

Compiler-in-the-Loop: Decomposing the Value of Static Verification for Low-Resource Code Generation

Stanford CS224N Custom Project

Hlumelo Notshe
Department of Computer Science
Stanford University
hnotshe@stanford.edu

Joshua Martinez
Department of Computer Science
Stanford University
joshuakm@stanford.edu

Abstract

Large language models achieve strong code generation performance in high-resource languages like Python but struggle with low-resource languages such as Rust, where data scarcity compounds the difficulty of strict static typing and ownership semantics. We investigate how much of the value in synthetic training data comes from compiler-based filtering alone versus full test-suite verification. Using a generate-and-filter pipeline, we produce synthetic Rust solutions from two generators – CodeLlama-7B-Instruct (self-generation) and DeepSeek-Coder-6.7B-Instruct – filter them at three verification levels (unfiltered, compiler-only, full pipeline), and fine-tune CodeLlama-7B-Instruct with LoRA on each resulting dataset. On the MultiPL-E Rust benchmark, our best model improves pass@1 from 19.2% to 25.3% and compilation rate from 58.3% to 69.6%, with gains driven primarily by eliminating Rust-specific compile failures such as duplicate scaffolding and API hallucinations. A detailed error taxonomy reveals that filtering addresses surface-level Rust mistakes but leaves deeper challenges – type-level reasoning, ownership semantics, and algorithmic correctness – largely intact. Our results suggest that compiler-based filtering provides meaningful but bounded value for low-resource code generation, and that the choice of data generator interacts strongly with the effect of filtering.

1 Key Information

- Mentor: Arpandeeep Khatua
- External Collaborators: Simon Guo (simonguo@stanford.edu)
- Sharing project: No
- Late Days: None used

2 Introduction

Large language models have achieved impressive performance on code generation tasks in high-resource languages like Python, where abundant training data enables models to learn both algorithmic reasoning and language-specific idioms. However, performance drops sharply for low-resource programming languages such as Rust, CUDA, Verilog, and FORTRAN – languages that underpin critical infrastructure in systems programming, hardware design, and scientific computing Cassano et al. [2023]. This gap is particularly consequential because these languages often have strict compilers or type systems that catch entire classes of errors statically, yet LLMs routinely produce outputs that fail at these basic checks.

The standard approach to overcoming data scarcity is synthetic data generation: use a capable model to produce training examples, filter them for quality, and fine-tune a smaller model on the results.

For Python, filtering typically relies on test suites – executing candidates and keeping those that pass. But test suites require human curation and are not always available, limiting scalability. Statically-typed, compiler-strict languages offer an alternative quality signal that is both free and universal: *the compiler itself*. Rust’s compiler enforces ownership semantics, borrow checking, lifetime constraints, and type safety at compile time, rejecting exactly the categories of errors that LLMs produce most frequently.

Our central research question is: **How much of the value in synthetic data filtering comes from the compiler alone, versus test suites?** We decompose the verification pipeline into its constituent stages – compilation and test passage – and measure each one’s independent contribution to downstream model quality. If compiler-only filtering captures most of the gain, this has significant practical implications: synthetic data pipelines for low-resource code generation could work with any language that has a strict compiler, without requiring curated test suites.

We use Rust as our case study because it has (a) a notoriously strict compiler that catches a rich class of errors statically, (b) well-established benchmarks with test suites enabling controlled experiments, and (c) sufficient practical importance to make results immediately relevant. As an additional axis of investigation, we compare two data generators of different capability – CodeLlama-7B-Instruct (self-generation) and DeepSeek-Coder-6.7B-Instruct – to examine how generator quality interacts with the effect of filtering.

3 Related Work

Code generation with LLMs. The Codex model Chen et al. [2021] established function-level code generation as a standard evaluation paradigm through the HumanEval benchmark. Code Llama Rozière et al. [2023] extended this line of work with open-weight models specialized for code, though its evaluation notably omits Rust from the MultiPL-E results – itself an indicator of the language’s low-resource status.

Synthetic data and filtering for code. WizardCoder Luo et al. [2023] demonstrated that synthetic data generated by a stronger model can substantially improve a weaker model’s coding ability through supervised fine-tuning. MultiPL-T Cassano et al. [2024] showed that LLM-based translation from Python to low-resource languages, combined with compiler filtering, can improve performance on underserved languages. Our work differs from MultiPL-T in that we generate solutions directly in Rust rather than translating from Python, and we explicitly decompose the filtering pipeline to isolate the compiler’s contribution.

