Learning to Solve and Verify: A Self-Play Framework for Mutually Improving Code and Test Generation

Zi Lin[*]
UC San Diego
lzi@ucsd.edu

Sheng Shen
xAI
shengs@x.ai

Jingbo Shang
UC San Diego
jshang@ucsd.edu

Jason Weston
Meta
jase@meta.com

Yixin Nie
Meta
ynie@meta.com

Abstract

Recent breakthroughs in Large Language Models (LLMs) have significantly advanced code generation. However, further progress is increasingly constrained by the limited availability of high-quality supervised data. Synthetic data generation via self-instruction shows potential, but naive approaches often suffer from error accumulation and generalization collapse, underscoring the critical need for robust quality control. This paper introduces SOL-VER, a novel self-play framework where an LLM simultaneously acts as a *solver* (generating code) and a *verifier* (generating tests). These two capabilities are mutually enhanced: improved tests lead to better code, which in turn enables the generation of more discerning tests. SOL-VER iteratively refines both code solutions and their corresponding unit tests, jointly improving both functionalities without requiring human annotations or larger, more capable teacher models. Our experiments using Llama 3.1 8B demonstrate substantial gains, achieving average relative improvements of 19.63% in code generation (pass@1) and 17.49% in test generation accuracy on the MBPP and LiveCodeBench benchmarks.

1 Introduction

Large language models (LLMs) have demonstrated impressive ability in code generation, significantly enhancing the programming efficiency and productivity of human developers [19, 23, 4]. The ability to code is largely due to high-quality online coding resources, e.g., coding problem and human-rewritten solutions. However, as these supervised data sources saturate, LLM improvement is diminishing, with data scarcity becoming a key bottleneck for further progress.

To address scarce supervised data for code generation, recent studies use synthetic data techniques like SELF-INSTRUCT [26] to augmenting LLM training sets. Typically, a high-capacity teacher LLM generates code responses to designed instructions, and this data is then employed to fine-tune a student LLM, thereby enhancing its code generation abilities. Although synthetic code data produced in this manner has demonstrated success, it relies on the availability of a strong teacher model, presumably with a larger parameter size and higher computation costs. Additionally, existing work has shown that training a model on data generated by itself is ineffective because errors introduced during generation tend to accumulate over iterations [10]. As a result, there is a critical need for effective methods to verify the generated data.

[*]Work done during internship at Meta.

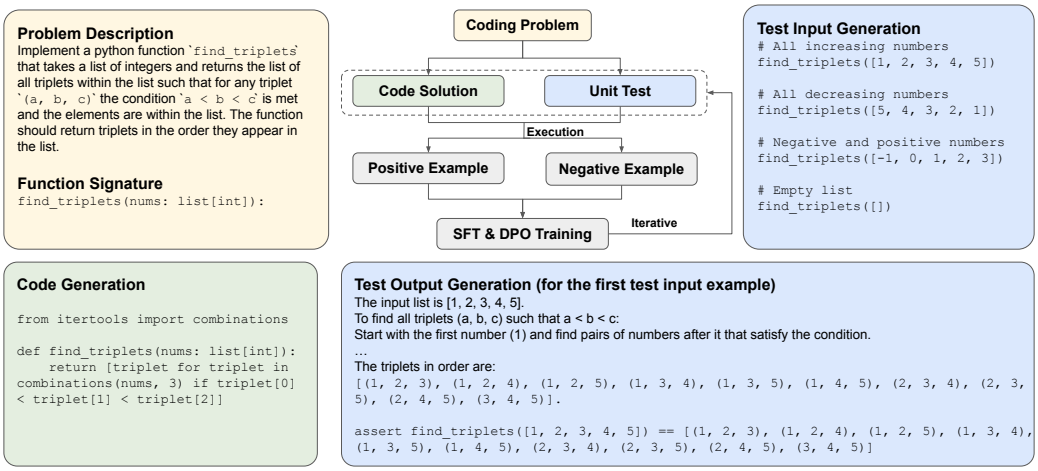

Figure 1: An overview of the SOL-VER framework. We train an LLM to both generate coding solutions (solver) and unit tests (verifier) in an iterative self-play framework, whereby synthetic preference pairs are constructed at each iteration depending on whether the code passes the generated tests or not. We show that this approach enables the model to self-improve in both capabilities (see Table 1).

While evaluating generated code correctness is challenging, often requiring expert human intervention, recent LLM-as-a-judge efforts involve models executing generated code against self-generated unit tests [20, 2, 9, 6, 5, 7, 10]. However, a critical bottleneck emerges: an LLM's proficiency as a *verifier* (generating effective unit tests) significantly lags its capability as a *solver* (generating code solutions), a disparity we quantify in Section 4.3). This gap is largely attributable to the scarcity of high-quality, diverse unit test generation data used during LLM fine-tuning, which predominantly focuses only on code generation.

To address this imbalance and unlock a new avenue for data generation, we introduce SOL-VER, a self-play solver-verifier framework to iteratively train a model for both code and test generation. The main idea is to let the LLM-as-a-solver and LLM-as-a-verifier help each other. Specifically, we ask the model to generate code solutions and unit tests for the same set of coding problems. By executing the generated test against the generated code, we obtain feedback for training, involving two steps: (1) SFT training: we take the passed examples for fine-tuning the model, and (2) DPO training: we take both passed and failed examples as preference pairs to further train the model aligning with the preference. These training steps are for both code generation and unit test generation, and they can be repeated in an iterative manner.

