is even compiled. Syzygy realizes this approach by pairing an LLM-driven translator with a dynamic specification miner. Applied to Zopfli (3 K lines of C), Syzygy produces roughly 7 K lines of safe Rust with no `unsafe` blocks, validated over 1000000 equivalence inputs. This represents the largest automatically produced, test-validated C-to-safe-Rust translation on record as of publication.

9.1 Introduction

C remains widely used in systems software — operating-system kernels, video and audio codecs, cryptographic libraries, and network stacks — and it is also, by a wide margin, the

[1]This chapter is based on joint work with **Manish Shetty**, **Adwait Godbole**, Sanjit A. Seshia, and Koushik Sen, released as the 2024 preprint *Syzygy: Dual Code-Test C-to-(safe)-Rust Translation using LLMs and Dynamic Analysis* [180] (also presented in the LLM4Code workshop track). Manish Shetty, Adwait Godbole, and Naman Jain are joint first authors.

language responsible for the most memory-safety CVEs [181]. Rust is a systems language with performance comparable to C, a CLANG-based toolchain that overlaps the C build path, and — critically — an ownership and borrow-checking discipline that prevents use-after-free, data races, and unchecked out-of-bounds accesses. Both the U.S. National Security Agency and DARPA's TRACTOR program [183] have encouraged migration of legacy C code toward memory-safe languages. For example, compression libraries form a project milestone of the TRACTOR program, and ZOPFLI [182], Google's production deflate/LZ77/Huffman compression library, is a relevant target. The question this chapter takes up is whether the translation step can be automated without giving up the very safety guarantees that motivate the migration.

Automation is hard in a specific way. Manual migration of even a few thousand lines of C consumes person-weeks of careful work, because the hardest cases arise where C types omit runtime information needed by Rust's type system. A `T*` argument in C can stand for a singleton, a stack array, a heap-allocated buffer, or a moving iterator into one of the above; it may or may not be nullable; it may or may not alias some other pointer argument on a given call site; and when it is a `void*` it does not even commit to an element type. The Rust borrow checker, by contrast, insists that the translator commit ahead of time to one of `\&T`, `\&mut T`, `Option<\&T>`, `Vec<T>`, `Rc<T>`, `Rc<RefCell<T»`, or a handful of close relatives, and it rejects every safe-Rust program in which those commitments fail to line up. The information that disambiguates the choice — aliasing, nullability, allocation size, underlying element type of a `void*` — is intrinsically *runtime* information. It is not in the C type, and it is difficult to recover precisely with static analysis in real codebases.

Approach	`safe` Rust	Validity	Generation Technique
c2rust [184]		Cons,	Rule-based (exposes C types in Rust)
VERT [185]		Verif,	LLMs + verification feedback
CROWN [186]		Cons,	Bootstrap analysis on c2rust
Shiraishi *et al.* [187]		Test,	LLMs + compile feedback
SACTOR [188]		Test,	LLMs + *static* analysis + verif
C2SAFERRUST [189]		Test,	Bootstrap LLMs + tests on c2rust
SYZYGY (ours)		Test,	LLMs + *dynamic* analysis (spec mining + test)

Table 9.1: Comparison of SYZYGY with other C-to-Rust translation approaches. Legend: = partial satisfaction; Cons = follows by construction; Verif = verification-based equivalence; Test = test-based equivalence validation.

Prior automated translators have each conceded on one axis of the resulting problem (Table 9.1). The canonical baseline c2rust [184] sidesteps the borrow checker by emitting `unsafe` Rust that mirrors C types through the FFI; the output compiles but retains C's memory-safety posture. CROWN [186] post-processes that output with rule-based ownership analysis

and still leaves unsafe residue. VERT [185] pairs an LLM with verification feedback against an MSWASM oracle but scales only to programs under two hundred lines, while Shiraishi *et al.* [187]'s LLM-plus-differential-testing system stops at compile-only feedback, with no per-function equivalence signal. Two concurrent systems do target safe Rust: SACTOR [188] couples the LLM with *static* analysis and verification feedback, and C2SAFERRUST [189] bootstraps an LLM on top of c2rust output. SYZYGY tackles these gaps by generating fully safe-Rust code, validating it with function-level equivalence tests, and guiding the translation using *dynamic* rather than static property mining. Existing rule-based systems often require restricted inputs or retain unsafe Rust when applied at scale; prior LLM-based systems have been evaluated mostly on smaller programs or lack per-function equivalence feedback.

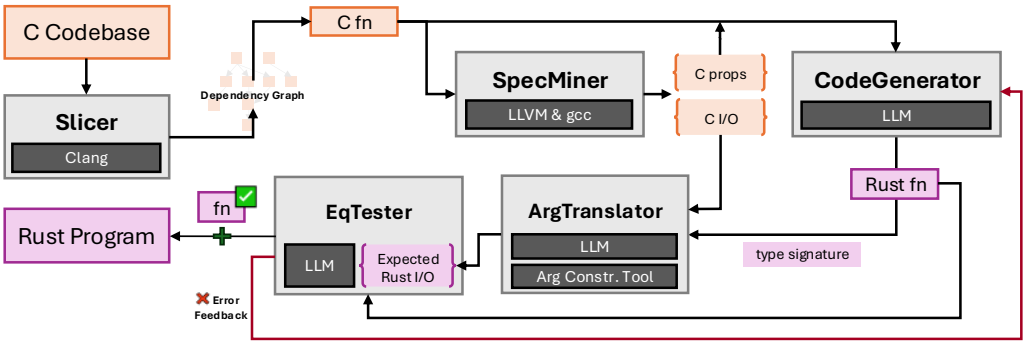


Figure 9.1: **Overview of our SYZYGY translation approach**: SLICER decomposes the codebase into translation units, CODEGENERATOR performs translation for that unit, and EQTESTER checks for equivalence of the generated Rust code with the original C code. ARGTRANSLATOR maps the C and Rust function arguments allowing appropriate equivalence checks in EQTESTER. SPECMINER, our dynamic analysis module mines (property and I/O) specifications for intermediate C functions using top-level tests. These specifications later assist our LLM-driven dual code-test translation.

Central tenets. SYZYGY decomposes (SLICER in Figure 9.1) large codebases into individual translation units. For each function, we generate a semantically equivalent Rust translation (CODEGENERATOR), then test it (ARGTRANSLATOR, EQTESTER) against its C counterpart, performing a multi-turn repair if necessary (see Figure 9.2).

SYZYGY is based on two main tenets:

LLMs + Dynamic Analysis. C constructs such as dynamic allocations of unknown sizes, aliasing between pointers, and pointer type casts (and the lack thereof in Rust) lead to challenging translation scenarios in which runtime information must be inferred from the code. Our dynamic analysis (SPECMINER) extracts this semantic information from test execution and makes it available to the LLM, thus aiding precise translation. Under the thesis framing, these extracted properties (TypeInfo, SizeInfo, NullInfo, AliasInfo) act as *structural*

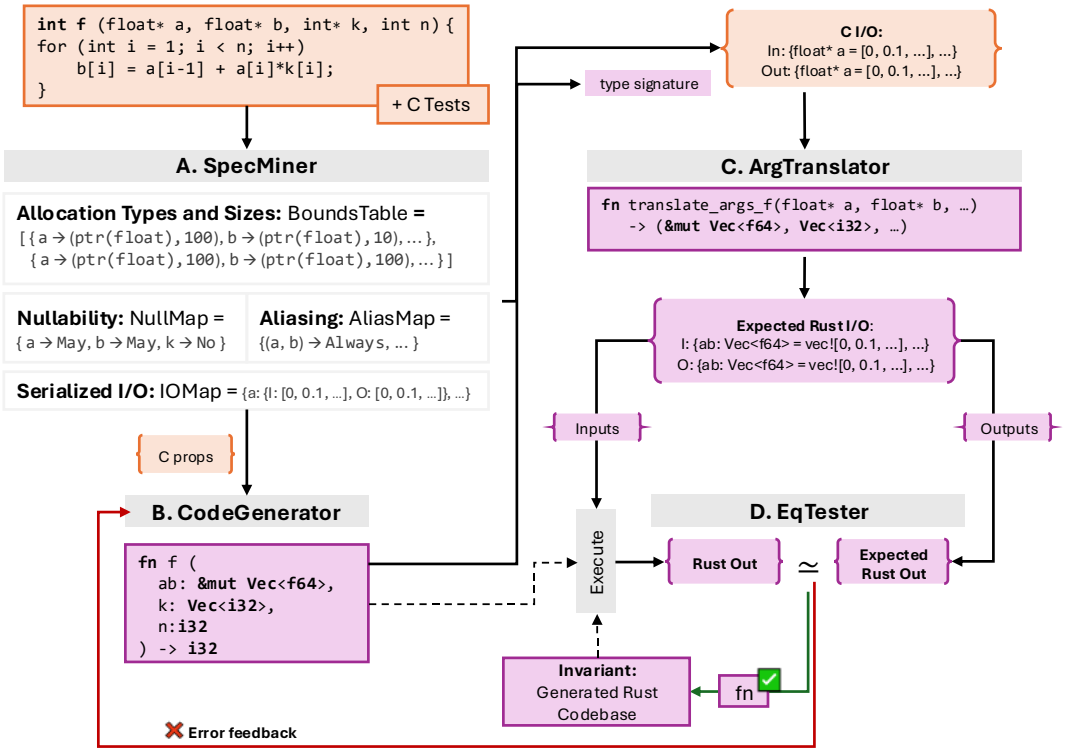


Figure 9.2: **Translation Pipeline for a C function** `f`: Given a C function and tests for the C codebase, first, the SpecMiner uses dynamic analysis to mine input-output (I/O) and property specifications for the translation. These properties (e.g., nullability and aliasing) guide a CodeGenerator to generate a candidate Rust translation of `f`. Given the translated signature and the collected C I/O, the ArgTranslator generates an translateArgs function that constructs aligned Rust inputs and expected outputs. Finally, the EqTester generates an equivalence test that executes the candidate translation, along with the previously generated Rust code, and checks whether the outputs match the expected values. If the equivalence checks pass, the translated function is committed to the Rust codebase; otherwise, error feedback is provided for multi-round repair.

dynamic-analysis annotations—a form of executable specification that captures empirically verifiable runtime facts.

Dual Code-Test Translation. To efficiently leverage the stochastic generative capabilities of LLMs for large codebases, we need *tests* to filter and/or repair incorrect candidates. Although tests for top-level functions are often available, testing intermediate functions is hard. We once again use our LLMs+Dynamic Analysis recipe: we use dynamic allocation analysis to mine I/O examples for intermediate functions from top-level C tests, and then use LLMs to programmatically map these I/O examples to Rust for equivalence testing. In the context of this dissertation, these serve as *per-step function-level equivalence tests* mined

from top-level runs.

Contributions. In summary, our contributions are as follows:

1. We develop an automated approach for **test-validated** translation from C to **safe** Rust. Our approach leverages the synergy between superior generative and search capabilities of LLMs and semantic execution information collected using dynamic analysis.
2. Our LLMs+Dynamic Analysis recipe performs **dual code and test generation**. Particularly, we enable reliable equivalence testing by (a) using dynamic analysis to mine intermediate I/O examples from top-level tests and (b) using LLMs to translate these examples to Rust.
3. We demonstrate the approach by translating Zopfli, a roughly 3 000 LoC high compression library. We validate our translation by performing top-level equivalence tests, achieving over 95% line coverage and 83% branch coverage on the source C code.

The remainder of the chapter is structured around these contributions. Section 9.2 sets up the C memory model, the safe-Rust type system, and the formal equivalence criterion against which we validate our translations. Section 9.3 develops the central insight that borrow-checker-relevant properties are dynamic, and cross-references the four property families with the Rust type decisions each drives. Section 9.4 describes the Syzygy architecture end-to-end. Section 9.9 reports the Zopfli main result and the ablations that isolate the contribution of dynamic-analysis specs and per-function equivalence tests.

9.2 Background

C and safe Rust share a surface syntax but commit to fundamentally different memory disciplines. The translation problem Syzygy solves is, at its core, about recovering properties that C leaves implicit and that Rust's type system insists on making explicit.

C Memory Model

C exposes memory as an unstructured address space manipulated through pointers. Programs allocate with `malloc` and release with `free`, cast between pointer types (including to and from the untyped `void*`), perform pointer arithmetic, and allow any number of pointers to *alias*, i.e., reference overlapping memory. Dynamic allocations, casts, and aliases challenge translation because they require non-local program reasoning; none of these uses are visible from the declared types alone.

Memory safety requires programs not to encounter runtime errors like out-of-bounds accesses (e.g., reading from beyond buffer bounds), use-after-free, and null pointer dereferences. One can exploit such behaviors to mount attacks such as reading memory to exfiltrate data (e.g., private keys), injecting data or code to perform arbitrary execution, and more [181]. C is memory-unsafe because it allows direct memory manipulation using pointers, burdening

the programmer with the task of avoiding such errors. Our dynamic analyses for allocation sizes, types, and aliasing help combat this by making these properties explicit.

Rust Memory Model

Rust addresses these issues through strong typing rules, strict compile-time checks, and an ownership model. Unlike C pointers, references in Rust have usage restrictions: they can be mutable (`&mut T`) or immutable (`&T`). Rust enforces *borrowing rules*—at any point there may be *either* one `&mut T` *or* any number of `&T`, but never both, and there cannot be any dangling references. This rule-based ownership model prevents dangling references and many memory-safety violations in safe code, though leaks through reference cycles can still require care.

These rules, however, make representing certain data structures (e.g., doubly linked lists) challenging. To allow more complex ownership and mutability scenarios, Rust provides smart pointers like `Rc<T>` and `RefCell<T>`. `Rc<T>` is a reference-counted (immutable) smart pointer that allows multiple owners, while `RefCell<T>` uses runtime checks to enable *interior mutability*, allowing data to be mutated even when there are immutable references. The composite `Rc<RefCell<T»` allows multiple owners to mutate shared data, which is useful for data structures like doubly-linked lists. Using smart pointers provides flexibility while maintaining safety, albeit with some runtime overhead and the potential for panics if misused.

Problem Formulation

We cast translation from C to safe Rust as program synthesis with observational equivalence [190] as the correctness criterion: *given as specification a C program, synthesize a safe Rust implementation with equivalent functional behavior.* Given a C program P_C $\mathcal{C}$ and a set of test inputs $T \subset \mathcal{I}$, we seek a Rust program P_R $\mathcal{R}_{\text{safe}}$ such that

$$t \quad T \quad P_C(t) \quad P_R(t)$$

where denotes equality of observable outputs.

In addition to generating equivalent Rust code, our main motivation for C-to-Rust translation is eliminating memory safety vulnerabilities. To do so, we disallow any `unsafe` blocks in the generated Rust program. The generated deliverable must compile under `##![forbid(unsafe_code)]`, meaning no `unsafe` blocks in the translated deliverable; FFI is restricted to temporary equivalence test and repair harnesses that call the original C functions through `bindgen` adapters. The translated program itself remains safe. The two main requirements—equivalence and memory safety—may conflict if input tests exercise unsafe behavior during execution. As a consequence of enforcing the `safe` fragment of Rust, we allow undefined behaviors (possibly runtime panics) in such cases. C programs may purposefully leverage unexploitable unsafety for performance or low-level control [191], but distinguishing between exploitable and unexploitable is extremely challenging, so we take the conservative approach and forbid all unsafety.

Finally, we assume three scope restrictions on the input C program. These are soft restrictions on the LLM code-generation pipeline, but our current dynamic analysis and testing infrastructure is limited in these cases:

1. **Acyclic data structures:** Data structures with pointer cycles require special care in Rust to avoid memory leaks (e.g., `Weak<T>` pointers to free disconnected memory cycles).
2. **No multithreading:** Concurrent access requires `Arc`-wrapped references to thread-safely perform borrowing.
3. **No type punning:** Performing raw memory accesses on untyped or multi-typed memory regions would hinder the best-effort type analysis that our approach performs.