RL from verifier feedback. CodeRL Le et al. [2022] frames code generation as an RL problem using unit test outcomes as reward. While CodeRL treats the verifier as a black-box pass/fail signal, we decompose that signal into its components and measure each one’s contribution at the *data filtering* level. This connects to the broader concept of the asymmetry of verification Wei [2025]: generating correct code is hard, but verifying it with a compiler is cheap and deterministic.

4 Approach

4.1 Overview

Our method follows a generate-and-filter pipeline (Figure 1). For each training problem, we generate multiple candidate Rust solutions using a generator model, tag each candidate by its verification status, and construct three training sets of matched size from this single pool. We then fine-tune CodeLlama-7B-Instruct on each set with identical training configuration, so the *only* variable across conditions is filter quality.

Our primary contribution is the *decomposition* of the verification pipeline: by training separate models on unfiltered, compiler-only, and fully verified data, we isolate the marginal value of each filtering stage. A secondary contribution is comparing self-generation (generator = student) against cross-model generation (generator $\neq$ student) as data sources, revealing an interaction between generator identity and filtering effectiveness.

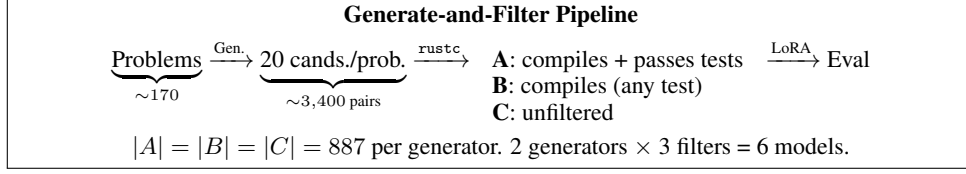


Figure 1: Experimental pipeline. Each generator produces candidates tagged by verification status, then split into three matched training sets. All models share identical hyperparameters; only data differs.

4.2 Generator pivot

Our original plan used a frontier model (Claude Sonnet 4) as the candidate generator. In a pilot study, it achieved $\sim 99\%$ full-pipeline yield, eliminating the variance across verification levels our experiment requires. We therefore pivoted to two peer-scale generators: (1) **CodeLlama-7B-Instruct (self-generation)**, where the generator and student are the same model, cleanly isolating the effect of data filtering from distillation at $\sim 15\%$ full-pipeline yield; and (2) **DeepSeek-Coder-6.7B-Instruct**, an independently trained code model of similar size, introducing a mild distillation dynamic to test whether filtering effects generalize across generators.

4.3 Verification hierarchy and dataset construction

For each of approximately 170 training problems, we generate 20 candidate solutions per generator, yielding roughly 3,400 raw (problem, solution) pairs per generator. Every candidate is tagged by the Rust toolchain into one of four levels: (1) does not parse, (2) parses but does not compile, (3) compiles but fails tests, or (4) compiles and passes all tests.

From this single pool, we construct three training sets per generator:

Table 1: Training data conditions. Per-problem sampling ensures identical size and problem distribution across sets; only filter quality varies.

Set	Filter	What it isolates
A – Full pipeline	Compiles + passes tests	Upper bound on filtering value
B – Compiler-only	Compiles (any test outcome)	Value of static verification alone
C – Unfiltered	None (raw generations)	Baseline synthetic data quality

Per-problem matched sampling. For each training problem, we count how many full-pipeline examples exist in Set A, then sample exactly that many compiler-only candidates for Set B and that many raw candidates for Set C. Problems with zero full-pipeline candidates are excluded from all three sets. This ensures identical size, per-problem counts, and problem distribution; the only variable is filter quality.

Both generators yielded 887 matched examples per set across 103 training problems (56 excluded due to zero full-pipeline yield). For the DeepSeek Set C, the 887 unfiltered examples break down as: 527 passing tests, 64 compiling but failing tests, 276 failing compilation, and 20 failing to parse – confirming meaningful separation between filter levels.

4.4 Fine-tuning

All models are fine-tuned from CodeLlama-7B-Instruct using LoRA Hu et al. [2022] with rank 16, $\alpha = 32$, and dropout 0.05, applied to all four attention projection matrices (q_proj, k_proj, v_proj, o_proj). We train for 3 epochs with a per-device batch size of 4, gradient accumulation over 4 steps (effective batch size 16), learning rate 2×10^{-4} with cosine scheduling, warmup ratio 0.05, weight decay 0.01, and maximum sequence length 2048. All training uses FP16 mixed precision with gradient checkpointing. The input format is an instruction-style prompt containing the problem description and function signature; the target is the candidate solution. Training configuration is identical across all six conditions (3 filter levels $\times$ 2 generators); only the data differs.