The experimental results on Llama 3.1 8B model show that we can successfully improve the model's performance on both code and test generation without relying on human-annotated data or larger models. Specifically, on MBPP and LiveCodeBench, we achieve an average of 19.63% and 17.49% relative improvement for code and test generation respectively.

In summary, our work makes the following contributions:

- **Identification of a *critical gap*:** We empirically demonstrate and analyze the significant gap in LLMs' abilities between code generation and unit test generation, motivating the need for targeted improvements in test generation.
- **Novel *Self-Play* Framework:** We propose a novel iterative framework where the model simultaneously functions as a code *solver* and a *verifier*. This methodology effectively self-aligns the model's outputs with desired performance criteria without relying on external annotations or teacher models.
- **High-Quality Synthetic Data Generation:** We contribute a generalizable method for creating high-quality synthetic data for both code and unit test generation. This data augmentation approach can be extended to various model training scenarios in the coding domain.

2 Related Work

In recent years, scaling laws highlight the critical role of data size in training foundation models [14, 12, 8], making LLM-driven synthetic data generation a popular solution. Methods like SELF-INSTRUCT [26, 24] use pre-trained LLMs to create instruction-output pairs from seed data. For code generation, this often involves stronger teacher models generating synthetic instructions to finetune weaker student models (e.g., CODEALPACA [23]). Efforts to enhance LLM coding abilities include generating more complex instructions (e.g., *Code Eval-Instruct* [28]).

However, training LLMs on their own generated data can be ineffective or detrimental [31, 1]. Consequently, post-processing or refinement steps are vital. Examples include CodeT's [5] execution-based validation, Self-Debug's [7] autonomous bug fixing, Llama 3.1's [10] iterative self-correction with execution feedback, Reinforcement learning from unit test feedback (RLTF) for optimizing against test pass rates [16, 21], and CodeDPO's [30] self-generation and validation for preference data. While similar to our work, AutoIF [18] also uses an iterative framework for general tasks, it doesn't focus on the co-evolution of code solver and verifier capabilities.

In this work, we propose leveraging both positive and negative examples generated by the model, treating pairs of passing and failing responses as chosen-rejected pairs for Direct Preference Optimization (DPO) [22]. Note that our method is complementary to self-correction and RLTF, rather than orthogonal. By improving the quality of unit tests, our framework enhances the accuracy of unit test execution feedback, and thereby can benefit self-correction and RLTF scenarios as well.

SOL-VER **compared to CodeDPO:** While CodeDPO [30] also utilizes self-verification for preference data, its efficacy is inherently limited by the initial, and potentially static, quality of the self-generated tests used for verification. If the verifier component is weak, it cannot reliably distinguish high-quality code, potentially leading to suboptimal preference learning. In contrast, SOL-VER's self-play mechanism is designed to explicitly address this: it **simultaneously improves both the code solver and the test verifier**. By iteratively enhancing the verifier's ability to generate more discerning and comprehensive tests, SOL-VER breaks this quality ceiling, thereby mitigating the verifier bottleneck and enabling more robust and effective training for both capabilities.

3 A Self-play Solver-verifier Framework

3.1 Problem Formulation

We consider that an LLM can play two roles.

Solver (S): Given a coding problem description P, it produces a candidate solution C (e.g., a piece of code). The objective of the solver is to produce a correct solution C that will pass any tests the verifier can come up with.

Verifier (V): Given a proposed solution C and the original problem P, the verifier tries to produce test cases $\mathbf{T}$[2] (e.g., a set of inputs and expected outputs) and can catch errors in C if it is incorrect. Essentially, it produces and selects challenging unit tests to determine if the code is correct or not. The objective of the verifier is to produce a set of tests $\mathbf{T}$ that will fail any incorrect solutions and thus distinguish correct solutions from incorrect ones.

Let $p(\mathbf{P})$ be the distribution over problem statements. We can think of having a training set of problems or a domain from which we can sample problems. The sampling strategies we consider are detailed in Section 3.2.

The solver S_θ is a model parameterized by θ that, given a problem P, generates a candidate solution C: $C \sim S_\theta(\cdot|P)$. The verifier V_ϕ is a model parameterized by ϕ, given the problem P and a candidate solution C, generates a test suite $\mathbf{T}$: $\mathbf{T} \sim V_\phi(\cdot|P, C)$. In practice, the solver and verifier can be the same LLM.

Scoring Function We define a function that executes C on the tests $\mathbf{T}$ as $\text{Score}(C, \mathbf{T}) \in [0, 1]$, which is the fraction of tests passed by solution C. A score of 1 means C passes every test T; a score of 0 means it failed all tests. Formally,

[2]We use bold Italic to represent a set.

$$\text{Score}(C, \mathbf{T}) = \mathbb{E}_{T \sim \mathbf{T}}[\mathbb{I}(C(T) = \text{expected\_output}(T))] \tag{1}$$

where $\mathbb{I}$ is the indicator function, and $C(T)$ means running one single test on code solution C.