9.3 Dynamic Specification Mining

A single C declaration `T*` carries almost no commitment about what the program does with the pointee at run time. Depending on whether the pointer addresses a singleton or a buffer, whether any call site passes `NULL`, whether the callee mutates the target, and whether two pointer arguments ever refer to the same cell, the same C type must be translated into any of a dozen distinct safe-Rust types — from `&T` and `&mut T`, to `Vec<T>` or `(Vec<T>, u32)`, to `Option<&T>`, `RefCell<T>`, or `Rc<RefCell<T>>`. Table 9.2 enumerates the four disambiguation axes that recur across real codebases; each row lists the candidate Rust types and the runtime property that discriminates them.

`T*` structure/usage scenario	Possible corresponding Rust type	Dyn. analysis
Array / iterator / singleton	`Vec<T>`, `[T;N]` `(Vec<T>, u32)` `&T`, `&mut T`	SizeInfo
Null pointers	`Option<&T>` `&T`, `&mut T`	NullInfo
Variable is/is not mutated	`&mut T`, `RefCell<T>`, `Rc<RefCell<T>>` `&T`	None (LLM-inferred)
Single / multiple references	`&T`, `&mut T` `Rc<T>`, `Arc<T>`	AliasInfo

Table 9.2: Scenarios in which a C type `T*` can be translated to different safe-Rust types depending on structure or usage, and the dynamic property that discriminates among the candidates. Grouped alternatives cannot be disambiguated by the listed analysis alone and are left to the LLM.

Pointers hide object structure. A `T*` may be a singleton, a fixed-size array, a heap-allocated buffer, or a moving iterator into one of the above. The Rust counterparts differ accordingly: `&T` versus `Vec<T>` versus the tuple `(Vec<T>, u32)` that pairs the owning buffer with an index. C types encode none of this. `void*` is worse still, erasing the element type `T` itself, so even the static skeleton of the Rust signature cannot be recovered without observing how the memory is used.

Pointers hide usage information. Aliasing, mutability, and nullability are all runtime properties. C permits arbitrary reads and writes through any pointer; safe Rust requires that we commit ahead of time to shared-immutable, exclusive-mutable, or run-time-checked interior mutability, and reject every construction that could allow two mutable references to coexist. The same `T*` may be a `&mut T` in one call site and must become an `Rc<RefCell<T>>` at another if callers ever share the pointee. No local syntactic cue tells the translator which case applies.

Challenge: Recovering C pointer information to identify corresponding Rust types
Solution: Supplement LLMs with dynamic analysis that collects semantic information about pointers and objects.

Using Dynamic Analysis for (partial) disambiguation. Our goal is to accurately determine which Rust type to use for a particular C pointer. While LLM-driven reasoning and code generation is powerful, accurately discriminating between these cases purely syntactically is challenging. Thus, we develop dynamic analyses (SpecMiner) to collect semantic information about pointers: aliasing, nullability, and allocation sizes. These analyses help discriminate between the target Rust types (Table 9.2) and thus guide the LLM during code and test generation.

Thesis Insight
LLMs produce more precise code when given *rich but incomplete* semantic specs than when given *imprecise but complete* specs. As documented in earlier chapters (Chapters 2 and 3), LLMs are trained primarily on static code tokens and are weak at predicting runtime behaviour. Dynamic analysis complements this weakness at exactly the places Rust's type system most needs it, while leaving the LLM to handle ambiguous cases not determined by the mined dynamic properties.

The four property types. Syzygy mines four families of dynamic properties from instrumented C executions; Section 9.5 specifies how each is computed, aggregated, and represented.

- TypeInfo : Args `T`, where `T` = `int` `uint` `char` `Struct` `ptr(T)` `fnptr`, recovers the underlying element type of each argument, resolving `void*` by following the `bitcast` chain.
- SizeInfo : Args $\mathbb{N}_{\geq 1}$ records the number of elements backing each pointer argument; singleton objects collapse to size one.
- NullInfo : Args NULL NONULL marks arguments observed to take the null value on any execution.
- AliasInfo : Args × Args Always Maybe Never classifies each pair of pointer arguments by whether they shared a backing address on every, some, or no execution.

Together with the per-function I/O traces that Syzygy also captures during instrumented runs, these four property types constitute the two kinds of executable specifications this chapter mines from runtime observation: *structural* specs that constrain the shape of the generated Rust signature, and *behavioural* specs that test each translated function for equivalence against its C counterpart. Section 9.4 shows how both streams feed the rest of the pipeline, and Section 9.5 unpacks the mining rules for each property.

9.4 System Architecture

Syzygy is realised as a five-module pipeline that converts a C codebase into safe Rust one function at a time, alternating between LLM generation and filters driven by runtime evidence. Figure 9.1 (shown earlier) illustrates the five modules and the data that flows between them, while Figure 9.2 zooms in on the per-function loop that repeats for each translation unit. This section glosses each module and the design principles that connect them, deferring detailed designs to the sections that follow.

Challenge: Scaling translation to multi-file/function codebases.

Solution: Decompose codebase into units and perform incremental aligned translation.

Slicer. The Slicer uses the Clang-17 frontend to parse the whole codebase and extract each top-level declaration—functions, structs, `typedef`s, macros—as a *translation unit*. A dependency graph is built over the units and topologically sorted. For function pointers, extra edges are added from the caller to every function whose signature matches. An iterator yields units in dependency order from the leaves up. For each unit, the Slicer exposes the *dependency slice*—the transitive closure of the unit's dependencies—which provides the context the CodeGenerator needs.

SpecMiner. The SpecMiner operates before any translation begins. An LLVM-14 instrumentation pass inserts handlers into internal functions to serialize arguments and return values to JSON. Running the instrumented binary under the program's top-level tests yields a stream of $(\mathsf{I}_\mathrm{C}\ \mathsf{O}_\mathrm{C})$ pairs together with the per-argument dynamic property maps (TypeInfo, SizeInfo, NullInfo, AliasInfo). Section 9.5 develops the capture and mining rules.

CodeGenerator. The CodeGenerator performs slicing-plus-greedy translation in dependency order. The prompt for each unit bundles the C source, its dependency slice, the already-committed Rust translations of those dependencies, and the SpecMiner property annotations. The model samples candidate translations, and a compile filter retains those that type-check under `#![forbid(unsafe_code)]`. Compile errors are fed back for a bounded number of repair rounds.

Challenge: Ensuring high-coverage equivalence tests for *all* internal functions.
Solution: (A) Capture inputs and outputs to internal C functions from top-level invocations, and (B) programmatically translate them to Rust to get equivalence tests.

ArgTranslator. The ArgTranslator produces an LLM-written translateArgs function that rehydrates the captured C arguments as the Rust objects the candidate expects. The model composes primitives from a bespoke Argument Construction Rust library, which the model calls as a tool. Section 9.6 details the API and the symbol table that preserves aliasing.

EqTester. The EqTester emits a short Rust harness that loads the pre- and post-state records from the SpecMiner JSON, runs translateArgs on both, invokes the candidate on the input, and asserts equality against the expected output. Value-based equivalence relies on recursively comparing derived `PartialEq` structures. Section 9.7 shows the generated harness and details the rejection-sampling loop.

Challenge: Successfully generating at least one correct translation for each unit.
Solution: Inference-time scaling (sample candidates) and equivalence-test-based filtering.

Design principles. Two principles run through the architecture. *Incremental Aligned Translation* (IAT) decomposes the codebase and translates one unit at a time, keeping Rust function signatures and the call graph aligned with their C counterparts. Focusing the LLM on a single function improves generation accuracy. *Rejection sampling* converts the unreliability of individual LLM generations into a filter pipeline: CodeGenerator produces N candidates filtered by compilation; ArgTranslator produces K translateArgs samples filtered by execution; EqTester produces K equivalence harnesses filtered by execution. When a stage has no survivors, the feedback is a *diff* against the expected behaviour rather than the full output.

9.5 SpecMiner: Dynamic Specification Mining

Syzygy uses dynamic analysis to identify semantic information about C objects. The analysis operates in two steps. First, we use code instrumentation to capture and dump (serialized) inputs and outputs of all functions invoked during execution. Then, this captured I/O is used to mine *likely* properties of the execution, such as types, bounds, and aliasing. We illustrate I/O capture and dynamic analysis using Figure 9.3 as a running example.

Capturing C I/O Objects

We capture inputs and outputs of each function in the C codebase by invoking it from the top-level (e.g.,) entry points. This guarantees that the captured inputs are valid and satisfy

```c
float kernel1(float a, float b, int k) { return b - a*k; }

float* f(float* arr, float* brr, int* krr, int num, float (*kernel)(float,float,int)) {
    for (int i = 1; i < num; i++) brr[i] = (*kernel)(arr[i-1], arr[i], krr[i]);
    return brr;
}

float* caller1() {
    float* arr = malloc(3*sizeof(float));
    float* brr = malloc(2*sizeof(float));
    int* krr = malloc(3*sizeof(int));
    // ... populate arr, brr, krr ...
    return f(arr, brr, krr, 2, kernel1);
}

float* caller2() {
    float* arr = malloc(3*sizeof(float));
    int* krr = malloc(3*sizeof(int));
    // ... populate arr, krr ...
    return f(arr, arr, krr, 3, kernel1);
}

int main() {
    float* res;
    if (...) res = caller1();
    else     res = caller2();
}
```

Figure 9.3: **Dynamic Specification Mining**: An example C code snippet which features dynamic allocations (in `caller1` and `caller2`), function pointers (in `f`), and aliasing between arguments (in the `main` – `caller2` – `f` call chain). While this information can greatly assist code and test generation, inferring it statically is challenging. We use a combination of dynamic analyses to collect this information.

complex pre-conditions.

We instrument the entry and exit points of each function with handlers that dump serialized input and output values. Non-pointer objects are serialized directly. For pointers, we track allocation sizes and types using a runtime bounds table based on existing shadow-memory approaches [192, 193]. We use this allocation tracking to dereference the pointer and serialize the internal object. At the exit point, we dump both the arguments and return values to capture side effects. A single invocation of f emits an input-output test example: $(\mathsf{I}_C[f]\ \mathsf{O}_C[f])$. Overall, this process is summarized as:

$$\mathsf{I}_C[\ ] \xrightarrow{\text{instrument + execute + serialize}} (\mathsf{I}_C[f]\ \mathsf{O}_C[f])_{\ f}$$

For example, for the function `f` in Figure 9.3, we would capture the following I/O example

(represented as a serialized JSON object) for the `main–caller2–f` call chain:

$$\begin{aligned}
\mathsf{I}_C[\mathtt{f}] = \quad & \mathtt{arr} : \mathtt{addr} : 0x1234 \ \ \mathtt{size} : 3 \ \ \mathtt{data} : [100\ 0\ \ 200\ 0\ \ 300\ 0] \\
& \mathtt{brr} : \mathtt{addr} : 0x1234 \ \ \mathtt{size} : 3 \ \ \mathtt{data} : [100\ 0\ \ 200\ 0\ \ 300\ 0] \\
& \mathtt{krr} : \mathtt{addr} : 0x9abc \ \ \mathtt{size} : 3 \ \ \mathtt{data} : [1\ \ 2\ \ 3] \\
& \mathtt{num} : 3 \ \ \mathtt{kernel} : \ \ \mathtt{isfun} : \mathtt{true} \ \ \ \mathtt{name} : \mathtt{kernel1} \\
\mathsf{O}_C[\mathtt{f}] = \quad & \mathtt{arr} : \mathtt{addr} : 0x1234 \ \ \mathtt{size} : 3 \ \ \mathtt{data} : [100\ 0\ \ 0\ 0\ \ 300\ 0] \\
& \mathtt{brr} : \text{same as } \mathtt{arr} \quad \mathtt{krr} : \text{same as } \mathsf{I}_C[\mathtt{f}][\ \mathtt{krr}\ ]
\end{aligned}$$

Note how the matching `addr` values indicate aliasing between `arr` and `brr` in this execution. Since `f` does not have a return value, $\mathsf{O}_C[\mathtt{f}]$ only contains the modified arguments.

Mining Specifications from C Executions

Next, we mine likely properties of executions from the captured I/O examples. Depending on the property, we either aggregate properties across multiple executions (i.e., (I_C O_C) pairs) or maintain information *per execution*.

Types. The TypeInfo map provides the type of the underlying object for each argument:

$$\mathsf{TypeInfo} : \mathsf{Args} \quad \mathtt{T} \quad \text{where } \mathtt{T} = \mathtt{int} \ \ \mathtt{uint} \ \ \mathtt{char} \ \cdots \ \text{Structs} \ \ \mathsf{ptr}(\mathtt{T}) \ \ \text{fnptr}$$

For function `f` (Figure 9.3), we would extract the following type mapping: TypeInfo = `arr brr ptr(float) krr ptr(int) num int kernel fnptr` .

We recover this information using the LLVM runtime. Since `malloc` calls return an untyped `void*`, we perform a forward SSA-dependency pass starting from `malloc` until we hit the first `bitcast` operation. For example, for `float* arr = malloc(size*sizeof(float));`, we observe a `bitcast` from `i8*` to `f32*`. We employ a conservative approach, mapping the pointer to `void*` if unable to find a `bitcast`.

Allocation sizes. The SizeInfo map captures argument sizes dynamically:

$$\mathsf{SizeInfo} : \mathsf{Args} \quad \mathbb{N}^{\geq 1}$$

For function `f`, in the `caller1` path, we get SizeInfo = `arr 3 brr 2` while in the `caller2` path, we get SizeInfo = `arr 3 krr 3` .

Nullability. The NullInfo map captures whether an argument is *ever* `NULL`, in any execution. It performs aggregation across multiple executions:

$$\mathsf{NullInfo} : \mathsf{Args} \quad \mathtt{NULL}\ \mathtt{NONULL}$$

For function `f`, since the `arr` argument is never `NULL` (for both `caller1` and `caller2`), we have NullInfo = `arr NONULL` . If it were null, this information would direct whether a Rust type should be wrapped in the `Option` tag.

Aliasing. Aliasing information is captured in the AliasInfo map:

$$\mathsf{AliasInfo} : \mathsf{Args} \times \mathsf{Args} \qquad \texttt{Always Maybe Never}$$

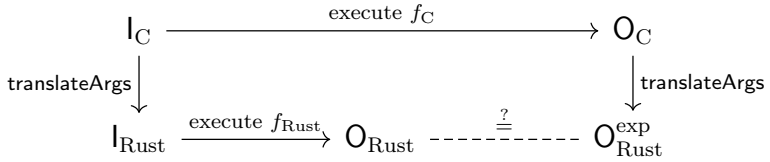
Similar to NullInfo, AliasInfo also aggregates across executions to determine whether arguments always/sometimes/never alias, implemented by comparing `addr` fields. For function `f`, since the `caller2` path does alias between `arr` and `brr` (but `caller1` does not), aggregation gives:

$$\mathsf{AliasInfo} = \quad (\texttt{arr brr}) \quad \texttt{Maybe}\ (\texttt{arr krr}) \quad \texttt{Never}\ (\texttt{brr krr}) \quad \texttt{Never}$$

Given Rust's strict borrowing rules, identifying aliasing between function arguments aids in determining mapping Rust types.