5 Experiments

5.1 Data

We use two problem sources: (1) **MultiPL-E Rust** Cassano et al. [2023] (~ 160 problems from the multilingual HumanEval extension; ~ 110 train, 5 val, 50 test), and (2) **Exercism Rust track** (~ 95 exercises emphasizing Rust-specific idioms such as ownership, pattern matching, and iterators; ~ 60 train, 5 val, 35 test). Exercism serves as an out-of-distribution evaluation set since its problems emphasize language-specific patterns not well represented in HumanEval.

5.2 Evaluation method

We built a fully automated evaluation harness: extract code from model output $\rightarrow$ compile with `rustc` (2021 edition, 30s timeout) $\rightarrow$ run test suite (60s timeout) $\rightarrow$ score. The harness was validated on known-correct solutions before running any evaluations.

We generate $n = 20$ samples per problem and report:

- **pass@ k** : the probability that at least one of k samples passes all tests, estimated unbiasedly following Chen et al. [2021]:

$$\text{pass@}k = \mathbb{E}_{\text{problems}} \left[1 - \frac{\binom{n-c}{k}}{\binom{n}{k}} \right] \quad (1)$$

where n is the total number of samples and c is the number of correct samples for a given problem. We report pass@1, pass@5, and pass@10.

- **Compilation rate**: the fraction of generated outputs that successfully compile, independent of test outcomes. This metric is particularly relevant to our central question, as it directly reflects the compiler’s filtering effect.

We evaluate on four splits: MultiPL-E val (5 problems), MultiPL-E test (50 problems), Exercism val (5 problems), and Exercism test (35 problems).

5.3 Experimental conditions

We train six SFT models (3 filter levels $\times$ 2 generators) plus the zero-shot baseline, for a total of seven conditions. We adopt the following shorthand: **CL-A/B/C** for CodeLlama-generated Sets A/B/C, and **DS-A/B/C** for DeepSeek-generated Sets A/B/C.

5.4 Results

Main results. Table 2 presents results on the two primary test splits. On MultiPL-E test, the best model overall is **CL-A** (CodeLlama self-generation, full pipeline), which improves pass@1 from 19.2% to 25.3% (+6.1 pp), compilation rate from 58.3% to 69.6% (+11.3 pp), and pass@10 from 49.7% to 53.7% (+4.0 pp). The ordering $A > B > C$ is consistent for CodeLlama on this split, confirming that each filtering stage adds value.

On Exercism test (out-of-distribution), the picture is more nuanced. The CodeLlama variants improve compilation rate substantially (from 35.4% to approximately 41–42%), and CL-B achieves the best pass@1 (24.3% vs. 20.6% baseline). However, the baseline retains a small edge on pass@10 (57.9%), suggesting better tail-sample diversity on this split.

Generator comparison. The most striking finding is that **filtering helps much more for CodeLlama-generated data than for DeepSeek-generated data**. On MultiPL-E test, CL-A materially improves over both the baseline and its own unfiltered variant, while DS-A is essentially tied with the baseline on pass@1 (19.1% vs. 19.2%). On Exercism test, DeepSeek-generated models consistently underperform the baseline, with the unfiltered variant DS-C falling to just 11.6% pass@1.

We attribute this asymmetry to the self-generation property of the CodeLlama pipeline. When the generator and student are the same model, filtered training data directly corrects the model’s own

Table 2: Results on held-out test splits. Best value in each row is **bolded**. CL = CodeLlama generator, DS = DeepSeek generator. A = full pipeline, B = compiler-only, C = unfiltered.