We sample a set of problems $P \sim p(\mathbf{P})$, generate some candidate solutions $C \sim S_\theta(\cdot|P)$, and generate candidate tests $\mathbf{T} \sim V_\phi(\cdot|P, C)$. We now have tuple $(P, C, \mathbf{T})$. We consider:

$$y = \begin{cases} 1 & \text{if Score}(C, \mathbf{T}) = 1 \text{ (i.e., passes all tests)} \\ 0 & \text{otherwise.} \end{cases} \tag{2}$$

We employ two stages of training to make use of both chosen ($y = 1$) and rejected ($y = 0$) examples for training the solver and verifier, described as follows:

Stage 1: SFT Training For pairs where $y = 1$, we have a correct solution-test pair. These are high-quality examples that reflect desired behavior, i.e., the solution C solves the problem P, and the test suite $\mathbf{T}$ properly validates that the solution is correct. We use $(P, C, \mathbf{T}, y = 1)$ tuples to fine-tune the model directly. The training signal here encourages the model (1) as a solver, to generate similar correct solutions for similar problems, and (2) as a verifier, to produce meaningful tests that confirm correctness. We call this the supervised fine-tuning (SFT) stage, where we optimize for both solver and verifier:

$$\mathcal{L}_{\text{SFT}_{\text{solver}}}(\theta) = -\mathbb{E}_{(P,C,\mathbf{T}):y=1}[\log S_\theta(C|P)] \tag{3}$$
$$\mathcal{L}_{\text{SFT}_{\text{verifier}}}(\phi) = -\mathbb{E}_{(P,C,\mathbf{T}):y=1}[\log V_\phi(\mathbf{T}|P, C)] \tag{4}$$

In practice, both objectives can be trained using a mixture of data consisting of chosen examples for solver and verifier.

Stage 2: DPO Training We now aim to form pairwise comparisons (preferences) to train both solver and verifier roles more effectively. In practice, we adopted the Direct Preference Optimization (DPO) method, but any preference tuning methods can be used at this stage.

For the solver perspective, for each problem P, and each chosen tuple $(P, C^+, \mathbf{T}, y = 1)$, we find a rejected tuple $(P, C^-, \mathbf{T}, y = 0)$. Following standard DPO training [22], we can formulate our policy objective as:

$$\mathcal{L}_{\text{DPO}_{\text{solver}}}(S_\theta^*; S_\theta) = -\mathbb{E}[\log \sigma(\beta \log \frac{S_\theta^*(C^+|P)}{S_\theta(C^+|P)} - \beta \log \frac{S_\theta^*(C^-|P)}{S_\theta(C^-|P)}] \tag{5}$$

where β is the hyperparameter to regulate the strength of weight updates; $S_\theta(C|P)$ is the probability that our model (with parameter θ) assigns to generating code solution C given problem P.

For the verifier perspective, similarly, for each problem P, and each chosen tuple $(P, C, \mathbf{T}^+) : y = 1$, find a rejected tuple $(P, C, \mathbf{T}^-) : y = 0$[3]. The verifier-related DPO loss is then:

$$\mathcal{L}_{\text{DPO}_{\text{verifier}}}(V_\phi^*; V_\phi) = -\mathbb{E}[\log \sigma(\beta \log \frac{V_\phi^*(\mathbf{T}^+|P, C)}{V_\phi(\mathbf{T}^+|P, C)} - \beta \log \frac{V_\phi^*(\mathbf{T}^-|P, C)}{V_\phi(\mathbf{T}^-|P, C)}] \tag{6}$$

3.2 Synthetic Data Generation

In this section, we describe our approach to generating the synthetic data, including problem description generation, code generation, test generation and preference data generation. All related prompts can be found in Appendix A.

Problem Description Generation Inspired by Magicoder [28], we prompt the model with random code snippets to generate diverse programming problems, including long-tail topics. This allows

[3]This can be achieved by selecting any expected output that is not equal to the chosen one in the sampling space for T.

us to tap into a wide range of topics and create a comprehensive set of problem descriptions (as demonstrated in Figure 5).

It is noted that code generation tasks can follow different problem description formats despite having the same content. For example, here is the same problem but stated in different ways:

- Write a python function to remove the kth element from a given list.
- In the ancient Library of Alexandria, scrolls are stored in a mystical list. The librarian needs to remove a specific scroll whenever a visitor requests it.

To make our prompt sets accommodate these diverse problem description formats, we adopt some templates from the training set of different coding benchmarks (e.g., MBPP, APPS) into the original prompt to generate the problem description.

To create self-contained coding problems, we ensure clear statements and starter code with necessary libraries and detailed input/output function signatures. The model first generates problem descriptions, then adds these function signatures. After deduplication, this yielded 103,280 problem descriptions.

Code Generation We prompt the LLM to solve each problem given the generated function signature. Following the Llama 3.1 [10], we also require the model to explain its thought process in comments, which improves code generation in both accuracy and readability.

Test Generation To generate unit test sets, we first ask the model to generate a set of valid inputs and then the expected outputs based on the inputs.

For input generation, we ask the model to generate different types of function inputs to cover different cases including general, corner or difficult cases. For example, the following problem description should contain two different cases:

Problem Description:
Write a function to find the longest string in a list of strings. If the strings are not comparable due to being of different lengths, the function should return None.
`longest_string(strings: list[str])`
Case 1: `strings` is a list of strings.
Case 2: `strings` is empty.