9.6 ArgTranslator

With SpecMiner supplying the captured I/O stream $(\mathsf{I}_\mathrm{C}[f]\ \mathsf{O}_\mathrm{C}[f])$ and the four property maps, the next question is how to turn that stream into a test that can actually execute a candidate Rust translation f_Rust and compare its behaviour against f_C. The link between the two sides is a generated translation function $\mathsf{translateArgs} : (\mathsf{I}_\mathrm{C}\ \mathsf{O}_\mathrm{C}) \quad (\mathsf{I}_\mathrm{Rust}\ \mathsf{O}^{\mathrm{exp}}_\mathrm{Rust})$ that maps the C I/O to corresponding Rust I/O. Equivalence can then be checked by executing f_Rust and comparing its output to the expected Rust output. This closes the commutative square:



The Argument-Translation Problem

Generating translateArgs is challenging. A purely rule-based translation is infeasible because the generated I_Rust input must be compatible with the signature of f_Rust. As seen in Table 9.2, the same C type maps to different Rust types depending on structure and usage. This context-dependent type alignment makes hardcoded rule-based $\mathsf{I}_\mathrm{C} \quad \mathsf{I}_\mathrm{Rust}$ translation infeasible.

Pure LLM-generation of translateArgs also fails to scale robustly. A naive approach would directly feed the serialized I_C into the LLM and ask it to generate I_Rust. However, the LLM would have to copy array elements one-by-one, which is error-prone for large structures (> 1000 elements) where LLMs struggle with long outputs. Additionally, accurate object reconstruction requires parsing size and aliasing from the serialized I_C. This makes purely freehand LLM generation of translateArgs brittle.

Our solution enables programmatic translation by assisting the LLM through a translation API that consumes I_C and exposes it through high-level primitives.

Challenge: Generate accurate translateArgs that adapts to argument types/semantics.

Solution: Programmatic translation using an Argument Construction API that the LLM *uses as a tool* to generate translateArgs.

The Argument Construction API

The Argument Construction API (Table 9.3) translates I_C into *auxiliary* Rust objects that adhere to size and aliasing relations. Size and contents can be directly replicated by parsing the serialized I_C. To replicate aliasing, the API maintains a symbol table that maps C pointers to *smart pointers* (`Rc<RefCell<RT>>`) to Rust objects. If two C pointers (`T* a, b`) reference the same object (as captured in I_C), the API will (a) create a fresh Rust object for `a`, (b) update the symbol table to point `b` to the same object, and (c) return two `Rc<RefCell<RT>>` references to it.

Function	(input) C Type	(output) Auxiliary Rust Type
translate_prim_single	int, char, ...	u32, i8, ...
translate_prim_vec	int*, char*, ...	Vec<u32>, Vec<i8>, ...
translate_prim_vec_rc_refcell	int*, char*, ...	Rc<RefCell<Vec<u32>>>, ...
translate_string	char*	String
translate_struct	struct T*	Rc<RefCell<Struct>>
translate_1d	T*	Vec<Rc<RefCell<RT>>
translate_2d	T**	Vec<Vec<Rc<RefCell<RT>>>

Table 9.3: Argument Construction API functions: primitives to translate singletons, arrays, strings, and structs. For n-D arrays, we provide functions to translate them into nested `Vec` objects. We preserve aliasing matching C using `Rc<RefCell<RT>>`, where `RT` is the Rust type that the C type `T` maps to.

These functions are exposed as tools to the LLM. The LLM writes translateArgs using these functions, and then further coerces the auxiliary Rust objects into final objects that match the candidate function signature.

Worked Example

Figure 9.4 shows an example translateArgs handling three possible target Rust argument types. The left side shows the tedious boilerplate required to accurately replicate array sizes, contents, and aliasing from I_C, which is what a pure-LLM approach must generate. By contrast, the right side shows equivalent translateArgs function bodies that use our Argument Construction API. The LLM can either use the API functions directly when the auxiliary objects match the target signature (cases ①, ②), or write short Rust code (case ③) to type-coerce the auxiliary objects.

```rust
// Maintain symbol tables for tracking object aliasing

// Rust Symbol Table is populated as we translate arguments
pub static ref rst: Mutex<SymbolTable> = Mutex::new(HashMap::new());

pub unsafe fn translateArgs_f(arr: *mut c_int, brr: *mut c_int, ...) ->
        // Possible target argument types for f_R translation
      ①// (Vec<i32>, Rc<RefCell<Vec<i32>>, ... )
      ②// (Rc<RefCell<Vec<i32>>, Rc<RefCell<Vec<i32>>, ... )
      ③// (Vec<i32>, &mut [i32; 3], ... )
{
    // Check if (* c_int) arr is already translated to maintain aliasing
    let rust_arr = if let Some(arr) = rst.get(arr as *mut c_void) {
        arr as *mut c_void
    } else {
        // Copy individual c_int from arr to rust_arr
        // Build Vec<i32> from (* c_int) arr
        let len = // ... get length from I_C
        let mut v = Vec::with_capacity(len);
        for i in 0..len {
            v.push(*arr.offset(i as isize) as i32);
        }
        // Insert arr into rst  to maintain aliasing
        ...
    };

    // same procedure for (* c_int) brr
    let rust_brr = if let Some(brr) =
        rst.get(brr as *mut c_void) {
            brr as *mut c_void
        } else {
            // Build Vec<i32> from (* c_int) brr
            ...
        };

    // Based on case 1/2/3, convert rust_arr,rust_brr (Rc<RefCell<Vec<i32>>>) to match f_R
    (rust_arr, rust_brr, ...)
}
```

①

```rust
// Deepcopy Vec from Rc<RefCell<Vec>>
let rust_arr = translate_prim_vec<i32>(arr);
// Return Rc<RefCell<Vec<T>>> as is
let rust_brr = translate_prim_vec_rc_refcell<i32>(brr);
```

②

```rust
let rust_arr = translate_prim_vec_rc_refcell<i32>(arr);
let rust_brr = translate_prim_vec_rc_refcell<i32>(brr);
```

③

```rust
let rust_arr = translate_prim_vec<i32>(arr);
let rust_brr_vec = translate_prim_vec<i32>(brr);
// Convert into [i32; 3] ...
let rust_brr: &mut [i32; 3] = rust_brr_vec
    .as_mut_slice().try_into().expect("Wrong length!");
```

Figure 9.4: Example of a translateArgs function with three possible target Rust argument types. **Left.** Boilerplate required to accurately replicate array sizes, contents, and aliasing from I_C. This involves building Rust `Vec`s and using a symbol table to manage aliasing. **Right.** The equivalent translateArgs bodies using our Argument Construction API, which performs these tasks internally. High-level primitives make the translateArgs concise and easy for an LLM to write.

Each translateArgs is LLM-generated and executed to filter invalid samples before EqTester asserts observational equivalence.

9.7 Equivalence Testing and Iterative Refinement

With a valid translateArgs in hand, the EqTester closes the verification loop, and the surrounding pipeline wraps every LLM-driven stage in a rejection-sampling filter with diff-based feedback.

Equivalence Testing

The EqTester prompts an LLM to emit a value-based equivalence test harness `eqtest_f(filename: &str) -> bool`. As shown in Figure 9.5 for the `caps` function, the generated test (i) loads (I_C O_C) from the SpecMiner trace at `filename`, (ii) runs translateArgs on both sides to obtain the Rust input and expected Rust post-state, (iii) executes f_{Rust} on the translated input, and (iv) asserts value equivalence against the expected state.

Value equivalence relies on deriving `PartialEq` on every translated type so that nested structs, `Vec`s, and `Rc<RefCell<_>>` trees compare recursively; floating-point fields relax the check to $x - y <$. When side-by-side C execution is needed directly in the harness, `bindgen` exposes the C symbol over FFI so both implementations can run against the same inputs.

```
pub unsafe fn eqtest_caps(filename: &str) -> bool {
    let c_pre = load_pre_args_from_json(filename);
    let mut rust_string = translateArgs_caps(c_pre);

    let c_post = load_post_args_from_json(filename);
    let expected = translateArgs_caps(c_post);

    caps(&mut rust_string);

    assert_eq!(rust_string, expected, "Rust string does not match expected");
    true
}
```

Figure 9.5: Generated equivalence test for the `caps` function.

Iterative Refinement

Every LLM-invoking stage is a rejection-sampled filter, composed as Figure 9.6 illustrates. The CodeGenerator draws N candidate Rust translations of f_C; compilation filters them to $N \leq N$. Each survivor is paired with K sampled translateArgs functions, yielding $M = N \times K$ pairs; executing translateArgs on the serialized C inputs filters these to M . The EqTester then samples K equivalence tests per pair, producing $S = M \times K$ triples ($f_{\text{Rust}\,i}$ t_{ij} eq_{ijk}); a final execution pass filters these to S triples whose behaviour matches every captured C trace.

When a filter empties its candidate set, the pipeline re-prompts the LLM with the failing assertion's message and a `diff` between observed and expected outputs; raw post-states would otherwise make the feedback less focused.

A worked example. Consider an anecdotal example from our Zopfli case study. The `BoundaryPMFinal` function assigns `pool->next` to a freshly allocated child (Figure 9.7, left). In the first round, the LLM interpreted `node->next` as an iterator advancement and emitted

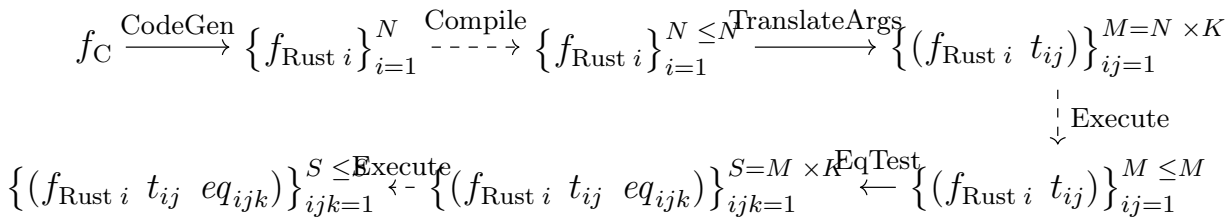


Figure 9.6: Rejection-sampling loop across the three LLM-invoking stages. Solid arrows denote generation; dashed arrows denote verification filters.

a clone followed by `pool.index += 1`, mutating the pool's cursor on every call (Figure 9.7, right). The EqTester caught the divergence because the captured pool state still held the original index after the C call; the diff fed back to the LLM flagged the field, and the next sample removed the spurious increment.

```c
if (lcount < nsymb && sum > leaves[lcount].
    weight) {
  Node *newch = pool->next;
  // ^^^^^^^^^^^^^^^^^^^^^^ this line
  Node *oldch = lists[index][1]->tail;
  ...
}
```

```rust
let newch = pool
  .next.get(pool.index)
  .expect("Pool has no more nodes")
  .clone();
// Move to next free node in the pool.
pool.index += 1;
// ^^^^^^^^^^^^ this line
```

Figure 9.7: **Execution feedback enables repair.** The C function `BoundaryPMFinal` grabs the `next` field from a `pool` of nodes (left). In the first round, the LLM incorrectly interprets `node->next` as advancing an iterator and increments the index (right). The EqTester catches this bug using deserialized objects, and the resulting diff allows the LLM to repair the code.

Post-translation Validation and Repair

Per-function equivalence does not guarantee whole-program equivalence on inputs never exercised initially. Once the pipeline completes, we validate the complete Rust codebase with an equivalence test suite from the top-level entry point. When an input fails, we reuse SpecMiner to trace the failing path and localize the divergence to the innermost offending function. Our incremental strategy allows translation to recover by restarting from the failing function, prompting the LLM to fix it with the new feedback. Because corrections to an erroneous function may propagate to its dependencies, cascading repairs are possible. To achieve minimal and targeted refinements, we manually determine the scope of this repair process, leaving more systematic techniques for future work.

9.8 Implementation

We now detail the concrete implementation of the SYZYGY modules, following the structure from the system's development.

SLICER

The program slicer uses the CLANG-17 frontend to consume the entire C codebase, extract the top-level statements as translation units, and build a topologically sorted dependency graph. We handle function pointers by adding extra dependency edges between function types and the functions matching those types. This graph allows us to iterate over translation units in dependency order and determine each unit's dependency slice.

SPECMINER

The SPECMINER uses an LLVM-14-based instrumentation pass to populate a runtime bounds table with allocation types and sizes. (We target LLVM-14 because opaque pointers in later LLVM versions discard the `bitcast` information needed for type recovery.) Stack variable types are obtained directly from the LLVM IR. For heap allocations, we perform a forward SSA-dependency pass starting from `malloc` until we hit the first `bitcast` operation, falling back to `void*` if none is found. The bounds table records all allocations in an interval tree, allowing conversion between array length and allocation size. This table is accessed to serialize the C I/O pairs (I_C O_C) and evaluate the property annotations.

CODEGENERATOR

For each translation unit, we prompt the LLM with: (a) all its dependencies, (b) the corresponding already-translated Rust elements, and (c) the dynamic analysis annotations (NullInfo, AliasInfo, etc.). As detailed in the system's prompt appendices, the generator is steered by rules to ensure compatibility: e.g., `int*` maps to `Vec<i32>` by default, C function pointers map to `Fn`/`FnMut` closures, and `void*` maps to `Box<dyn std::any::Any>`. We sample multiple candidate translations and use the Rust compiler with `##![forbid(unsafe_code)]` to identify safe compiling translations, prompting the LLM with compile-error feedback for a bounded number of repair rounds.

Test-generation based on ARGTRANSLATOR

We use LLMs with our Argument Construction API to generate the Rust translateArgs functions. The API primitives, alongside the process-wide symbol table, handle the boilerplate of size and aliasing replication. We filter candidate translateArgs functions by their ability to successfully load (I_C O_C) from the serialized JSON and map them to appropriately typed Rust objects.

EqTester

The EqTester module uses the generated translateArgs in a Rust equivalence testing harness. The harness performs value-based equivalence checks by adding the `PartialEq` trait to Rust types, leveraging the built-in equality operator for nested comparisons. For a given C function and its candidate Rust function, the generated test compares outputs. Since the harness must also call into the C function (e.g., during side-by-side localization for repair), we use `bindgen` to export all C functions in the codebase to Rust through the FFI. Equivalence testing serves as the final filter for candidates, providing error feedback for multi-round repair if none pass.

9.9 Evaluation

We evaluate Syzygy by translating Zopfli [182], Google's deflate/LZ77/Huffman compression library. Zopfli is the primary case study for this approach: it is the largest C codebase for which any LLM-driven translator has, to our knowledge, produced a fully test-validated Rust translation.

Setup

Experiments run on a 96-core Intel Xeon machine with 720 GB of RAM. Unless otherwise noted, every LLM-invoking stage of Syzygy — CodeGenerator, ArgTranslator, EqTester, and the rejection-sampling repair loop — uses o1-preview for generation and o1-mini for cheaper repair drafts. All Rust output compiles cleanly under `#![forbid(unsafe_code)]` at the crate root.

Zopfli Case Study

Zopfli consists of over 3,000 lines of C (excluding comments), comprising 98 functions and 10 structs, spanning 21 files. It provides a challenging testbed due to the diversity of C constructs, including heap-allocated data structures, linked lists, iterators, function pointers, and `void*` arguments. It also respects the scope assumptions of Section 9.2: no cyclic data structures, no multithreading, and no type punning.

Methodology. We first run the `unifdef` preprocessor [194] on the codebase to standardize the `#ifdef` configuration options for OS and compiler features, giving Syzygy a single canonical source. We then invoke the pipeline with `ZopfliDeflate` — the top-level entry point that consumes an input string and a `ZopfliOptions` configuration struct — as the root of the dependency graph. We semi-automatically collect 26 top-level test inputs for `ZopfliDeflate`; these achieve 88% line coverage and 70% branch coverage on the original C program.