Split	Metric	CodeLlama gen.			DeepSeek gen.			Base
		CL-A	CL-B	CL-C	DS-A	DS-B	DS-C	
ME test	Compile %	69.6	69.2	63.3	62.4	62.2	55.9	58.3
	pass@1	25.3	19.5	17.3	19.1	19.1	18.6	19.2
	pass@10	53.7	49.6	43.6	51.5	45.4	49.0	49.7
Ex. test	Compile %	40.3	41.4	41.9	32.3	29.4	23.0	35.4
	pass@1	22.7	24.3	24.3	18.9	14.7	11.6	20.6
	pass@10	56.4	57.0	54.8	49.2	46.4	43.4	57.9

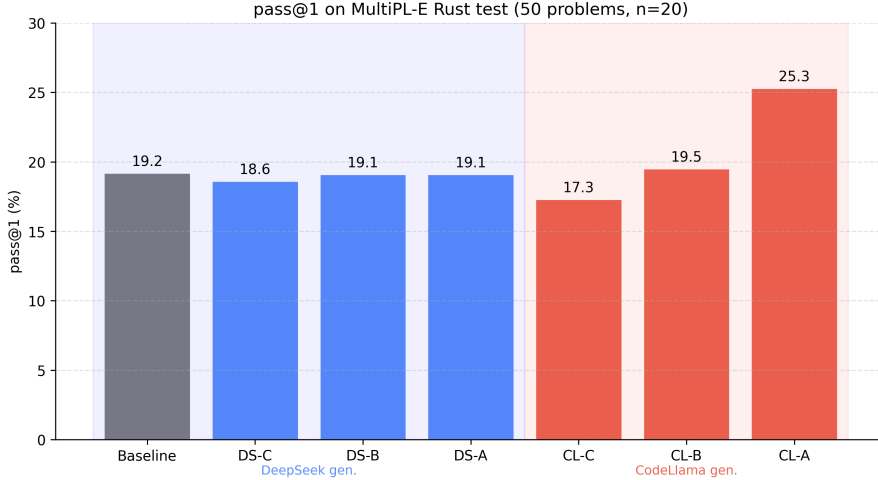


Figure 2: pass@1 on MultiPL-E Rust test across all seven conditions. Only CL-A (CodeLlama self-generation with full-pipeline filtering) materially improves over baseline (+6.1pp); all DeepSeek variants remain within noise of the baseline.

failure modes – the training signal is maximally relevant. With DeepSeek as generator, the student must adapt to a different model’s coding style, and the distillation overhead may dilute the filtering signal.

Filtering hierarchy within CodeLlama. On the primary MultiPL-E test split (Figure 4), the CodeLlama conditions show a clear ordering: CL-A (full pipeline) > CL-B (compiler-only) > CL-C (unfiltered) on both pass@1 and pass@10. The gap between CL-A and CL-B (+5.8 pp on pass@1) is larger than the gap between CL-B and CL-C (+2.2 pp), suggesting that test-suite filtering contributes more marginal value than the compiler alone on this benchmark. On Exercism, the ordering is less clean: CL-B matches or slightly beats CL-A on pass@1, possibly because the Exercism problems emphasize Rust-specific compilation patterns where compiler-only filtering is sufficient.

Discussion. Two results were surprising. First, the full-pipeline CodeLlama model (CL-A) improved pass@1 more than we expected (+6.1 pp from only 887 training examples), suggesting that even modest amounts of high-quality, compiler-verified data can meaningfully shift a 7B model’s Rust behavior. Second, we initially hypothesized that the compiler would capture *most* of the filtering value, since Rust’s compiler is unusually strict. Instead, the A vs. B gap (+5.8 pp) exceeded the B vs. C gap (+2.2 pp) on MultiPL-E test, indicating that test suites provide substantial value beyond compilation – likely because many candidates compile but contain subtle algorithmic errors that only tests can catch. The DeepSeek results were also unexpected: we anticipated that data from a different model of similar capability would transfer well, but the consistent underperformance suggests that generator–student alignment matters more than raw data quality for SFT at this scale.

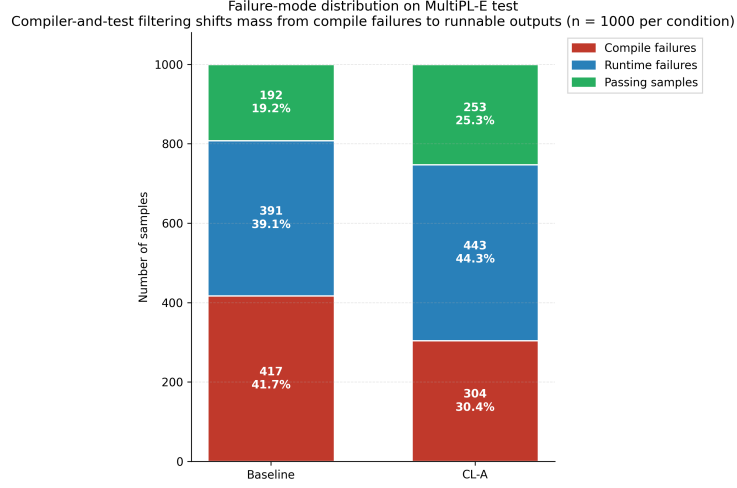


Figure 3: Failure-mode distribution on MultiPL-E test ($n = 1000$ per condition). Compiler-and-test filtering shifts mass from compile failures to runnable outputs: compile failures drop from 41.7% to 30.4%, while passing samples increase from 19.2% to 25.3%.