For output generation, we ask the model to generate the expected output based on the problem description and input. We also applied majority voting [25] and chain-of-thought reasoning [27] for LLM to boost the performance of unit test generation (as demonstrated in Section 4.3):

We also employ the following strategies to ensure the quality of the generated unit tests:

- **Test Coverage Optimization:** We sample multiple candidate test cases and select a subset that maximizes branch coverage of the solution code (a maximum coverage problem), ensuring diversity of execution paths.
- **Output Diversification:** We notice that if the outputs of the test cases are not diverse, the solution code can cheat by exploiting patterns in test cases. For example, if all test cases return the same value (e.g., `True`), the model could trivially pass by implementing a function that always returns that value. Therefore, we explicitly select test samples with diverse output values.

Synthetic Preference Data Generation For DPO preference tuning, we construct chosen/rejected pairs. As detailed in Section 3.1,"chosen" examples are where the solver's generated code passes the verifier's generated tests. Identifying "rejected" examples is more complex, as disagreements don't clearly assign fault.

While [9] used automated quality cross-verification, we apply a similar strategy for both code and test generation. Specifically, if at least one generated solution passes all the generated tests, we will assume the solution and the test are "correct". For training the solver, any other sampled code solutions that fail to pass these "correct" tests are treated as "rejected" examples. Similarly, for training the verifier, when we find a "correct" test $f(x) == y$, we revisit the original sampling space for generating expected outputs, and treat any expected outputs $y' \neq y$ as "rejected" tests $f(x) == y'$. In this way, we can reuse all the chain-of-thought explanations generated by the model during the majority voting process.

Table 1: Evaluation results over training iterations for our Solver-Verifier (SOL-VER) method. $\Delta(\%)$ means the relative percentage change from Baseline to $\text{Iter3}_{\text{+DPO}}$. Pass% means average pass rate, Acc% is accuracy, and FP% is false positive rate. SOL-VER provides large gains over the baseline on both code generation (solver) and unit test generation (verifier) tasks, which increase across iterations.

		Baseline	**Iter1$_{\text{+SFT}}$**	**Iter1$_{\text{+DPO}}$**	**Iter2$_{\text{+SFT}}$**	**Iter2$_{\text{+DPO}}$**	**Iter3$_{\text{+SFT}}$**	**Iter3$_{\text{+DPO}}$**	**$\Delta(\%)$**
MBPP	$\text{Code}_{\text{Pass\%}}$	38.60	38.60	40.80	38.60	40.80	40.80	41.00	6.22
	$\text{Test}_{\text{Acc\%}}$	42.68	49.01	49.50	50.69	51.01	51.54	51.76	17.54
	$\text{Test}_{\text{FP\%}}$	12.75	12.57	10.32	10.12	10.06	9.80	9.60	24.71
LiveCodeBench	$\text{Code}_{\text{Pass\%}}$	18.23	24.86	25.97	25.12	26.38	26.41	27.24	33.08
	$\text{Test}_{\text{Acc\%}}$	20.14	36.43	37.09	39.40	40.65	41.25	41.50	51.47
	$\text{Test}_{\text{FP\%}}$	20.76	20.65	19.34	19.28	19.05	18.94	18.63	10.26

4 Experiments

4.1 Experimental Setup

Models and Datasets We conduct experiments using Llama 3.1 8B [10].[4] For the SFT and DPO training, we use the fairseq2 infrastructure [29] and run inference using vLLM [15].

We sample the problem descriptions using Llama 3.1 based on open source snippets from the OSS-Instruct dataset [28][5]. For the problem description templates, we use the training sets of some standard coding benchmark: MBPP [3], APPS [11] and CodeContest [19]. For our experiments, we only focus on Python-related coding questions.

We test both code and unit test generation on the python coding benchmarks including:

- MBPP [3]: A popular benchmark for Python code generation which focuses on relatively simple, self-contained functions.
- LiveCodeBench [13]: A comprehensive and contamination-free evaluation of LLMs for code, which continuously collects new problems over time from contests across three competition platforms, namely LeetCode, AtCoder, and CodeForces.

LiveCodeBench contains both code generation and test output prediction tasks. Since MBPP does not originally include a unit test generation task, we utilize the existing gold unit tests to assess the model's accuracy in generating expected outputs given the inputs in gold unit tests.

Evaluation Metrics For the code generation task, all results are obtained using greedy decoding with the pass@1 metric (Pass%). For unit test generation, we consider the following metrics:

- Accuracy (Acc%): The accuracy of test output prediction. A test output is correct only when its literal value is equal to the gold one.
- False Positive Rate (FP%): This measures how well unit tests differentiate correct from incorrect code by evaluating the pass rate of known flawed solutions. These flawed solutions are obtained by generating 20 candidate codes per problem and using gold unit tests to identify failures (treated as negative examples).[6]

4.2 Evaluating SOL-VER

Table 1 reports SOL-VER's iterative training performance for code and test generation on MBPP and LiveCodeBench.[7] To further show the improvement trend for SOL-VER, we plot the performance change across iterations for test generation in Figure 2 (other figures for iterative change can be found in Appendix B). Results demonstrate SOL-VER consistently improves the base Llama 3.1 model in both tasks, increasing code pass rates and test generation accuracy while decreasing false positives.