We run SpecMiner to collect I/O and property specifications for every function using these tests. Next, we run our incremental translation pipeline to generate Rust translations in dependency order. Since we do not have equivalence tests for non-function units such as structs, global assignments, and macros, we manually verify their translations. During this manual step, we discovered and repaired a corner-case bug in the LLM's translation of the `ZOPFLI_APPEND_DATA` macro.

Coverage and cost. The pipeline translated all 98 functions, producing roughly 4,500 lines of substantive Rust code (over 7,000 lines including generated docstrings and verbose comments). End-to-end wall-clock time was approximately 15 hours, with an LLM-API bill of about $2 500; we believe the cost could be optimized closer to $1 000 with better hyperparameters.

Pass rates. Each function translation is the output of a multi-stage filter: CodeGenerator produces N candidates, a compile filter retains those that type-check, and EqTester retains those that pass the value-based equivalence harness. Figure 9.8 shows the per-function pass rates across all 98 functions. We observe that the mean execution pass rates are high, ranging over 75%, implying that Syzygy frequently generates high-quality translations. However, the distribution has a long tail of challenging functions with low pass rates (below 20%). This tail highlights the need for equivalence tests, as compile-only filtering cannot separate these failing functions from correct ones.

Equivalence validation suite. To check the correctness of our translation beyond the 26 inputs used during translation, we assembled a validation suite of 1000000 input strings for `ZopfliDeflate` ranging in length from 10^1 to 10^7 characters, totalling roughly 20 GB. These validation inputs achieve 95% line coverage and 83% branch coverage on the C source. We note that a portion of the code (5 9% of branches) is supposed to be unreachable (e.g., `assert` statements, `exit` branches). Thus, while testing is fundamentally prone to incomplete coverage, these high coverage tests provide a strong validation of correctness.

Runtime performance. Replacing tuned C with safe Rust incurs some performance overhead. We ran both programs on two test suites: six strings of repeated `'a'` between 10^1 and 10^6 characters, and 12 random strings in the same length range. We compiled the programs with default (`gcc -O0 -g` and `cargo debug`) and optimized (`gcc -O3` and `cargo build -release`) configurations. As Table 9.4 shows, the translated code is slower, with larger slowdowns in the default configuration. A manual comparison of flamegraphs indicates `ZopfliUpdateHash` — a function that operates on large array data structures, copying `Vec` allocations and performing bounds checks — is a large contributor to this slowdown.

LLM shortcomings and post-generation repair. While some C code accompanies array pointers with explicit length variables, the model often emits Rust code that instead

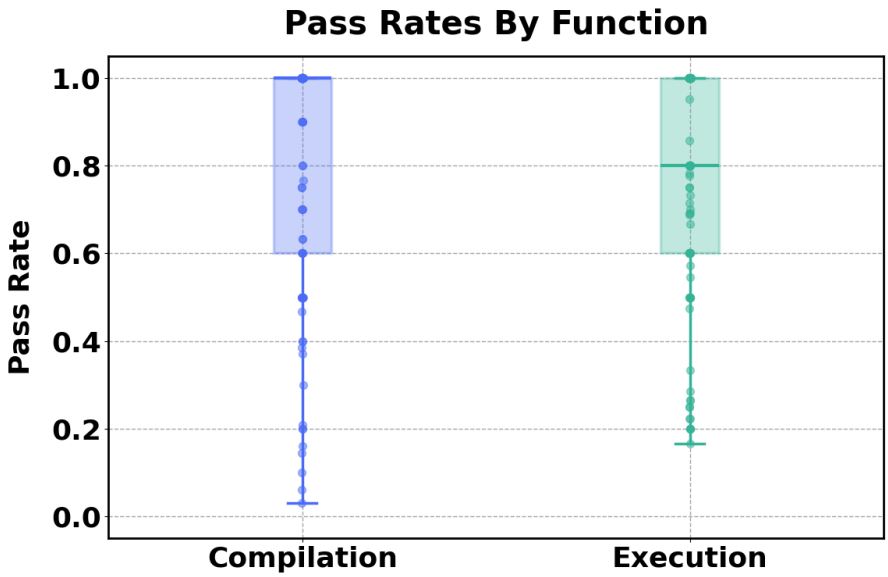


Figure 9.8: **Box plot demonstrating pass rates for code compilation and test execution.** We measure the per-function pass rates across the 98 ZOPFLI functions. For test execution, we use the compiling solutions as the denominator.

Table 9.4: Performance comparison of C and Rust implementations on two input families, under default and optimized compilation.

	Default			Optimized		
Input Type	C	Rust	Slowdown	C (−O3)	Rust (−−release)	Slowdown
Repeated Input	0 170	1 56	8 9×	0 040	0 059	1 47×
Random Input	0 242	3 34	13 8×	0 094	0 330	3 67×

uses `Vec`'s `len()` method. However, LLMs are inconsistent in this behavior, frequently making mistakes that break equivalence when callers pass explicit lengths. Another challenge involves C `for` loops that modify their iterator inside the body; since Rust does not allow this, the translation must convert the loop to a `while` loop and carefully increment the iterator at all `continue` statements, which the model occasionally misses.

Despite guiding translation with the 26 tests, we observed runtime exceptions on a fraction of our larger validation suite. As discussed in Section 9.7, we used SPECMINER to localize the error to the innermost failing function (`zopfli_block_split_lz77`) and restarted the pipeline from that point. With the new feedback, the subsequent repair run fixed the bug via a simple local change. This highlights that greedy incremental translation is sensitive to the

quality of the equivalence tests used to guide the translation; a low-coverage set may require backtracking or post-generation repair.

Ablations

Two ablations test which parts of the pipeline are structural versus incidental.

RQ 2.3: Does choice of LLM matter? To evaluate the impact of the choice of LLM, we experiment with the cheaper GPT-4o model. We observe that GPT-4o did not succeed in our multi-turn execution-feedback-based repair setting. Hence, we fall back to o1 for the repair loops. With the significantly cheaper GPT-4o model driving the initial generation, we can generate a compiling translation for less than $800. However, this translation is up to 20× slower than the original C implementation and crashes on some long test inputs. The o1 model was necessary in our experiments for generating valid translations, though future work might optimize cost by combining the two models.

RQ 2.4: Do incremental equivalence tests aid translation? As an ablation, we remove the testing module to generate code without any intermediate filtering except compiler checks. Using this approach, we successfully generate a compiling version of Zopfli for under $100. However, we find that the generated code does not run even for small inputs such as `"Hello, World"`, crashing with an `index out of bounds` error. This aligns with our findings from Figure 9.8, where compile-only filtering leaves a large execution-pass-rate drop for hard functions. Incremental testing was necessary for generating correct translations in this experiment.

9.10 Worked Example: `GetCostModelMinCost`

We close the technical exposition by analyzing one Zopfli function through the full process. The `GetCostModelMinCost` routine combines several constructs flagged as hard in prior sections: an indirect call through a `CostModelFun` pointer, a `void*` context whose underlying type is callee-dependent, and a floating-point return value requiring numerical-tolerance equality.

The C Source and Rust Candidate

Figure 9.9 reproduces the C definition, which scans distance symbols and returns the minimum cost of any valid (length, distance) pair. The inner call `costmodel(i, 1, costcontext)` is indirect: `costmodel` may bind either to `GetCostFixed`, which ignores `costcontext`, or to `GetCostStat`, which casts `costcontext` back to a `SymbolStats*`. No static C type carries this discipline.

Figure 9.10 shows the CodeGenerator's candidate. The loops preserve the C structure; only the signature is substantive, swapping the raw function pointer and `void*` for safe trait objects. The `&dyn Fn`/`&dyn Any` pair matches the pattern identified by the SpecMiner properties.

```
static double GetCostModelMinCost(CostModelFun *costmodel, void *costcontext)
{
  double mincost;
  int bestlength = 0; /* length that has lowest cost in the cost model */
  int bestdist = 0;   /* distance that has lowest cost in the cost model */
  int i;
  /*
  Table of distances that have a different distance symbol in the deflate
  specification. Each value is the first distance that has a new symbol. Only
  different symbols affect the cost model so only these need to be checked.
  See RFC 1951 section 3.2.5. Compressed blocks (length and distance codes).
  */
  static const int dsymbols[30] = {
      1, 2, 3, 4, 5, 7, 9, 13, 17, 25, 33, 49, 65, 97, 129, 193, 257, 385, 513,
      769, 1025, 1537, 2049, 3073, 4097, 6145, 8193, 12289, 16385, 24577};

  mincost = ZOPFLI_LARGE_FLOAT;
  for (i = 3; i < 259; i++) {
    double c = costmodel(i, 1, costcontext);
    if (c < mincost) { bestlength = i; mincost = c; }
  }

  mincost = ZOPFLI_LARGE_FLOAT;
  for (i = 0; i < 30; i++) {
    double c = costmodel(3, dsymbols[i], costcontext);
    if (c < mincost) { bestdist = dsymbols[i]; mincost = c; }
  }

  return costmodel(bestlength, bestdist, costcontext);
}
```

Figure 9.9: **The `GetCostModelMinCost` function from Zopfli.** Complexity arises from the indirect call through `costmodel` coupled with the `void*` payload `costcontext`, neither of which admits a sound Rust type from static information alone.

translateArgs and EqTester

Because the dynamic type of `costcontext` depends on which callee `costmodel` resolves to, translateArgs dispatches on the symbol name recovered from the raw function-pointer address (Figure 9.11). For `GetCostFixed` it installs a context-free closure and an empty-tuple `Box<dyn Any>`; for `GetCostStat` it translates the `SymbolStats` payload through the Argument Construction API and synthesizes a closure that `downcast_ref`-es the context at every invocation.

```rust
pub fn get_cost_model_min_cost(
    costmodel: &dyn Fn(u32, u32, &dyn std::any::Any) -> f64,
    costcontext: &dyn std::any::Any,
) -> f64 {
    let mut min_cost: f64;
    let mut best_length: u32 = 0;
    let mut best_dist: u32 = 0;

    let dsymbols: [u32; 30] = [
        1, 2, 3, 4, 5, 7, 9, 13, 17, 25, 33, 49, 65, 97, 129, 193, 257, 385,
        513, 769, 1025, 1537, 2049, 3073, 4097, 6145, 8193, 12289, 16385, 24577,
    ];

    min_cost = ZOPFLI_LARGE_FLOAT;
    for i in 3..259 {
        let c = costmodel(i as u32, 1, costcontext);
        if c < min_cost { best_length = i as u32; min_cost = c; }
    }

    min_cost = ZOPFLI_LARGE_FLOAT;
    for &dist in dsymbols.iter() {
        let c = costmodel(3, dist, costcontext);
        if c < min_cost { best_dist = dist; min_cost = c; }
    }

    costmodel(best_length, best_dist, costcontext)
}
```

Figure 9.10: Candidate Rust translation emitted by the CodeGenerator. The `&dyn Fn`/`&dyn Any` pair safely translates the `void*`-plus-function-pointer pattern.

The EqTester-generated harness (Figure 9.12) loads the SpecMiner JSON, feeds both sides through translateArgs, calls the C function through `bindgen` FFI, calls the Rust candidate on the translated arguments, and asserts scalar equality with a 10^{-6} tolerance. This illustrates how all five components compose on a function involving function pointers, `void*`, and dynamic context typing.

9.11 Discussion and Limitations

Syzygy deliberately trades generality for reliability, and the resulting translations are test-validated rather than formally verified. We discuss the residual limitations across three categories: systemic limits of the problem formulation, implementation limits, and external validity.

Systemic Limitations

An ideal translation specification. Our problem formulation requires that the generated Rust program not use `unsafe`. In general, requiring no-`unsafe` may be an overly strong

```rust
pub unsafe fn translate_args_GetCostModelMinCost(
    costmodel_ptr: Option<
        unsafe extern "C" fn(
            litlen: ::std::os::raw::c_uint,
            dist: ::std::os::raw::c_uint,
            context: *mut ::std::os::raw::c_void,
        ) -> f64,
    >,
    costcontext: *mut ::std::os::raw::c_void,
) -> (Box<dyn Fn(u32, u32, &dyn std::any::Any) -> f64>, Box<dyn std::any::Any>) {
    rs_symtab_clear();

    let fn_ptr  = costmodel_ptr.expect("Function pointer is null");
    let fn_name = get_fn_name_from_fn_ptr(fn_ptr as *mut c_void)
        .expect("Failed to get function name from function pointer");

    if fn_name == "GetCostFixed" {
        let costmodel = Box::new(|litlen: u32, dist: u32, _ctx: &dyn Any| -> f64 {
            GetCostFixed(litlen, dist, std::ptr::null_mut())
        }) as Box<dyn Fn(u32, u32, &dyn std::any::Any) -> f64>;
        let costcontext = Box::new(()) as Box<dyn Any>;
        (costmodel, costcontext)
    } else if fn_name == "GetCostStat" {
        let rust_context = translate_singleton_and_peel::<SymbolStats>(costcontext)
            .expect("Failed to translate SymbolStats context");
        let costmodel = Box::new(move |litlen: u32, dist: u32, ctx: &dyn Any| -> f64 {
            let stats = ctx.downcast_ref::<SymbolStats>()
                .expect("Failed to downcast context to SymbolStats");
            zopfli_get_cost_stat(litlen as i32, dist as i32, stats)
        }) as Box<dyn Fn(u32, u32, &dyn std::any::Any) -> f64>;
        let costcontext = Box::new(rust_context) as Box<dyn Any>;
        (costmodel, costcontext)
    } else {
        panic!("Unknown function name: {}", fn_name);
    }
}
```

Figure 9.11: The LLM-generated translateArgs for `GetCostModelMinCost`, handling the translation of C function pointers to Rust closures. The symbol name recovered from the raw function-pointer address resolves the C function pointer to one of two observed targets.

constraint, as some unsafe executions are unexploitable in context. Low-level C code often uses such behaviors in contexts considered acceptable, and prohibiting them can lead to less idiomatic Rust code that may suffer from other bugs [191]. An ideal specification would forbid only *exploitable* `unsafe` behaviors, but formally characterizing exploitability remains open [181]. A lightweight mitigation is to let users annotate specific functions as `NO_TRANSLATE` and keep them in C, invoked through the foreign function interface.

No global refactoring. Our incremental-aligned translation preserves the source call graph and function signatures by construction. While this helps reduce the search space and provides per-function test feedback, it prevents the translation from performing global

```rust
pub unsafe fn eqtest_GetCostModelMinCost(filename: &str) -> bool {
    let (c_pre_arg0, c_pre_arg1) = load_pre_args_from_json(filename);

    let (rust_costmodel, rust_costcontext) =
        translate_args_GetCostModelMinCost(c_pre_arg0, c_pre_arg1);

    let c_result    = bindings::GetCostModelMinCost(c_pre_arg0, c_pre_arg1);
    let rust_result = get_cost_model_min_cost(&*rust_costmodel, &*rust_costcontext);

    let epsilon = 1e-6;
    assert!((c_result - rust_result).abs() < epsilon,
        "C and Rust functions returned different values: C={} vs Rust={}",
        c_result, rust_result);
    true
}
```

Figure 9.12: The LLM-generated equivalence test. The C side runs through `bindgen`-generated FFI; the Rust candidate runs on the translated closure and boxed context; return values are compared with floating-point tolerance.

refactoring. In some cases, global refactoring could lead to more idiomatic and efficient code.

Cascading backtracking. A side effect of our greedy strategy is the need for backtracking due to bad choices made earlier in translation. For example, translating an `int*` argument to a Rust array (`[i32; N]`) might clash with a later function that requires a `Vec`. While we currently orchestrate minimal repairs manually, future work can explore more systematic techniques.