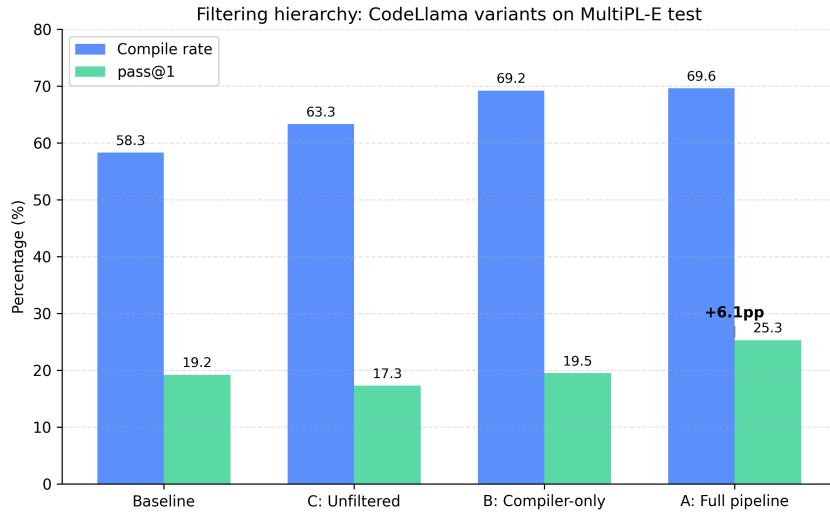


Figure 4: Filtering hierarchy for CodeLlama variants on MultiPL-E test. Compile rate improves steadily from baseline through each filtering stage, but the largest pass@1 gain (+6.1pp) requires full-pipeline filtering (Set A), indicating that test-suite verification contributes substantial value beyond the compiler alone.

6 Analysis

6.1 Failure mode decomposition

To understand *where* improvements come from, we decompose all 1,000 samples on MultiPL-E test (50 problems $\times$ 20 samples) into three categories: compile failures, runtime failures (compiled but did not pass), and passing samples. Figure 3 shows this breakdown for the baseline and CL-A.

The gain is driven more by reducing compile failures than by reducing semantic mistakes. Baseline produces 417 compile failures, 391 runtime failures, and 192 passing samples; CL-A shifts that to 304, 443, and 253 respectively. Many formerly uncompileable programs become runnable but still incorrect, which is why the runtime failure count *increases* – a natural consequence of more candidates surviving compilation.

Table 3: Compile error categories on MultiPL-E test (counts out of 1,000 samples per model).

Category	Baseline	CL-A	Reading
Type / numeric	143	146	Persistent; filtering does not solve type reasoning
Trait / method halluc.	77	50	Reduced; fewer hallucinated Rust APIs
Import resolution	58	43	Partially reduced
Ownership / move	32	29	Persistent; borrow semantics remain hard
Duplicate scaffolding	30	0	Eliminated ; cleanest filtering win
Mutability	20	15	Small improvement
Syntax / parse	3	3	Rare in both

6.2 Rust error taxonomy

We categorized compile errors from the first useful line of `compile_stderr` across all 1,000 baseline and 1,000 CL-A samples. Table 3 shows the main categories.

What filtering fixes. The clearest win is **duplicate scaffolding**: the baseline frequently emits extra main functions causing E0428 failures; this pattern completely disappears in CL-A. Filtering also reduces **trait and method hallucinations** – e.g., calling `.is_integer()` on `f64` or `.sorted_by()` on an iterator – and some import-resolution errors.

What filtering does not fix. **Type and numeric reasoning** (E0308 mismatches, E0277 invalid indexing) remains the dominant compile family at similar counts in both models. **Ownership and borrow semantics** (E0382, E0507) also persist largely unchanged. These categories represent deeper Rust reasoning beyond what ~ 900 filtered examples can teach.