[4] `https://huggingface.co/meta-llama/Llama-3.1-8B`

[5] `https://huggingface.co/datasets/ise-uiuc/Magicoder-OSS-Instruct-75K`

[6] We use the same Llama 3.1 8B base model to generate negative examples (temperature is set to 0.6 and top p is set to 0.9). As a result, we get 400 examples for both MBPP and LiveCodeBench separately.

[7] Our baseline performance for code generation differs from that reported in the Llama 3.1 technical report because we employ a unified three-shot prompt template for both MBPP and LiveCodeBench, rather than using the prompts specifically provided for MBPP. We do not include the gold test in the prompt.

Table 2: Code generation and test generation performance for Llama 3.1 8B on the MBPP benchmark. The case pass rate represents the average pass rate per test set. CoT means Chain-of-Thought reasoning, and MV means majority voting.

Llama 3.1 8B	**Pass Rate**	**Case Pass Rate**
Code Generation	38.60%	44.20%
Test Generation	18.60%	37.60%
+ CoT	19.60%	39.60%
+ MV	24.80%	42.40%
+ CoT + MV	31.40%	47.60%
Code Reranked by Synthetic Test	35.00%	42.40%

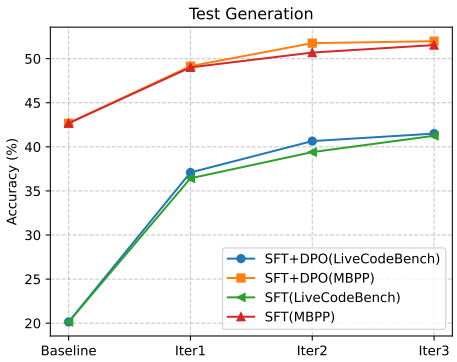

Figure 2: Iterative performance of our method, SOL-VER, for test generation. Our method outperforms the baseline and SFT training for both LiveCodeBench and MBPP benchmarks, and improve across training iterations.

We achieved average relative improvements of 19.63% (code) and 17.49% (test). In particular, two interesting conclusions can be made:

- The improvement in unit test generation performance is more significant than that observed in code generation. This greater enhancement may be due to that Llama 3.1's pre-training included less data focused on tests.
- SFT+DPO outperforming SFT-only models suggests DPO, by incorporating negative examples, helps the model learn from errors and refine its generation strategies.

4.3 Evaluating the Base Model

To contextualize SOL-VER's improvements and underscore the motivation for our work, we first evaluated the baseline Llama 3.1 8B model's capabilities on MBPP (Table 2, first two rows). This initial assessment immediately revealed the critical performance disparity that SOL-VER aims to address: the model achieved a Pass@1 of 38.60% for code generation (solver) when evaluated against gold tests. However, when tasked with generating test outputs for gold code solutions (verifier), its accuracy (Case Pass Rate) was only 37.60% (with a raw pass rate of 18.60% for individual test cases), confirming that the LLM's inherent test generation abilities significantly lag its code generation prowess.

We then explored conventional prompting strategies to enhance baseline test generation, applying Chain-of-Thought (CoT) reasoning and Majority Voting (MV) (Section 3.2). As shown in Table 2, these techniques, particularly when combined (+CoT+MV), did improve test generation accuracy to 31.40% (Pass Rate). This demonstrates that while sophisticated prompting can provide some uplift, it does not fully bridge the solver-verifier gap.

Critically, we investigated the impact of using these baseline-generated tests (even the improved CoT+MV versions) to rerank code solutions – a common approach in self-improvement. The last row

Table 3: Agreement between Iter 1 and Iter 2, and the test accuracy for the model ensemble. The ensemble approach (Ens.) refers to selecting code solutions that successfully pass both the tests generated in the first iteration and those from the second iteration.

Dataset	**Acc%$_{\text{Iter1}}$**	**Acc%$_{\text{Iter2}}$**	**Agreemnt**	**Acc%$_{\text{Ens.}}$**
MBPP	49.50%	51.01%	75.14%	51.12%
LiveCodeBench	37.09%	40.65%	72.38%	40.68%

Table 4: Illustrative examples of test case refinement by SOL-VER's verifier component across iterations for two distinct programming problems. Red highlighting indicates tests that were incorrect or suboptimal in earlier iterations but were corrected or improved by later iterations, demonstrating the verifier's learning progress.

Problem	Write a function that takes an integer number of seconds as input and returns the number of minutes in that time, disregarding any remaining seconds.	Write a function that calculates and returns the greatest common divisor (GCD) of two integers using the Euclidean algorithm, with the output formatted as "GCD(a, b)= result".