Manual struct definitions. Accurately translating structs requires a global context of struct usage. Because structs are the leaves of the dependency graph, sampling struct definitions produces long backtracking chains when conflicts are detected later. Due to these constraints, we currently translate structs manually. Future work could explore using usage analysis to guide the automatic translation of structs.

Implementation Limitations

Type punning and `void*`. Our dynamic analysis depends on dynamic type inference (e.g., using `bitcast` operations). Raw memory accesses (e.g., manual address construction, type-punning) and `void*` pointers can obfuscate type information, leading to imprecise or incomplete analysis. Reverse-engineering programs that use pointer hacks forms an active research area [195].

Cyclic structs and multi-threading. Our current approach does not support cyclic C structs and multi-threading. Cyclic structs require `Weak` references to break cycles and avoid

memory leaks. Similarly, multi-threading requires synchronization primitives like `Arc` and `Mutex`. While LLMs can adapt to these cases, implementing the necessary program analysis to enable specification mining for them requires further work.

Cost and External Validity

Translation quality is bounded by the PASS@ANY of the underlying LLM [196]. Multiple samples improve reliability but increase translation time and cost. Furthermore, we rely on test-based validation, which is fundamentally prone to incomplete coverage. While we use over 1000000 validation tests to achieve high coverage, test-based validation has some chance of missing bugs. Finally, applying our translation approach to ZOPFLI could cause overfitting in the prompts and design decisions; going forward, we will evaluate SYZYGY on a broader suite of programs.

9.12 Related Work

SYZYGY sits at the intersection of automated C-to-Rust translation, LLM-driven code translation more broadly, dynamic analysis, and symbolic/formal translation techniques.

C-to-Rust Interest and Applications

C-to-Rust full code translation has recently garnered much interest owing to memory safety vulnerabilities [181] in C, strong typing and ownership in Rust, and the relative ease of combining C and Rust through CLANG compilation toolchains and the FFI. In particular, DARPA launched the TRACTOR program [183] aimed at automating translation. Compression libraries form a project milestone of the TRACTOR program, which motivates our main case study on ZOPFLI [182]. While translation is especially relevant for sensitive codebases (e.g., cryptographic/network libraries and OS kernels), opinion is divided on what part of the source C codebase to migrate. For example, Li et al. [191] argue that certain low-level utilities (e.g., drivers) are best left in C since migration can cause subtler bugs (e.g., issues with memory mapping).

Automated C-to-Rust Translation

The canonical baseline is c2rust [184], which mechanically translates C into *unsafe* Rust by mirroring C types through the Rust FFI. Beyond `unsafe`-ness, this results in artificial and less interpretable code. CROWN [186] post-processes c2rust output with ownership analysis and rule-based rewriting to convert it to safe Rust; while it scales well, usage of `unsafe` remains. Immunant and Galois aim to extend c2rust to produce safer Rust and have recently explored LLM-based nullability analysis [197, 198]; SYZYGY instead draws nullability from classical dynamic analysis.

Closest in spirit is Shiraishi and Shinagawa [187], which performs LLM-driven C-to-Rust translation but relies only on compile feedback, leaving per-function equivalence unchecked. As a result, they achieve lower accuracy on validation tests. VERT [185] pairs an LLM with an MSWasm oracle on programs under 200 lines, but without incremental translation and internal function tests. Flourine [199] adds differential testing and repair, again at small scale and without intermediate tests. Two concurrent systems target safe Rust: SACTOR [188] couples the LLM with static analysis and verification feedback, and C2SaferRust [189] bootstraps LLM repair on top of c2rust output. Relative to these approaches, SYZYGY generates fully safe-Rust code, validates it at every function boundary, and uses dynamic rather than static property mining.

LLM-driven Code Translation

A broader line of work trains LLMs for code translation, from unsupervised seq-to-seq [200, 201] and structure-aware decoders [202] to IR- and summarisation-aided approaches [203], and prompting variants that explain, rectify, or critique translations [204–207]. However, these works primarily focus on simple competition or interview-style programming problems.

Two recent works use LLMs to perform repository-level translation with test-based validation: Ibrahimzada et al. [208] study Java to Python, and Zhang et al. [209] study Go to Rust translation. While SYZYGY shares their incremental, test-gated philosophy, key aspects are unique to C-to-Rust. C constructs such as dynamic allocations, pointer-based direct memory accesses, aliasing, and nullability make interpreting C code challenging. The dynamic spec miner recovers these properties to aid generation.

Dynamic Analysis

SYZYGY's spec miner is a modern instance of instrumentation-based dynamic analysis [210]. Such techniques have been explored for test generation [211] and memory safety analysis [192]. The closest conceptual ancestor is Daikon's mining of likely invariants from executions [212]; our aliasing, nullability, and size properties are type-system-oriented cousins of Daikon's relational invariants. IODINE [213] applies similar ideas to hardware. The bounds table is a form of shadow memory [192, 193], reusing ideas from memory-safety tooling for a constructive purpose.

Symbolic and Formal Approaches

Previous work incorporates classical PL-oriented techniques in C-to-Rust translation. Hong and Ryu [214] use abstract interpretation over read/write sets and nullity to eliminate output parameters from c2rust output. While they focus on optimizing a pre-translated codebase, we apply dynamic analysis with LLMs to translate from scratch. Rule-based translation of restricted C fragments achieves formal guarantees at the cost of expressiveness [215]. Grammar-based synthesis and sketching [216–218] provide strong correctness guarantees

but have not been shown to scale beyond a handful of functions. SYZYGY occupies the complementary point: empirical rather than formal guarantees, at real-codebase scale.

9.13 Chapter Summary

In this chapter, we presented SYZYGY, an automated approach that synergistically combines dynamic analysis and testing with LLM sampling-driven search to translate C programs to Rust. By decomposing translation into individual units and performing dual code and test generation, we produce fully Rust-safe translations validated at the function boundary. We demonstrated the efficacy of our approach by correctly translating ZOPFLI, a data compression C program with over 3 000 lines of code, into roughly 4 500 lines of substantive Rust. The translation was validated by top-level equivalence testing on over one million compression inputs, achieving 95% line and 83% branch coverage on the source C program.

In the language of this thesis, this chapter introduces two new forms of executable specification—per-step function-level equivalence tests (mined from top-level traces) and structural dynamic-analysis annotations (TypeInfo, SizeInfo, NullInfo, AliasInfo). Where earlier chapters used executable specifications for evaluation (Chapters 2 to 4), training (Chapters 5 to 7), and inference-time aggregation (Chapter 8), this chapter uses them for intermediate decomposition, turning a long-horizon task into many short-horizon, individually testable steps.

Our approach outlines a path toward broader test-driven translation. Adding stronger dynamic analysis (e.g., to handle cyclic data structures and multi-threading), formal equivalence checks, and improved inference-time schedules may help scale the approach to more complex C-to-safe-Rust translations.

The next chapter closes the thesis by returning to the taxonomy of executable specifications introduced in Chapter 1.

Chapter 10

Conclusion

10.1 What We Learned

The three parts of this dissertation use executable specifications for evaluation (Chapters 2 to 4), training (Chapters 5 to 7), and agent design (Chapters 8 and 9). Across those settings, three lessons recur: executable specifications make program behavior usable as a signal, they can be collected at scale, and execution is a practical interface between programming agents and repository-level software tasks.

Executable specifications turn program behavior into a usable signal. Executable specifications are a useful intermediate form: they are concrete enough to run, easier to obtain than full formal specifications, and closer to semantics than raw source code. In LiveCodeBench they appear as input generators and gold-solution checks (Chapter 2); in R_2E and GSO as equivalence and performance harnesses (Chapters 3 and 4); in R2E-Gym and DeepSWE as Fail-to-Pass tests (Chapters 5 and 7); in Hybrid Verifiers as self-generated tests (Chapter 8); and in Syzygy as LLVM-derived runtime properties (Chapter 9). The form varies, but the role is the same: turn behavior into an artifact that another program can use.

We can reliably collect correctness, performance, and runtime properties at scale. The same construction pattern appears throughout the thesis: use an LLM to propose a spec-like artifact, run it, filter failures, and keep only artifacts with useful coverage. R_2E synthesizes 246 equivalence harnesses across 137 repositories (Chapter 3); R2E-Gym scales the recipe to 8 135 executable training environments (Chapter 5); GSO adapts it to per-commit performance harnesses across five languages (Chapter 4); and Syzygy applies the same idea to runtime-property mining with LLVM instrumentation (Chapter 9). The result is not a claim that specifications are free, but that much of the work can be shifted from manual authoring to generation, execution, and filtering.

Execution is a source of feedback for programming agents working on software. Execution is not only a final grade. It is also the mechanism through which an agent gets feedback while working. DeepSWE trains a 32B agent from the Fail-to-Pass reward alone (Chapter 7); Hybrid Verifiers use execution scores to aggregate rollouts at inference time (Chapter 8); Syzygy breaks C-to-Rust translation into checked per-function steps (Chapter 9); and Chapter 6 uses tests as a behavior-preservation constraint while improving code quality. Running code lets agents learn, verify, repair, and decompose tasks against the software they are changing.

These lessons also point to the limits of the work so far: how specifications are produced, where execution is harder to obtain, and what kinds of feedback execution cannot provide by itself.

10.2 Future Directions

The projects in this thesis make executable specifications useful in several settings, but leave three important questions open: how to synthesize better specifications, how to use execution when running code is itself difficult, and how to combine execution with other forms of feedback.

Beyond hand-engineered specs. Most specifications in this thesis are hand-crafted once, then applied at scale, or derived from artifacts humans already wrote. LiveCodeBench uses per-problem generators and gold solutions (Chapter 2); R_2E uses fixed prompts for harness and test generation (Chapter 3); R2E-Gym builds on human-written Fail-to-Pass tests (Chapter 5); and Syzygy relies on hand-authored dynamic analyses (Chapter 9). A natural next step is to make specification synthesis part of the agent's work: an agent should not only check against a fixed spec, but also generate the specs that help it solve the task. Self-generated tests in Hybrid Verifiers (Chapter 8) are an early example. More generally, we need methods that learn which specifications detect behaviorally relevant defects, and systems that can generate the instrumentation needed for new languages and runtime properties.

Domains where execution is non-trivial. The evaluated settings are relatively friendly to execution: competition programs run quickly (Chapter 2), repository tests run in Docker (Chapters 3, 5 and 7), and Syzygy checks statically compiled binaries (Chapter 9). Many programming tasks are less tidy. Open-ended requests may need preference-based specs rather than Boolean correctness. Long-horizon tasks, such as multi-file refactors or full application builds, need intermediate checks that do not require humans to specify every step. GPU kernels, embedded systems, security-critical code, and distributed systems may require safety invariants, resource constraints, or reasoning about non-deterministic executions. Extending executable specifications to these domains is a central test of the framework.

Richer feedback in addition to execution. Execution gives strong evidence about behavior, but it does not capture every property users care about. Readability, modularity, idiomaticity, review feedback, latency, energy use, and cost all matter. Chapter 6 points to one pattern: use tests as a behavior-preservation constraint while another signal drives improvement. Future systems can generalize this approach by treating execution as an independently checkable behavioral constraint and combining it with learned metrics, human feedback, and deployment-time measurements.

Together, these directions ask whether executable specifications can extend from fixed artifacts to learned ones, from easy-to-run programs to difficult execution settings, and from correctness checks to broader software-engineering feedback.

References

[1] S. Yang, W.-L. Chiang, L. Zheng, J. E. Gonzalez, and I. Stoica. *Rethinking Benchmark and Contamination for Language Models with Rephrased Samples*. 2023. arXiv: 2311.04850 [cs.CL].

[2] G. Team. *Gemini 1.5: Unlocking multimodal understanding across millions of tokens of context*. 2024.

[3] 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". In: *arXiv preprint arXiv:2107.03374* (2021).

[4] 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". In: *arXiv preprint arXiv:2108.07732* (2021).

[5] Y. Li, D. Choi, J. Chung, N. Kushman, J. Schrittwieser, R. Leblond, T. Eccles, J. Keeling, F. Gimeno, A. D. Lago, T. Hubert, P. Choy, C. de Masson d'Autume, I. Babuschkin, X. Chen, P.-S. Huang, J. Welbl, S. Gowal, A. Cherepanov, J. Molloy, D. J. Mankowitz, E. S. Robson, P. Kohli, N. de Freitas, K. Kavukcuoglu, and O. Vinyals. "Competition-level code generation with AlphaCode". In: *Science* 378.6624 (2022), pp. 1092–1097. eprint: https://www.science.org/doi/pdf/10.1126/science.abq1158.

[6] M. Zhong, G. Liu, H. Li, J. Kuang, J. Zeng, and M. Wang. "CodeGen-Test: An Automatic Code Generation Model Integrating Program Test Information". In: *arXiv preprint arXiv:2202.07612* (2022).

[7] L. B. Allal, R. Li, D. Kocetkov, C. Mou, C. Akiki, C. M. Ferrandis, N. Muennighoff, M. Mishra, A. Gu, M. Dey, et al. "SantaCoder: don't reach for the stars!" In: *arXiv preprint arXiv:2301.03988* (2023).

[8] 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!" In: *arXiv preprint arXiv:2305.06161* (2023).

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

[10] D. AI. *DeepSeek Coder: Let the Code Write Itself*. https://github.com/deepseek-ai/DeepSeek-Coder. 2023.

[11] Z. Luo, C. Xu, P. Zhao, Q. Sun, X. Geng, W. Hu, C. Tao, J. Ma, Q. Lin, and D. Jiang. "WizardCoder: Empowering Code Large Language Models with Evol-Instruct". In: *arXiv preprint arXiv:2306.08568* (2023).

[12] Y. Wei, Z. Wang, J. Liu, Y. Ding, and L. Zhang. "Magicoder: Source code is all you need". In: *arXiv preprint arXiv:2312.02120* (2023).

[13] T. Ridnik, D. Kredo, and I. Friedman. "Code Generation with AlphaCodium: From Prompt Engineering to Flow Engineering". In: *arXiv preprint arXiv:2401.08500* (2024).

[14] A. Lozhkov, R. Li, L. B. Allal, F. Cassano, J. Lamy-Poirier, N. Tazi, A. Tang, D. Pykhtar, J. Liu, Y. Wei, T. Liu, M. Tian, D. Kocetkov, A. Zucker, Y. Belkada, Z. Wang, Q. Liu, D. Abulkhanov, I. Paul, Z. Li, W.-D. Li, M. Risdal, J. Li, J. Zhu, T. Y. Zhuo, E. Zheltonozhskii, N. O. O. Dade, W. Yu, L. Krauß, N. Jain, Y. Su, X. He, M. Dey, E. Abati, Y. Chai, N. Muennighoff, X. Tang, M. Oblokulov, C. Akiki, M. Marone, C. Mou, M. Mishra, A. Gu, B. Hui, T. Dao, A. Zebaze, O. Dehaene, N. Patry, C. Xu, J. McAuley, T. Scholak, S. Paquet, J. Robinson, C. J. Anderson, N. Chapados, M. Patwary, N. Tajbakhsh, Y. Jernite, C. M. Ferrandis, L. Zhang, S. Hughes, T. Wolf, A. Guha, L. von Werra, and H. de Vries. "StarCoder 2 and The Stack v2: The Next Generation". In: (2024).

[15] T. Zheng, G. Zhang, T. Shen, X. Liu, B. Y. Lin, J. Fu, W. Chen, and X. Yue. "OpenCodeInterpreter: Integrating Code Generation with Execution and Refinement". In: *https://arxiv.org/abs/2402.14658* (2024).

[16] T. X. Olausson, J. P. Inala, C. Wang, J. Gao, and A. Solar-Lezama. "Demystifying GPT Self-Repair for Code Generation". In: *arXiv preprint arXiv:2306.09896* (2023).