6.3 Problem-level contrast analysis

Comparing problem-level outcomes (solved vs. unsolved based on any passing sample out of 20) between baseline and CL-A on MultiPL-E test: 25 solved by both, 3 only by CL-A, 3 only by baseline, 19 by neither. The small number of exact flips (6 total) reflects distributed rather than concentrated improvements. A characteristic pattern emerges: several problems move from *Rust failure* to *algorithmic failure* (e.g., `HumanEval_160_do_algebra` goes from 1/20 to 11/20 compiling, but 0/20 passing in both cases), confirming that filtering addresses surface-level Rust errors while algorithmic reasoning remains the dominant residual bottleneck. Full contrast set details appear in Appendix C.

6.4 Self-generation vs. cross-model generation

The contrast between generators reveals that self-generation (CodeLlama generating its own training data) consistently outperforms cross-model generation (DeepSeek generating data for CodeLlama), despite both generators being of similar scale. On Exercism test, DS-C achieves only 11.6% pass@1 versus CL-C’s 24.3%. We hypothesize that self-generated data is more effective because it directly targets the model’s own failure distribution: candidates that happen to pass verification represent solutions the model was *almost* capable of producing, converting near-misses into clean training signal.

7 Conclusion

We decomposed the value of static verification for synthetic code generation data in Rust by training separate models on unfiltered, compiler-only, and fully verified data from two generators. Our results support three main conclusions.

First, **compiler-and-test filtering provides meaningful improvements** over unfiltered synthetic data, with the best configuration (CL-A) achieving a 6.1 percentage point improvement in pass@1 on MultiPL-E test. The gains are driven primarily by eliminating Rust-specific compilation failures – duplicate scaffolding, API hallucinations, and some import resolution errors.

Second, **filtering has bounded value**: it addresses surface-level Rust mistakes but does not solve deeper challenges in type-level reasoning, ownership semantics, or algorithmic correctness. Many

problems move from “Rust failure” to “algorithmic failure” rather than to solved, indicating that compilation is a necessary but insufficient condition for correctness.

Third, **the choice of generator matters as much as the choice of filter**. Self-generation produces more effective training data than cross-model generation of similar scale, likely because it targets the student model’s own failure distribution. This has practical implications: for low-resource language improvement, a model generating and filtering its own training data may be more effective than distilling from a different model of comparable capability.

Limitations and future work. Our study is limited to a single base model size (7B), a single target language (Rust), and relatively modest dataset sizes (~ 900 examples from ~ 100 problems). We did not explore the RLVR extension due to time constraints. Natural extensions include scaling to larger models, applying the decomposition to other compiler-strict languages (Haskell, Ada), and exploring shaped compiler rewards to address the type-level and ownership errors that filtering alone cannot fix.

Team Contributions

Hlumelo Notshe: Evaluation harness, data generation pipeline, filter implementation, training runs, error analysis, and report writing. **Joshua Martinez:** Problem curation, Exercism dataset preparation, DeepSeek generation runs, baseline evaluation, figure generation, and report writing. Both authors contributed equally to experimental design and analysis.

References

- Federico Cassano, John Gouwar, Daniel Nguyen, Sydney Nguyen, Luna Phipps-Costin, Donald Pinckney, Ming-Ho Yee, Yangtian Zi, Carolyn Jane Anderson, Molly Q Feldman, et al. MultiPLE: A scalable and polyglot approach to benchmarks for code generation. *IEEE Transactions on Software Engineering*, 2023.
- Mark Chen, Jerry Tworek, Heewoo Jun, Qiming Yuan, Henrique Ponde de Oliveira Pinto, Jared Kaplan, Harri Edwards, Yuri Burda, Nicholas Joseph, Greg Brockman, et al. Evaluating large language models trained on code. *arXiv preprint arXiv:2107.03374*, 2021.
- Baptiste Rozière, Jonas Gehring, Fabian Gloeckle, Sten Sootla, Itai Gat, Xiaoqing Ellen Tan, Yossi Adi, Jingyu Liu, Romain Sauvestre, Tal Remez, et al. Code llama: Open foundation models for code. *arXiv preprint arXiv:2308.12950*, 2023.
- Ziyang Luo, Can Xu, Pu Zhao, Qingfeng Sun, Xiubo Geng, Wenxiang Hu, Chongyang Tao, Jing Ma, Qingwei Lin, and Daxin Jiang. WizardCoder: Empowering code large language models with Evol-Instruct. *arXiv preprint arXiv:2306.08568*, 2023.
- Federico Cassano, John Gouwar, Francesca Lucchetti, Claire Schlesinger, Anders Freeman, Carolyn Jane Anderson, Molly Q Feldman, Michael Greenberg, Abhinav Jangda, and Arjun Guha. Knowledge transfer from high-resource to low-resource programming languages for code LLMs. *arXiv preprint arXiv:2308.09895*, 2024.
- Hung Le, Yue Wang, Akhilesh Deepak Gotmare, Silvio Savarese, and Steven C. H. Hoi. CodeRL: Mastering code generation through pretrained models and deep reinforcement learning. In *Advances in Neural Information Processing Systems*, 2022.
- Jason Wei. Asymmetry of verification and verifier’s law, 2025. Blog post.
- Edward J Hu, Yelong Shen, Phillip Wallis, Zeyuan Allen-Zhu, Yuanzhi Li, Shanen Wang, Lu Wang, and Weizhu Chen. LoRA: Low-rank adaptation of large language models. *arXiv preprint arXiv:2106.09685*, 2022.