Iter 1	`assert minutes_in(1) == 0`
	`assert minutes_in(3660 + 60 + 60 + 1) == 3`
	`assert minutes_in(3660 + 60 + 60 + 60 + 60 + 1) == 5`
		`assert pgcd(8, 7) == 1`
		`assert pgcd(20, 25) == 5`
		`assert pgcd(14, 6) == 2`

Iter2	`assert minutes_in(1) == 0`
	`assert minutes_in(3660 + 60 + 60 + 1) == 61`
	`assert minutes_in(3660 + 60 + 60 + 60 + 60 + 1) == 61`
		`assert pgcd(8, 7) == "PGCD(8,7) = 1"`
		`assert pgcd(20, 25) == "PGCD(20,25) = 5"`
		`assert pgcd(14, 6) == "PGCD(14,6) = 2"`

Iter3	`assert minutes_in(1) == 0`
	`assert minutes_in(3660 + 60 + 60 + 1) == 63`
	`assert minutes_in(3660 + 60 + 60 + 60 + 60 + 1) == 65`
		`assert pgcd(8, 7) == "GCD(8, 7) = 1"`
		`assert pgcd(20, 25) == "GCD(20, 25) = 5"`
		`assert pgcd(14, 6) == "GCD(14, 6) = 2"`

of Table 2 ("Code Reranked by Synthetic Test") shows that this led to a **decrease** in code generation performance (35.00% Pass@1) compared to the original baseline (38.60%). This finding is crucial: it highlights that naively using imperfect, self-generated tests for code refinement can be detrimental. This underscores the necessity of a framework like SOL-VER, which doesn't just use tests, but actively **improves** them in tandem with code generation, preventing such performance degradation and instead fostering mutual improvement.

4.4 Evaluating Agreement between Iterations

To monitor progress, as models from different iterations are trained on distinct synthetic data, we examine their agreement. Specifically, both models generate unit tests for MBPP benchmarks, measuring agreement on gold code solutions (Table 3). Both iterations exhibit performance enhancements and maintain high agreement, indicating that despite distinct synthetic training data, they produce largely consistent unit test outputs.

Given the agreement between iterations, we evaluated an ensemble that selects code solutions successfully passing tests from both the first and second iterations. This ensemble demonstrated modest improvement over the second iteration alone, leading us to adopt it for generating third-iteration synthetic data for SOL-VER, as reported in Table 1.

5 Ablation Study

5.1 Prompt and Test Analysis

Prompt Coverage Analysis To analyze the domain coverage of our generated coding problem set, in Figure 5 at Appendix C, we visualize the embedding distributions of our synthetic problem descriptions with those from established coding benchmarks, including MBPP, APPS, and LiveCodeBench. Specifically, we use Gecko, a compact and versatile text embedding model distilled from LLMs [17] for obtaining the sentence embeddings. The embedding distribution plot reveals that our synthetic problem set exhibits a broad and diverse coverage, encompassing a wide range of topics, difficulty levels, and programming paradigms present in the compared benchmarks.

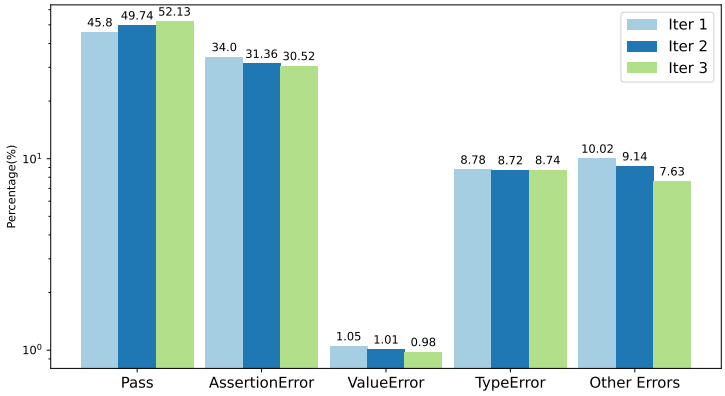

Figure 3: Pass or error distribution of synthetic data generated at each iteration.

Table 5: Code generation performance for different settings of the scoring function for selecting DPO pairs. Results indicate better results are obtained with less, but higher quality, data ($\epsilon > 0$).

ϵ	> 0	> 0.5	> 0.75	SOL-VER
Data Size	25,525	20,457	13,158	12,525
MBPP	36.00	37.00	**40.80**	**40.80**
LiveCodeBench	22.41	25.75	**26.18**	25.97
Score(C^-, T)	Random	Lowest	Median	SOL-VER
MBPP	**40.80**	39.00	38.60	**40.80**
LiveCodeBench	**26.18**	**26.18**	25.98	25.97

Progress Analysis per Iteration To evaluate the iterative advancements of our model, we present a case study on test generation across iterations in Table 4. The results illustrate that SOL-VER progressively refines its test generation for the same set of coding problems, thereby enhancing the quality of the synthetic data. These enhancements include the generation of more accurate expected values and better adherence to required format specifications. Additionally, we monitor the execution results at each iteration and display the distribution of pass and error rates in Figure 3. As shown, the pass rate increases with each iteration, primarily due to a reduction in assertion errors, indicating an improvement in the accuracy of the predicted expected outputs.

5.2 Discussion on the Scoring Function

In Section 3.1, we define a binary scoring function for selecting solution-test pairs. Our Iteration 1 experiments found that only 45% of examples yielded "agreed" pairs (where at least one generated code passed all generated tests), thereby limiting the total number of available preference tuning pairs.