[17] A. Madaan, A. Shypula, U. Alon, M. Hashemi, P. Ranganathan, Y. Yang, G. Neubig, and A. Yazdanbakhsh. "Learning performance-improving code edits". In: *arXiv preprint arXiv:2302.07867* 8 (2023).

[18] B. Steenhoek, M. Tufano, N. Sundaresan, and A. Svyatkovskiy. "Reinforcement Learning from Automatic Feedback for High-Quality Unit Test Generation". In: *arXiv preprint arXiv:2310.02368* (2023).

[19] Q. Luo, Y. Ye, S. Liang, Z. Zhang, Y. Qin, Y. Lu, Y. Wu, X. Cong, Y. Lin, Y. Zhang, et al. "RepoAgent: An LLM-Powered Open-Source Framework for Repository-level Code Documentation Generation". In: *arXiv preprint arXiv:2402.16667* (2024).

[20] S. G. Patil, T. Zhang, X. Wang, and J. E. Gonzalez. "Gorilla: Large language model connected with massive apis". In: *arXiv preprint arXiv:2305.15334* (2023).

[21] Y. Qin, S. Liang, Y. Ye, K. Zhu, L. Yan, Y. Lu, Y. Lin, X. Cong, X. Tang, B. Qian, S. Zhao, L. Hong, R. Tian, R. Xie, J. Zhou, M. Gerstein, dahai li, Z. Liu, and M. Sun. "ToolLLM: Facilitating Large Language Models to Master 16000+ Real-world APIs". In: *The Twelfth International Conference on Learning Representations*. 2024.

[22] R. Sun, S. O. Arik, H. Nakhost, H. Dai, R. Sinha, P. Yin, and T. Pfister. "SQL-PaLM: Improved Large Language ModelAdaptation for Text-to-SQL". In: *arXiv preprint arXiv:2306.00739* (2023).

[23] Y. Li, S. Bubeck, R. Eldan, A. Del Giorno, S. Gunasekar, and Y. T. Lee. "Textbooks are all you need ii: phi-1.5 technical report". In: *arXiv preprint arXiv:2309.05463* (2023).

[24] 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". In: *Advances in Neural Information Processing Systems* 36 (2023), pp. 21558–21572.

[25] R. OpenAI. "GPT-4 technical report. arXiv 2303.08774". In: *View in Article* (2023).

[26] Y. Huang, Z. Lin, X. Liu, Y. Gong, S. Lu, F. Lei, Y. Liang, Y. Shen, C. Lin, N. Duan, et al. "Competition-Level Problems are Effective LLM Evaluators". In: *arXiv preprint arXiv:2312.02143* (2023).

[27] 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". In: *arXiv preprint arXiv:2312.14852* (2023).

[28] C. Liu, S. D. Zhang, and R. Jabbarvand. "CodeMind: A Framework to Challenge Large Language Models for Code Reasoning". In: *arXiv preprint arXiv:2402.09664* (2024).

[29] M. Singhal, T. Aggarwal, A. Awasthi, N. Natarajan, and A. Kanade. "NoFunEval: Funny How Code LMs Falter on Requirements Beyond Functional Correctness". In: *arXiv preprint arXiv:2401.15963* (2024).

[30] 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". In: *arXiv preprint arXiv:2403.07974* (2024).

[31] B. Boehm. "A view of 20th and 21st century software engineering". In: *Proceedings of the 28th International Conference on Software Engineering*. ICSE '06. Shanghai, China: Association for Computing Machinery, 2006, pp. 12–29.

[32] N. Shinn, F. Cassano, A. Gopinath, K. R. Narasimhan, and S. Yao. "Reflexion: Language agents with verbal reinforcement learning". In: *Thirty-seventh Conference on Neural Information Processing Systems*. 2023.

[33] X. Chen, M. Lin, N. Schärli, and D. Zhou. "Teaching large language models to self-debug". In: *arXiv preprint arXiv:2304.05128* (2023).

[34] 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: *arXiv preprint arXiv:2401.03065* (2024).

[35] S. Kulal, P. Pasupat, K. Chandra, M. Lee, O. Padon, A. Aiken, and P. S. Liang. "Spoc: Search-based pseudocode to code". In: *Advances in Neural Information Processing Systems* 32 (2019).

[36] W. Kwon, Z. Li, S. Zhuang, Y. Sheng, L. Zheng, C. H. Yu, J. E. Gonzalez, H. Zhang, and I. Stoica. "Efficient Memory Management for Large Language Model Serving with PagedAttention". In: *Proceedings of the ACM SIGOPS 29th Symposium on Operating Systems Principles*. 2023.

[37] D. Hendrycks, C. Burns, S. Kadavath, A. Arora, S. Basart, E. Tang, D. Song, and J. Steinhardt. "Measuring mathematical problem solving with the math dataset". In: *arXiv preprint arXiv:2103.03874* (2021).

[38] L. Ouyang, J. Wu, X. Jiang, D. Almeida, C. Wainwright, P. Mishkin, C. Zhang, S. Agarwal, K. Slama, A. Ray, et al. "Training language models to follow instructions with human feedback". In: *Advances in Neural Information Processing Systems* 35 (2022), pp. 27730–27744.

[39] Y. Wang, Y. Kordi, S. Mishra, A. Liu, N. A. Smith, D. Khashabi, and H. Hajishirzi. "Self-instruct: Aligning language model with self generated instructions". In: *arXiv preprint arXiv:2212.10560* (2022).

[40] Y. Wang, W. Wang, S. Joty, and S. C. Hoi. "CodeT5: Identifier-aware Unified Pre-trained Encoder-Decoder Models for Code Understanding and Generation". In: *Proceedings of the 2021 Conference on Empirical Methods in Natural Language Processing*. 2021, pp. 8696–8708.

[41] Y. Wang, H. Le, A. D. Gotmare, N. D. Bui, J. Li, and S. C. Hoi. "CodeT5+: Open code large language models for code understanding and generation". In: *arXiv preprint arXiv:2305.07922* (2023).

[42] 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". In: *The Eleventh International Conference on Learning Representations*. 2022.

[43] D. Fried, A. Aghajanyan, J. Lin, S. Wang, E. Wallace, F. Shi, R. Zhong, W.-t. Yih, L. Zettlemoyer, and M. Lewis. "InCoder: A Generative Model for Code Infilling and Synthesis". In: *preprint arXiv:2204.05999* (2022).

[44] Q. Zheng, X. Xia, X. Zou, Y. Dong, S. Wang, Y. Xue, Z. Wang, L. Shen, A. Wang, Y. Li, et al. "Codegeex: A pre-trained model for code generation with multilingual evaluations on humaneval-x". In: *arXiv preprint arXiv:2303.17568* (2023).

[45] X. Bi, D. Chen, G. Chen, S. Chen, D. Dai, C. Deng, H. Ding, K. Dong, Q. Du, Z. Fu, et al. "DeepSeek LLM: Scaling Open-Source Language Models with Longtermism". In: *arXiv preprint arXiv:2401.02954* (2024).

[46] A. Ni, P. Yin, Y. Zhao, M. Riddell, T. Feng, R. Shen, S. Yin, Y. Liu, S. Yavuz, C. Xiong, et al. "L2CEval: Evaluating Language-to-Code Generation Capabilities of Large Language Models". In: *arXiv preprint arXiv:2309.17446* (2023).

[47] F. Cassano, J. Gouwar, D. Nguyen, S. Nguyen, L. Phipps-Costin, D. Pinckney, M.-H. Yee, Y. Zi, C. J. Anderson, M. Q. Feldman, et al. "MultiPL-E: A Scalable and Extensible Approach to Benchmarking Neural Code Generation". In: *arXiv preprint arXiv:2208.08227* (2022).

[48] S. Wang, Z. Li, H. Qian, C. Yang, Z. Wang, M. Shang, V. Kumar, S. Tan, B. Ray, P. Bhatia, et al. "ReCode: Robustness Evaluation of Code Generation Models". In: *arXiv preprint arXiv:2212.10264* (2022).

[49] 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". In: *arXiv preprint arXiv:2210.14868* (2022).

[50] 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: *International Conference on Machine Learning*. PMLR. 2023, pp. 18319–18345.

[51] P. Yin, W.-D. Li, K. Xiao, A. Rao, Y. Wen, K. Shi, J. Howland, P. Bailey, M. Catasta, H. Michalewski, et al. "Natural language to code generation in interactive data science notebooks". In: *arXiv preprint arXiv:2212.09248* (2022).

[52] K. Zhang, G. Li, J. Li, Z. Li, and Z. Jin. "ToolCoder: Teach Code Generation Models to use APIs with search tools". In: *arXiv preprint arXiv:2305.04032* (2023).

[53] N. Jain, S. Vaidyanath, A. Iyer, N. Natarajan, S. Parthasarathy, S. Rajamani, and R. Sharma. "Jigsaw: Large language models meet program synthesis". In: *Proceedings of the 44th International Conference on Software Engineering*. 2022, pp. 1219–1231.

[54] Z. Wang, S. Zhou, D. Fried, and G. Neubig. "Execution-based evaluation for open-domain code generation". In: *arXiv preprint arXiv:2212.10481* (2022).

[55] R. Agashe, S. Iyer, and L. Zettlemoyer. "Juice: A large scale distantly supervised dataset for open domain context-based code generation". In: *arXiv preprint arXiv:1910.02216* (2019).

[56] T. Liu, C. Xu, and J. McAuley. "RepoBench: Benchmarking Repository-Level Code Auto-Completion Systems". In: *arXiv preprint arXiv:2306.03091* (2023).

[57] D. Shrivastava, H. Larochelle, and D. Tarlow. "Repository-level prompt generation for large language models of code". In: *International Conference on Machine Learning*. PMLR. 2023, pp. 31693–31715.

[58] F. Zhang, B. Chen, Y. Zhang, J. Liu, D. Zan, Y. Mao, J.-G. Lou, and W. Chen. "Repocoder: Repository-level code completion through iterative retrieval and generation". In: *arXiv preprint arXiv:2303.12570* (2023).

[59] Y. Ding, Z. Wang, W. U. Ahmad, M. K. Ramanathan, R. Nallapati, P. Bhatia, D. Roth, and B. Xiang. "Cocomic: Code completion by jointly modeling in-file and cross-file context". In: *arXiv preprint arXiv:2212.10007* (2022).

[60] 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?" In: *arXiv preprint arXiv:2310.06770* (2023).

[61] M. A. M. Khan, M. S. Bari, X. L. Do, W. Wang, M. R. Parvez, and S. Joty. "xCodeEval: A Large Scale Multilingual Multitask Benchmark for Code Understanding, Generation, Translation and Retrieval". In: *arXiv preprint arXiv:2303.03004* (2023).

[62] K. Zhang, D. Wang, J. Xia, W. Y. Wang, and L. Li. "ALGO: Synthesizing Algorithmic Programs with Generated Oracle Verifiers". In: *arXiv preprint arXiv:2305.14591* (2023).

[63] E. Zelikman, Q. Huang, G. Poesia, N. D. Goodman, and N. Haber. *Parsel : Algorithmic Reasoning with Language Models by Composing Decompositions*. 2022.

[64] N. Jain, T. Zhang, W.-L. Chiang, J. E. Gonzalez, K. Sen, and I. Stoica. "LLM-Assisted Code Cleaning For Training Accurate Code Generators". In: *arXiv preprint arXiv:2311.14904* (2023).

[65] M. Yasunaga, X. Chen, Y. Li, P. Pasupat, J. Leskovec, P. Liang, E. H. Chi, and D. Zhou. "Large Language Models as Analogical Reasoners". In: *arXiv preprint arXiv:2310.01714* (2023).

[66] A. Madaan, N. Tandon, P. Gupta, S. Hallinan, L. Gao, S. Wiegreffe, U. Alon, N. Dziri, S. Prabhumoye, Y. Yang, et al. "Self-refine: Iterative refinement with self-feedback". In: *arXiv preprint arXiv:2303.17651* (2023).

[67] B. Peng, M. Galley, P. He, H. Cheng, Y. Xie, Y. Hu, Q. Huang, L. Liden, Z. Yu, W. Chen, and J. Gao. "Check your facts and try again: Improving large language models with external knowledge and automated feedback". In: *arXiv preprint arXiv:2302.12813* (2023).

[68] K. Zhang, Z. Li, J. Li, G. Li, and Z. Jin. "Self-Edit: Fault-Aware Code Editor for Code Generation". In: *Proceedings of the 61st Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*. Toronto, Canada: Association for Computational Linguistics, July 2023, pp. 769–787.

[69] M. Nye, A. J. Andreassen, G. Gur-Ari, H. Michalewski, J. Austin, D. Bieber, D. Dohan, A. Lewkowycz, M. Bosma, D. Luan, et al. "Show your work: Scratchpads for intermediate computation with language models". In: *arXiv preprint arXiv:2112.00114* (2021).

[70] B. Chen, F. Zhang, A. Nguyen, D. Zan, Z. Lin, J.-G. Lou, and W. Chen. "Codet: Code generation with generated tests". In: *arXiv preprint arXiv:2207.10397* (2022).

[71] Z. Yuan, Y. Lou, M. Liu, S. Ding, K. Wang, Y. Chen, and X. Peng. "No More Manual Tests? Evaluating and Improving ChatGPT for Unit Test Generation". In: *arXiv preprint arXiv:2305.04207* (2023).

[72] M. Schäfer, S. Nadi, A. Eghbali, and F. Tip. "An Empirical Evaluation of Using Large Language Models for Automated Unit Test Generation". In: *IEEE Transactions on Software Engineering* 50.1 (2024), pp. 85–105.

[73] M. Tufano, S. K. Deng, N. Sundaresan, and A. Svyatkovskiy. "Methods2Test: A dataset of focal methods mapped to test cases". In: *Proceedings of the 19th International Conference on Mining Software Repositories*. 2022, pp. 299–303.

[74] C. Watson, M. Tufano, K. Moran, G. Bavota, and D. Poshyvanyk. "On learning meaningful assert statements for unit test cases". In: *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering*. 2020, pp. 1398–1409.

[75] O. Sainz, J. A. Campos, I. García-Ferrero, J. Etxaniz, and E. Agirre. *Did ChatGPT cheat on your test?* June 2023.

[76] S. Golchin and M. Surdeanu. "Time travel in llms: Tracing data contamination in large language models". In: *arXiv preprint arXiv:2308.08493* (2023).

[77] O. Weller, M. Marone, N. Weir, D. Lawrie, D. Khashabi, and B. Van Durme. "" According to..." Prompting Language Models Improves Quoting from Pre-Training Data". In: *arXiv preprint arXiv:2305.13252* (2023).

[78] Y. Oren, N. Meister, N. Chatterji, F. Ladhak, and T. B. Hashimoto. "Proving Test Set Contamination for Black-Box Language Models". In: *The Twelfth International Conference on Learning Representations*. 2024.

[79] W. Shi, A. Ajith, M. Xia, Y. Huang, D. Liu, T. Blevins, D. Chen, and L. Zettlemoyer. "Detecting pretraining data from large language models". In: *arXiv preprint arXiv:2310.16789* (2023).

[80] 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*. 2024.

[81] K. Zhou, Y. Zhu, Z. Chen, W. Chen, W. X. Zhao, X. Chen, Y. Lin, J.-R. Wen, and J. Han. "Don't Make Your LLM an Evaluation Benchmark Cheater". In: *arXiv preprint arXiv:2311.01964* (2023).

[82] M. Ivanković, G. Petrović, R. Just, and G. Fraser. "Code coverage at Google". In: *Proceedings of the 2019 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. 2019, pp. 955–963.

[83] B. G. Ryder. "Constructing the call graph of a program". In: *IEEE Transactions on Software Engineering* 3 (1979), pp. 216–226.