A Full Results Across All Splits

Table 4 presents complete results across all four evaluation splits for all seven conditions.

Table 4: Complete results across all evaluation splits. Best value per row is **bolded**.

Split	Metric	Base	DS-A	DS-B	DS-C	CL-A	CL-B	CL-C
ME val	Compile %	90.0	62.0	66.0	66.0	92.0	97.0	85.0
	pass@1	79.0	51.0	55.0	62.0	81.0	82.0	70.0
	pass@10	100.0	99.7	99.1	99.9	100.0	100.0	99.1
ME test	Compile %	58.3	62.4	62.2	55.9	69.6	69.2	63.3
	pass@1	19.2	19.1	19.1	18.6	25.3	19.5	17.3
	pass@10	49.7	51.5	45.4	49.0	53.7	49.6	43.6
Ex. val	Compile %	38.0	36.0	41.0	38.0	62.0	69.0	68.0
	pass@1	24.0	11.0	16.0	14.0	33.0	36.0	32.0
	pass@10	50.0	53.1	63.2	69.1	50.0	59.1	60.0
Ex. test	Compile %	35.4	32.3	29.4	23.0	40.3	41.4	41.9
	pass@1	20.6	18.9	14.7	11.6	22.7	24.3	24.3
	pass@10	57.9	49.2	46.4	43.4	56.4	57.0	54.8

B Detailed Error Taxonomy

Table 5 presents runtime failure categories for baseline and CL-A on MultiPL-E test.

Table 5: Runtime failure categories on MultiPL-E test.

Category	Baseline	CL-A	Reading
Logic / assertion	350	391	More candidates compile, more reach wrong-answer assertions
Panic / overflow	5	18	CL-A gets farther but hits Rust numeric edge cases
Panic / index	6	3	Slightly better in CL-A
Timeout	4	9	More compiled candidates $\Rightarrow$ more chances for slow solutions

C Problem-Level Contrast Set

Table 6 presents a sampled set of 10 illustrative problems from the 16-problem contrast set, showing how outcomes shift between baseline and CL-A. Notation: c/p = compile-ok / test-pass out of 20 samples.

Table 6: Selected problem-level contrasts between baseline and CL-A on MultiPL-E test.

Problem	Direction	Base c/p	CL-A c/p	Main reading
HumanEval_84_solve	CL-A win	11/0	17/6	Higher compile & pass rate
HumanEval_151_double_diff	CL-A win	4/0	10/4	Escapes f64 halluc.
HumanEval_144_simplify	CL-A win	14/0	4/1	Rare correct sample
HumanEval_101_words_string	Base win	18/3	17/0	Semantic regression
HumanEval_120_maximum	Base win	7/1	8/0	Indexing regression
HumanEval_160_do_algebra	Rust $\rightarrow$ Algo	1/0	11/0	Compiles, still wrong
HumanEval_145_order_by_pts	Rust $\rightarrow$ Algo	2/0	13/0	Past compiler, not tests
HumanEval_97_multiply	Rust $\rightarrow$ Algo	9/0	16/0	More runnable, still wrong
HumanEval_111_histogram	Both fail	0/0	0/0	Persistent HashMap issue
HumanEval_138_is_equal...	Both fail	20/0	20/0	Pure semantic bottleneck