To explore whether we can utilize the rest of the data where the pass rate is not necessarily 100%, but is still high enough to rely on, we used a *soft pass rate* for selecting preference pairs. We change the chosen / rejected pair as (C^-, C^+), where for the same test suite $\mathbf{T}$, Score$(C^+, \mathbf{T}) >$ Score$(C^-, \mathbf{T})$, and Score$(C^+, \mathbf{T}) \geq \epsilon$, where ϵ is a threshold to determine above which pass rate the test set is relatively reliable. For simplicity, we discuss three cases for ϵ: (1) $\epsilon > 0$ (can be any number); (2) $\epsilon > 0.5$; (3) $\epsilon > 0.75$. For Score$(C^-, \mathbf{T})$, we also consider three cases: (1) Score$(C^-, \mathbf{T})$ is a random score; (2) Score$(C^-, \mathbf{T})$ is the lowest score among all sampling candidate; (3) Score$(C^-, \mathbf{T})$ is the median score from the lowest to Score$(C^+, \mathbf{T})$.

In Table 5, we first present various ϵ settings for assigning Score$(C^-, \mathbf{T})$ as a random score. After identifying the optimal setting from our results ($\epsilon > 0.75$), we examine the impact on Score$(C^-, \mathbf{T})$. The findings reveal that randomly selecting the threshold for a high-quality test set significantly degrades performance, despite an increase in data size. This underscores that the quality of synthetic data is more critical than its quantity. Regarding Score$(C^-, \mathbf{T})$, we found that its impact is less

sensitive compared to ϵ. Considering the overall performance, we have chosen to retain the original settings for SOL-VER to maintain simplicity.

6 Conclusion

We introduced SOL-VER, a novel self-play framework designed to address the critical challenge of data scarcity and the performance disparity between LLM-based code solvers and test verifiers. By enabling an LLM to simultaneously embody both roles, SOL-VER facilitates a co-evolutionary process where improvements in test generation lead to better code synthesis, and vice-versa. This iterative refinement loop generates high-quality synthetic code-test pairs, demonstrably enhancing both capabilities without reliance on human annotations or larger teacher models. Our experiments with Llama 3.1 8B validate SOL-VER's efficacy, achieving significant performance gains on established benchmarks. SOL-VER represents a step towards more autonomous, data-efficient, and robust systems for automated code and test generation, offering a scalable and adaptable paradigm for continuous self-improvement in code AI.

References

[1] Sina Alemohammad, Josue Casco-Rodriguez, Lorenzo Luzi, Ahmed Imtiaz Humayun, Hossein Babaei, Daniel LeJeune, Ali Siahkoohi, and Richard G Baraniuk. The curse of recursion: Training on generated data makes models forget. *arXiv preprint arXiv:2305.17493*, 2023.

[2] Nadia Alshahwan, Jubin Chheda, Anastasia Finogenova, Beliz Gokkaya, Mark Harman, Inna Harper, Alexandru Marginean, Shubho Sengupta, and Eddy Wang. Automated unit test improvement using large language models at meta. In *Companion Proceedings of the 32nd ACM International Conference on the Foundations of Software Engineering*, pages 185–196, 2024.

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

[4] Sahil Chaudhary. Code alpaca: An instruction-following llama model for code generation. `https://github.com/sahil280114/codealpaca`, 2023.

[5] Bei Chen, Fengji Zhang, Anh Nguyen, Daoguang Zan, Zeqi Lin, Jian-Guang Lou, and Weizhu Chen. Codet: Code generation with generated tests. *arXiv preprint arXiv:2207.10397*, 2022.

[6] Mouxiang Chen, Zhongxin Liu, He Tao, Yusu Hong, David Lo, Xin Xia, and Jianling Sun. B4: Towards optimal assessment of plausible code solutions with plausible tests. In *Proceedings of the 39th IEEE/ACM International Conference on Automated Software Engineering*, pages 1693–1705, 2024.

[7] Tianlan Chen, Zhiye Zhang, Zelin Wang, Yihong Liu, Shankai Zhang, Yuxiang Wang, Bill Yuchen Lin, Peidong Wang, Bei Yin, Yefan Lu, et al. Self-debug: Autonomous self-debugging for large language models in code generation. *arXiv preprint arXiv:2304.05128*, 2023.

[8] Hyung Won Chung, Le Hou, Shayne Longpre, Barret Zoph, Yi Tay, William Fedus, Yunxuan Li, Xuezhi Wang, Mostafa Dehghani, Siddhartha Brahma, et al. Scaling instruction-finetuned language models. *arXiv preprint arXiv:2210.11416*, 2022.

[9] Guanting Dong, Keming Lu, Chengpeng Li, Tingyu Xia, Bowen Yu, Chang Zhou, and Jingren Zhou. Self-play with execution feedback: Improving instruction-following capabilities of large language models. *arXiv preprint arXiv:2406.13542*, 2024.

[10] Abhimanyu Dubey, Abhinav Jauhri, Abhinav Pandey, Abhishek Kadian, Ahmad Al-Dahle, Aiesha Letman, Akhil Mathur, Alan Schelten, Amy Yang, Angela Fan, et al. The llama 3 herd of models. *arXiv preprint arXiv:2407.21783*, 2024.

[11] Dan Hendrycks, Steven Basart, Saurav Kadavath, Mantas Mazeika, Akul Arora, Ethan Guo, Collin Burns, Samir Puranik, Horace He, Dawn Song, and Jacob Steinhardt. Measuring coding challenge competence with apps. *NeurIPS*, 2021.