[84] V. Salis, T. Sotiropoulos, P. Louridas, D. Spinellis, and D. Mitropoulos. "Pycg: Practical call graph generation in python". In: *2021 IEEE/ACM 43rd International Conference on Software Engineering (ICSE)*. IEEE. 2021, pp. 1646–1657.

[85] N. Jain, M. Shetty, T. Zhang, K. Han, K. Sen, and I. Stoica. "R2E: Turning any Github Repository into a Programming Agent Environment". In: *ICML 2024*. 2024.

[86] Y. Ding, Z. Wang, W. U. 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". In: *arXiv preprint arXiv:2310.11248* (2023).

[87] T. Zhang, T. Yu, T. Hashimoto, M. Lewis, W.-t. Yih, D. Fried, and S. Wang. "Coder reviewer reranking for code generation". In: *International Conference on Machine Learning*. PMLR. 2023, pp. 41832–41846.

[88] E. C. Blog. "Quantifying GitHub Copilot's impact on code quality". In: *https://www.expresscomputer.in/news/quantifying-github-copilots-impact-on-code-quality-ai/104480/* ().

[89] P. Yin, B. Deng, E. Chen, B. Vasilescu, and G. Neubig. "Learning to Mine Aligned Code and Natural Language Pairs from Stack Overflow". In: *International Conference on Mining Software Repositories*. MSR. ACM, 2018, pp. 476–486.

[90] D. Shrivastava, D. Kocetkov, H. de Vries, D. Bahdanau, and T. Scholak. "RepoFusion: Training Code Models to Understand Your Repository". In: *arXiv preprint arXiv:2306.10998* (2023).

[91] 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*. 2023.

[92] C. Liu, S. Lu, W. Chen, D. Jiang, A. Svyatkovskiy, S. Fu, N. Sundaresan, and N. Duan. "Code Execution with Pre-trained Language Models". In: *arXiv preprint arXiv:2305.05383* (2023).

[93] D. Key, W.-D. Li, and K. Ellis. "I speak, you verify: Toward trustworthy neural program synthesis". In: *arXiv preprint arXiv:2210.00848* (2022).

[94] E. Zelikman, Q. Huang, G. Poesia, N. D. Goodman, and N. Haber. "Parsel: A (de-) compositional framework for algorithmic reasoning with language models". In: *arXiv preprint arXiv:2212.10561* (2023).

[95] B. Romera-Paredes, M. Barekatain, A. Novikov, M. Balog, M. P. Kumar, E. Dupont, F. J. Ruiz, J. S. Ellenberg, P. Wang, O. Fawzi, et al. "Mathematical discoveries from program search with large language models". In: *Nature* (2023), pp. 1–3.

[96] A. Ni, S. Iyer, D. Radev, V. Stoyanov, W.-t. Yih, S. Wang, and X. V. Lin. "Lever: Learning to verify language-to-code generation with execution". In: *International Conference on Machine Learning*. PMLR. 2023, pp. 26106–26128.

[97] R. Bairi, A. Sonwane, A. Kanade, V. D C, A. Iyer, S. Parthasarathy, S. Rajamani, B. Ashok, and S. Shet. "CodePlan: Repository-level Coding using LLMs and Planning". In: *Neural Information Processing Systems Workshop on Foundation Models for Decision Making (FMDM-NeurIPS)*. Nov. 2023.

[98] S. Yao, J. Zhao, D. Yu, N. Du, I. Shafran, K. R. Narasimhan, and Y. Cao. "ReAct: Synergizing Reasoning and Acting in Language Models". In: *The Eleventh International Conference on Learning Representations*. 2022.

[99] S. Yao, D. Yu, J. Zhao, I. Shafran, T. L. Griffiths, Y. Cao, and K. Narasimhan. "Tree of thoughts: Deliberate problem solving with large language models". In: *arXiv preprint arXiv:2305.10601* (2023).

[100] A. Singh, J. D. Co-Reyes, R. Agarwal, A. Anand, P. Patil, P. J. Liu, J. Harrison, J. Lee, K. Xu, A. Parisi, et al. "Beyond human data: Scaling self-training for problem-solving with language models". In: *arXiv preprint arXiv:2312.06585* (2023).

[101] 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* 35 (2022), pp. 21314–21328.

[102] G. Bradski. "The OpenCV Library". In: *Dr. Dobb's Journal of Software Tools* (2000).

[103] G. Sheng, C. Zhang, Z. Ye, X. Wu, W. Zhang, R. Zhang, Y. Peng, H. Lin, and C. Wu. "HybridFlow: A Flexible and Efficient RLHF Framework". In: *arXiv preprint arXiv: 2409.19256* (2024).

[104] C. E. Jimenez, J. Yang, A. Wettig, S. Yao, K. Pei, O. Press, and K. R. Narasimhan. "SWE-bench: Can Language Models Resolve Real-world Github Issues?" In: *The Twelfth International Conference on Learning Representations*. 2024.

[105] A. Ouyang, S. Guo, and A. Mirhoseini. *KernelBench: Can LLMs Write GPU Kernels?* 2024.

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

[107] X. Wang, B. Li, Y. Song, F. F. Xu, X. Tang, M. Zhuge, J. Pan, Y. Song, B. Li, J. Singh, H. H. Tran, F. Li, R. Ma, M. Zheng, B. Qian, Y. Shao, N. Muennighoff, Y. Zhang, B. Hui, J. Lin, R. Brennan, H. Peng, H. Ji, and G. Neubig. *OpenHands: An Open Platform for AI Software Developers as Generalist Agents*. 2024. arXiv: 2407.16741 [cs.SE].

[108] B. Jacob and T. N. Mudge. *Notes on calculating computer performance*. University of Michigan, Computer Science and Engineering Division ..., 1995.

[109] H. Lightman, V. Kosaraju, Y. Burda, H. Edwards, B. Baker, T. Lee, J. Leike, J. Schulman, I. Sutskever, and K. Cobbe. "Let's Verify Step by Step". In: *arXiv preprint arXiv:2305.20050* (2023).

[110] M. Shetty, N. Jain, J. Liu, V. Kethanaboyina, K. Sen, and I. Stoica. "GSO: Challenging Software Optimization Tasks for Evaluating SWE-Agents". In: *Advances in Neural Information Processing Systems (NeurIPS)*. 2025.

[111] T. X. Olausson, J. P. Inala, C. Wang, J. Gao, and A. Solar-Lezama. "Is self-repair a silver bullet for code generation?" In: *arXiv preprint arXiv:2306.09896* (2023).

[112] J. Li, S. Tworkowski, Y. Wu, and R. Mooney. "Explaining competitive-level programming solutions using llms". In: *arXiv preprint arXiv:2307.05337* (2023).

[113] E. Wang, F. Cassano, C. Wu, Y. Bai, W. Song, V. Nath, Z. Han, S. Hendryx, S. Yue, and H. Zhang. "Planning in natural language improves llm search for code generation". In: *arXiv preprint arXiv:2409.03733* (2024).

[114] H. Pham, X. Wang, Y. Yang, and G. Neubig. "Meta back-translation". In: *arXiv preprint arXiv:2102.07847* (2021).

[115] R. Sennrich, B. Haddow, and A. Birch. "Improving neural machine translation models with monolingual data". In: *arXiv preprint arXiv:1511.06709* (2015).

[116] 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". In: *arXiv preprint arXiv:2406.15877* (2024).

[117] 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: *NeurIPS* (2021).

[118] W. Yan, H. Liu, Y. Wang, Y. Li, Q. Chen, W. Wang, T. Lin, W. Zhao, L. Zhu, H. Sundaram, et al. "Codescope: An execution-based multilingual multitask multidimensional benchmark for evaluating llms on code understanding and generation". In: *arXiv preprint arXiv:2311.08588* (2023).

[119] J. Liu, S. Xie, J. Wang, Y. Wei, Y. Ding, and L. ZHANG. "Evaluating Language Models for Efficient Code Generation". In: *First Conference on Language Modeling*. 2024.

[120] D. Huang, Y. Qing, W. Shang, H. Cui, and J. Zhang. "Effibench: Benchmarking the efficiency of automatically generated code". In: *Advances in Neural Information Processing Systems* 37 (2024), pp. 11506–11544.

[121] S. Waghjale, V. Veerendranath, Z. Z. Wang, and D. Fried. "ECCO: Can We Improve Model-Generated Code Efficiency Without Sacrificing Functional Correctness?" In: *arXiv preprint arXiv:2407.14044* (2024).

[122] T. Coignion, C. Quinton, and R. Rouvoy. "A performance study of llm-generated code on leetcode". In: *Proceedings of the 28th International Conference on Evaluation and Assessment in Software Engineering*. 2024, pp. 79–89.

[123] C. Niu, T. Zhang, C. Li, B. Luo, and V. Ng. "On evaluating the efficiency of source code generated by llms". In: *Proceedings of the 2024 IEEE/ACM First International Conference on AI Foundation Models and Software Engineering*. 2024, pp. 103–107.

[124] 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?" In: *arXiv preprint arXiv:2410.03859* (2024).

[125] 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. arXiv: 2504.02605 [cs.SE].

[126] K. Kabir, J. Yang, C. E. Jimenez, et al. *SWE-Bench Multilingual*. https://kabirk.com/multilingual. Accessed: May 2024. 2024.

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

[128] T. Ahmed, M. Hirzel, R. Pan, A. Shinnar, and S. Sinha. "TDD-Bench Verified: Can LLMs Generate Tests for Issues Before They Get Resolved?" In: *arXiv preprint arXiv:2412.02883* (2024).

[129] Z. Chen, X. Tang, G. Deng, F. Wu, J. Wu, Z. Jiang, V. Prasanna, A. Cohan, and X. Wang. "Locagent: Graph-guided llm agents for code localization". In: *arXiv preprint arXiv:2503.09089* (2025).

[130] W. Zhao, N. Jiang, C. Lee, J. T. Chiu, C. Cardie, M. Gallé, and A. M. Rush. "Commit0: Library Generation from Scratch". In: *arXiv preprint arXiv:2412.01769* (2024).

[131] Y. Xie, A. Xie, D. Sheth, P. Liu, D. Fried, and C. Rose. "RepoST: Scalable Repository-Level Coding Environment Construction with Sandbox Testing". In: *arXiv preprint arXiv:2503.07358* (2025).

[132] METR. *Measuring Automated Kernel Engineering*. https://metr.org/blog/2025-02-14-measuring-automated-kernel-engineering/. Feb. 2025.

[133] A. Gu, N. Jain, W.-D. Li, M. Shetty, Y. Shao, Z. Li, D. Yang, K. Ellis, K. Sen, and A. Solar-Lezama. "Challenges and Paths Towards AI for Software Engineering". In: *arXiv preprint arXiv:2503.22625* (2025).

[134] R. T. Lange, A. Prasad, Q. Sun, M. Faldor, Y. Tang, and D. Ha. "The AI CUDA Engineer: Agentic CUDA Kernel Discovery, Optimization and Composition". In: (2025).

[135] J. Pan, X. Wang, G. Neubig, N. Jaitly, H. Ji, A. Suhr, and Y. Zhang. *Training Software Engineering Agents and Verifiers with SWE-Gym*. 2024. arXiv: 2412.21139 [cs.SE].

[136] X. Wang, B. Li, Y. Song, F. F. Xu, X. Tang, M. Zhuge, J. Pan, Y. Song, B. Li, J. Singh, et al. "Openhands: An open platform for ai software developers as generalist agents". In: *arXiv preprint arXiv:2407.16741* (2024).

[137] Anthropic. *Claude 3.7 Sonnet*. https://www.anthropic.com/news/claude-3-7-sonnet. Feb. 2025.

[138] A. Jaech, A. Kalai, A. Lerer, A. Richardson, A. El-Kishky, A. Low, A. Helyar, A. Madry, A. Beutel, A. Carney, et al. "Openai o1 system card". In: *arXiv preprint arXiv:2412.16720* (2024).

[139] C. Xie, B. Li, C. Gao, H. Du, W. Lam, D. Zou, and K. Chen. "SWE-Fixer: Training Open-Source LLMs for Effective and Efficient GitHub Issue Resolution". In: *arXiv preprint arXiv:2501.05040* (2025).

[140] X. Li, P. Yu, C. Zhou, T. Schick, O. Levy, L. Zettlemoyer, J. Weston, and M. Lewis. "Self-alignment with instruction backtranslation". In: *arXiv preprint arXiv:2308.06259* (2023).

[141] Y. Zheng, R. Zhang, J. Zhang, Y. Ye, Z. Luo, Z. Feng, and Y. Ma. "LlamaFactory: Unified Efficient Fine-Tuning of 100+ Language Models". In: *Proceedings of the 62nd Annual Meeting of the Association for Computational Linguistics (Volume 3: System Demonstrations)*. Bangkok, Thailand: Association for Computational Linguistics, 2024.

[142] B. Rozière, J. Gehring, F. Gloeckle, S. Sootla, I. Gat, X. E. Tan, Y. Adi, J. Liu, T. Remez, J. Rapin, et al. "Code llama: Open foundation models for code". In: *arXiv preprint arXiv:2308.12950* (2023).

[143] X.-Y. Li, J.-T. Xue, Z. Xie, and M. Li. "Think Outside the Code: Brainstorming Boosts Large Language Models in Code Generation". In: *arXiv preprint arXiv:2305.10679* (2023).

[144] C. Zhou, P. Liu, P. Xu, S. Iyer, J. Sun, Y. Mao, X. Ma, A. Efrat, P. Yu, L. Yu, et al. "Lima: Less is more for alignment". In: *arXiv preprint arXiv:2305.11206* (2023).

[145] Y. Cao, Y. Kang, and L. Sun. "Instruction mining: High-quality instruction data selection for large language models". In: *arXiv preprint arXiv:2307.06290* (2023).

[146] H. Chen, Y. Zhang, Q. Zhang, H. Yang, X. Hu, X. Ma, Y. Yanggong, and J. Zhao. "Maybe Only 0.5% Data is Needed: A Preliminary Exploration of Low Training Data Instruction Tuning". In: *arXiv preprint arXiv:2305.09246* (2023).

[147] S. Gunasekar, Y. Zhang, J. Aneja, C. C. T. Mendes, A. Del Giorno, S. Gopi, M. Javaheripi, P. Kauffmann, G. de Rosa, O. Saarikivi, et al. "Textbooks Are All You Need". In: *arXiv preprint arXiv:2306.11644* (2023).

[148] Y. Li, S. Bubeck, R. Eldan, A. Del Giorno, S. Gunasekar, and Y. T. Lee. "Textbooks Are All You Need II: phi-1.5 technical report". In: *arXiv preprint arXiv:2309.05463* (2023).

[149] R. Taori, I. Gulrajani, T. Zhang, Y. Dubois, X. Li, C. Guestrin, P. Liang, and T. B. Hashimoto. *Stanford Alpaca: An Instruction-following LLaMA model*. 2023.

[150] W.-L. Chiang, Z. Li, Z. Lin, Y. Sheng, Z. Wu, H. Zhang, L. Zheng, S. Zhuang, Y. Zhuang, J. E. Gonzalez, I. Stoica, and E. P. Xing. *Vicuna: An Open-Source Chatbot Impressing GPT-4 with 90%* ChatGPT Quality*. Mar. 2023.

[151] C. Xu, Q. Sun, K. Zheng, X. Geng, P. Zhao, J. Feng, C. Tao, and D. Jiang. "Wizardlm: Empowering large language models to follow complex instructions". In: *arXiv preprint arXiv:2304.12244* (2023).

[152] E. Zelikman, Y. Wu, J. Mu, and N. Goodman. "STaR: Bootstrapping Reasoning With Reasoning". In: *Advances in Neural Information Processing Systems*. Ed. by A. H. Oh, A. Agarwal, D. Belgrave, and K. Cho. 2022.

[153] L. H. Li, J. Hessel, Y. Yu, X. Ren, K.-W. Chang, and Y. Choi. "Symbolic Chain-of-Thought Distillation: Small Models Can Also "Think" Step-by-Step". In: *Proceedings of the 61st Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*. Toronto, Canada: Association for Computational Linguistics, July 2023, pp. 2665–2679.

[154] M. Sclar, P. West, S. Kumar, Y. Tsvetkov, and Y. Choi. "Referee: Reference-Free Sentence Summarization with Sharper Controllability through Symbolic Knowledge Distillation". In: *arXiv preprint arXiv:2210.13800* (2022).

[155] H. Luo, Q. Sun, C. Xu, P. Zhao, J. Lou, C. Tao, X. Geng, Q. Lin, S. Chen, and D. Zhang. "Wizardmath: Empowering mathematical reasoning for large language models via reinforced evol-instruct". In: *arXiv preprint arXiv:2308.09583* (2023).

[156] X. Yue, X. Qu, G. Zhang, Y. Fu, W. Huang, H. Sun, Y. Su, and W. Chen. "MAmmoTH: Building Math Generalist Models through Hybrid Instruction Tuning". In: *arXiv preprint arXiv:2309.05653* (2023).

[157] S. Zhang, Z. Chen, Y. Shen, M. Ding, J. B. Tenenbaum, and C. Gan. "Planning with large language models for code generation". In: *arXiv preprint arXiv:2303.05510* (2023).

[158] H. Le, H. Chen, A. Saha, A. Gokul, D. Sahoo, and S. Joty. "CodeChain: Towards Modular Code Generation Through Chain of Self-revisions with Representative Sub-modules". In: *arXiv preprint arXiv:2310.08992* (2023).

[159] N. Jain, S. Vaidyanath, A. Iyer, N. Natarajan, S. Parthasarathy, S. Rajamani, and R. Sharma. "Jigsaw: Large Language Models meet Program Synthesis". In: *ICSE 2022*. Pittsburgh, Pennsylvania.

[160] Y. Lai, C. Li, Y. Wang, T. Zhang, R. Zhong, L. Zettlemoyer, S. W.-t. Yih, D. Fried, S. Wang, and T. Yu. "DS-1000: A Natural and Reliable Benchmark for Data Science Code Generation". In: *ArXiv* abs/2211.11501 (2022).

[161] D. F. Zhiruo Wang Shuyan Zhou and G. Neubig. "Execution-Based Evaluation for Open-Domain Code Generation". In: (2022).

[162] P. Shojaee, A. Jain, S. Tipirneni, and C. K. Reddy. "Execution-based Code Generation using Deep Reinforcement Learning". In: *Transactions on Machine Learning Research* (2023).

[163] J. Liu, Y. Zhu, K. Xiao, Q. Fu, X. Han, W. Yang, and D. Ye. *RLTF: Reinforcement Learning from Unit Test Feedback*. 2023. arXiv: 2307.04349 [cs.AI].

[164] C. Liu, X. Bao, H. Zhang, N. Zhang, H. Hu, X. Zhang, and M. Yan. "Improving ChatGPT Prompt for Code Generation". In: *arXiv preprint arXiv:2305.08360* (2023).

[165] M. Luo, N. Jain, J. Singh, S. Tan, A. Patel, Q. Wu, A. Ariyak, C. Cai, T. V. Suhail, I. Stoica, M. Zaharia, and R. A. Popa. *DeepSWE: Training a Fully Open-sourced, State-of-the-Art Coding Agent by Scaling RL*. Together AI Blog. Blog post, published 2025-07-02. July 2025.

[166] N. Jain, J. Singh, M. Shetty, L. Zheng, K. Sen, and I. Stoica. "R2E-Gym: Procedural Environment Generation and Hybrid Verifiers for Scaling Open-Weights SWE Agents". In: *Conference on Language Modeling (COLM)*. 2025.

[167] Qwen Team. *Qwen3 Technical Report*. Technical report. 2025.

[168] DeepSeek-AI. *DeepSeek-R1-Zero: Incentivizing Reasoning Capability in LLMs via Reinforcement Learning*. Technical report. 2025.

[169] . *Understanding R1-Zero-like Training: A Critical Perspective (Dr.GRPO)*. arXiv preprint. 2025.

[170] . "DAPO: Decoupled Clip and Dynamic Sampling Policy Optimization". In: *arXiv preprint* (2024). If a similar entry exists in other paper bibs under a different key, this entry is intentionally preserved so the DeepSWE chapter citations resolve.

[171] A. Ahmadian et al. "Back to Basics: Revisiting REINFORCE-Style Optimization for Learning from Human Feedback in LLMs (RLOO/LOOP)". In: *arXiv preprint* (2024).

[172] C. S. Xia, Y. Deng, S. Dunn, and L. Zhang. "Agentless: Demystifying llm-based software engineering agents". In: *arXiv preprint arXiv:2407.01489* (2024).

[173] K. Cobbe, V. Kosaraju, M. Bavarian, M. Chen, H. Jun, L. Kaiser, M. Plappert, J. Tworek, J. Hilton, R. Nakano, et al. "Training verifiers to solve math word problems". In: *arXiv preprint arXiv:2110.14168* (2021).

[174] Y. Wei, O. Duchenne, J. Copet, Q. Carbonneaux, L. Zhang, D. Fried, G. Synnaeve, R. Singh, and S. I. Wang. "SWE-RL: Advancing LLM Reasoning via Reinforcement Learning on Open Software Evolution". In: *arXiv preprint arXiv:2502.18449* (2025).

[175] Y. Ma, R. Cao, Y. Cao, Y. Zhang, J. Chen, Y. Liu, Y. Liu, B. Li, F. Huang, and Y. Li. "Lingma swe-gpt: An open development-process-centric language model for automated software improvement". In: *arXiv preprint arXiv:2411.00622* (2024).

[176] C. S. Xia, Y. Deng, S. Dunn, and L. Zhang. *Agentless: Demystifying LLM-based Software Engineering Agents*. https://github.com/OpenAutoCoder/Agentless. 2024.

[177] K. Zhang, W. Yao, Z. Liu, Y. Feng, Z. Liu, R. RN, T. Lan, L. Li, R. Lou, J. Xu, et al. "Diversity empowers intelligence: Integrating expertise of software engineering agents". In: *The Thirteenth International Conference on Learning Representations*. 2024.

[178] J. P. Inala, C. Wang, M. Yang, A. Codas, M. Encarnación, S. Lahiri, M. Musuvathi, and J. Gao. "Fault-aware neural code rankers". In: *Advances in Neural Information Processing Systems* 35 (2022), pp. 13419–13432.

[179] A. Gu, W.-D. Li, N. Jain, T. X. Olausson, C. Lee, K. Sen, and A. Solar-Lezama. "The Counterfeit Conundrum: Can Code Language Models Grasp the Nuances of Their Incorrect Generations?" In: *arXiv preprint arXiv:2402.19475* (2024).

[180] M. Shetty, N. Jain, A. Godbole, S. A. Seshia, and K. Sen. "Syzygy: Dual Code-Test C to (safe) Rust Translation using LLMs and Dynamic Analysis". In: *arXiv preprint arXiv:2412.14234* (2024).

[181] L. Szekeres, M. Payer, T. Wei, and D. Song. "SoK: Eternal War in Memory". In: *2013 IEEE Symposium on Security and Privacy*. 2013, pp. 48–62.

[182] google. *google/zopfli*. Online. 2024.

[183] DARPA. *TRACTOR: Translating All C to Rust*. Online. 2024.

[184] Immunant Inc. *immunant/c2rust*. Online. 2020.

[185] A. Z. H. Yang, Y. Takashima, B. Paulsen, J. Dodds, and D. Kroening. "VERT: Verified Equivalent Rust Transpilation with Large Language Models as Few-Shot Learners". In: *arXiv preprint arXiv: 2404.18852* (2024).

[186] H. Zhang, C. David, Y. Yu, and M. Wang. "Ownership guided C to Rust translation". In: *International Conference on Computer Aided Verification* (2023).

[187] M. Shiraishi and T. Shinagawa. "Context-aware Code Segmentation for C-to-Rust Translation using Large Language Models". In: *arXiv preprint arXiv: 2409.10506* (2024).

[188] T. Zhou, H. Lin, S. Jha, M. Christodorescu, K. Levchenko, and V. Chandrasekaran. "LLM-Driven Multi-step Translation from C to Rust using Static Analysis". In: *arXiv preprint arXiv:2503.12511* (2025).

[189] V. Nitin, R. Krishna, L. L. d. Valle, and B. Ray. "C2SaferRust: Transforming C Projects into Safer Rust with NeuroSymbolic Techniques". In: *arXiv preprint arXiv:2501.14257* (2025).

[190] A. Albarghouthi, S. Gulwani, and Z. Kincaid. "Recursive program synthesis". In: *Computer Aided Verification: 25th International Conference, CAV 2013, Saint Petersburg, Russia, July 13-19, 2013. Proceedings 25*. Springer. 2013, pp. 934–950.

[191] H. Li, L. Guo, Y. Yang, S. Wang, and M. Xu. "An Empirical Study of Rust-for-Linux: The Success, Dissatisfaction, and Compromise". In: *2024 USENIX Annual Technical Conference (USENIX ATC 24)*. Santa Clara, CA: USENIX Association, July 2024, pp. 425–443.

[192] K. Serebryany, D. Bruening, A. Potapenko, and D. Vyukov. "AddressSanitizer: A Fast Address Sanity Checker". In: *USENIX ATC 2012*. 2012.

[193] N. Nethercote and J. Seward. "How to shadow every byte of memory used by a program". In: *Proceedings of the 3rd International Conference on Virtual Execution Environments*. VEE '07. San Diego, California, USA: Association for Computing Machinery, 2007, pp. 65–74.

[194] OpenBSD Project. *unifdef: remove preprocessor conditionals from code*. https://man.openbsd.org/unifdef. n.d.

[195] M. L. Nelson. "A Survey of Reverse Engineering and Program Comprehension". In: *arXiv preprint* (2005).

[196] B. Brown, J. Juravsky, R. Ehrlich, R. Clark, Q. V. Le, C. Ré, and A. Mirhoseini. "Large language monkeys: Scaling inference compute with repeated sampling". In: *arXiv preprint arXiv:2407.21787* (2024).

[197] Immunant Inc. *Emitting Safer Rust with C2Rust*. Online. 2024.

[198] Galois Inc. *Function Argument Nullability Using an LLM*. Online. 2024.

[199] H. F. Eniser, H. Zhang, C. David, M. Wang, M. Christakis, B. Paulsen, J. Dodds, and D. Kroening. "Towards translating real-world code with llms: A study of translating to rust". In: *arXiv preprint arXiv:2405.11514* (2024).

[200] B. Rozière, M. Lachaux, L. Chanussot, and G. Lample. "Unsupervised Translation of Programming Languages". In: *NeurIPS*. 2020.

[201] B. Rozière, J. Zhang, F. Charton, M. Harman, G. Synnaeve, and G. Lample. "Leveraging Automated Unit Tests for Unsupervised Code Translation". In: *ICLR*. OpenReview.net, 2022.

[202] S. Tipirneni, M. Zhu, and C. K. Reddy. "StructCoder: Structure-Aware Transformer for Code Generation". In: *ACM Trans. Knowl. Discov. Data* 18.3 (Jan. 2024).

[203] M. Szafraniec, B. Roziere, H. L. F. Charton, P. Labatut, and G. Synnaeve. "Code translation with Compiler Representations". In: *ICLR* (2023).

[204] Z. Tang, M. Agarwal, A. Shypula, B. Wang, D. Wijaya, J. Chen, and Y. Kim. "Explain-then-translate: an analysis on improving program translation with self-generated explanations". In: *Findings of the Association for Computational Linguistics: EMNLP 2023*. Association for Computational Linguistics, Dec. 2023, pp. 1741–1788.

[205] M. Jiao, T. Yu, X. Li, G. Qiu, X. Gu, and B. Shen. "On the Evaluation of Neural Code Translation: Taxonomy and Benchmark". In: *Automated Software Engineering (ASE)*. IEEE, Sept. 2023, pp. 1529–1541.

[206] P. Jana, P. Jha, H. Ju, G. Kishore, A. Mahajan, and V. Ganesh. "Attention, Compilation, and Solver-based Symbolic Analysis are All You Need". In: *arXiv preprint arXiv:2306.06755* (2023).

[207] X. Yin, C. Ni, T. N. Nguyen, S. Wang, and X. Yang. "Rectifier: Code Translation with Corrector via LLMs". In: *CoRR* abs/2407.07472 (2024). arXiv: 2407.07472.

[208] A. R. Ibrahimzada, K. Ke, M. Pawagi, M. S. Abid, R. Pan, S. Sinha, and R. Jabbarvand. "Repository-Level Compositional Code Translation and Validation". In: *arXiv preprint arXiv: 2410.24117* (2024).

[209] H. Zhang, C. David, M. Wang, B. Paulsen, and D. Kroening. "Scalable, Validated Code Translation of Entire Projects using Large Language Models". In: *arXiv preprint arXiv:2412.08035* (2024).

[210] T. Ball. "The concept of dynamic analysis". In: *SIGSOFT Softw. Eng. Notes* 24.6 (Oct. 1999), pp. 216–234.

[211] E. Andreasen, L. Gong, A. Møller, M. Pradel, M. Selakovic, K. Sen, and C.-A. Staicu. "A Survey of Dynamic Analysis and Test Generation for JavaScript". In: *ACM Comput. Surv.* 50.5 (Sept. 2017).

[212] M. D. Ernst, J. H. Perkins, P. J. Guo, S. McCamant, C. Pacheco, M. S. Tschantz, and C. Xiao. "The Daikon system for dynamic detection of likely invariants". In: *Science of Computer Programming* (2007).

[213] S. Hangal, N. Chandra, S. Narayanan, and S. Chakravorty. "IODINE: a tool to automatically infer dynamic invariants for hardware designs". In: *Proceedings of the 42nd Annual Design Automation Conference*. DAC '05. Anaheim, California, USA: Association for Computing Machinery, 2005, pp. 775–778.

[214] J. Hong and S. Ryu. "Don't Write, but Return: Replacing Output Parameters with Algebraic Data Types in C-to-Rust Translation". In: *Proc. ACM Program. Lang.* 8.PLDI (June 2024).

[215] A. Fromherz and J. Protzenko. "Compiling C to Safe Rust, Formalized". In: *arXiv preprint arXiv: 2412.15042* (2024).

[216] R. Alur, R. Bodik, G. Juniwal, M. M. K. Martin, M. Raghothaman, S. A. Seshia, R. Singh, A. Solar-Lezama, E. Torlak, and A. Udupa. "Syntax-guided synthesis". In: *2013 Formal Methods in Computer-Aided Design*. 2013, pp. 1–8.

[217] A. Solar-Lezama, L. Tancau, R. Bodik, S. Seshia, and V. Saraswat. "Combinatorial sketching for finite programs". In: *SIGARCH Comput. Archit. News* 34.5 (Oct. 2006), pp. 404–415.

[218] S. Bhatia, S. Kohli, S. A. Seshia, and A. Cheung. "Building Code Transpilers for Domain-Specific Languages Using Program Synthesis (Experience Paper)". In: *37th European Conference on Object-Oriented Programming, ECOOP 2023, July 17-21, 2023, Seattle, Washington, United States*. Ed. by K. Ali and G. Salvaneschi. Vol. 263. LIPIcs. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2023, 38:1–38:30.