[12] Jordan Hoffmann, Sebastian Borgeaud, Arthur Mensch, Elena Buchatskaya, Trevor Cai, Eliza Rutherford, Diego de Las Casas, Lisa Anne Hendricks, Johannes Welbl, Aidan Clark, et al. Training compute-optimal large language models. *arXiv preprint arXiv:2203.15556*, 2022.

[13] Naman Jain, King Han, Alex Gu, Wen-Ding Li, Fanjia Yan, Tianjun Zhang, Sida Wang, Armando Solar-Lezama, Koushik Sen, and Ion Stoica. Livecodebench: Holistic and contamination free evaluation of large language models for code. *arXiv preprint arXiv:2403.07974*, 2024.

[14] Jared Kaplan, Sam McCandlish, Tom Henighan, Tom B Brown, Benjamin Chess, Rewon Child, Scott Gray, Alec Radford, Jeffrey Wu, and Dario Amodei. Scaling laws for neural language models. *arXiv preprint arXiv:2001.08361*, 2020.

[15] Woosuk Kwon, Zhuohan Li, Siyuan Zhuang, Ying Sheng, Lianmin Zheng, Cody Hao Yu, Joseph Gonzalez, Hao Zhang, and Ion Stoica. Efficient memory management for large language model serving with pagedattention. In *Proceedings of the 29th Symposium on Operating Systems Principles*, pages 611–626, 2023.

[16] Hieu Tran Le, Yue Yuan, Miltiadis Allamanis, Peter C. Splash, and Oleksandr Polozov. Rltf: Reinforcement learning from unit test feedback. *arXiv preprint arXiv:2207.14577*, 2022.

[17] Jinhyuk Lee, Zhuyun Dai, Xiaoqi Ren, Blair Chen, Daniel Cer, Jeremy R Cole, Kai Hui, Michael Boratko, Rajvi Kapadia, Wen Ding, et al. Gecko: Versatile text embeddings distilled from large language models. *arXiv preprint arXiv:2403.20327*, 2024.

[18] Haoran Li, ZSEQH. Yuan, Ruomeng Guo, Yirui Zhang, Zhaoye Liu, Maosong Sun, and Jie Zhou. AutoIF: Aligning llms with divergent human preferences through iterative self-revised instruction following. *arXiv preprint arXiv:2406.13542*, 2024.

[19] Yujia Li, David Choi, Junyoung Chung, Nate Kushman, Julian Schrittwieser, Rémi Leblond, Tom Eccles, James Keeling, Felix Gimeno, Agustin Dal Lago, et al. Competition-level code generation with alphacode. *Science*, 378(6624):1092–1097, 2022.

[20] Nat McAleese, Rai Michael Pokorny, Juan Felipe Ceron Uribe, Evgenia Nitishinskaya, Maja Trebacz, and Jan Leike. Llm critics help catch llm bugs. *arXiv preprint arXiv:2407.00215*, 2024.

[21] Allen Ni, Loubna Ben Allal, Shuyang Hao, Paras Jain, Nan Mu, Martha Christopoulou, Cheng-Yu Peng, Charles Sutton, Erik Nijkamp, Oleksandr Polozov, et al. Teaching language models to rerank code with execution feedback. In *International Conference on Machine Learning*, pages 26262–26280. PMLR, 2023.

[22] Rafael Rafailov, Archit Sharma, Eric Mitchell, Christopher D Manning, Stefano Ermon, and Chelsea Finn. Direct preference optimization: Your language model is secretly a reward model. *Advances in Neural Information Processing Systems*, 36, 2024.

[23] Baptiste Roziere, 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.

[24] Rohan Taori, Ishaan Gulrajani, Tianyi Zhang, Yann Dubois, Xuechen Li, Carlos Guestrin, Percy Liang, and Tatsunori B. Hashimoto. Stanford alpaca: An instruction-following llama model. GitHub repository, 2023.

[25] Xuezhi Wang, Jason Wei, Dale Schuurmans, Quoc Le, Ed Chi, Sharan Narang, Aakanksha Chowdhery, and Denny Zhou. Self-consistency improves chain of thought reasoning in language models. *arXiv preprint arXiv:2203.11171*, 2022.

[26] Yizhong Wang, Yeganeh Kordi, Swaroop Mishra, Alisa Liu, Noah A Smith, Daniel Khashabi, and Hannaneh Hajishirzi. Self-instruct: Aligning language models with self-generated instructions. In *Proceedings of the 61st Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pages 13484–13508, 2023.

[27] Jason Wei, Xuezhi Wang, Dale Schuurmans, Maarten Bosma, Fei Xia, Ed Chi, Quoc V Le, Denny Zhou, et al. Chain-of-thought prompting elicits reasoning in large language models. *Advances in neural information processing systems*, 35:24824–24837, 2022.

[28] Yuxiang Wei, Zhe Wang, Jiawei Liu, Yifeng Ding, and Lingming Zhang. Magicoder: Empowering code generation with oss-instruct. In *Forty-first International Conference on Machine Learning*, 2024.

[29] Yu Yan, Fei Hu, Jiusheng Chen, Nikhil Bhendawade, Ting Ye, Yeyun Gong, Nan Duan, Desheng Cui, Bingyu Chi, and Ruofei Zhang. Fastseq: Make sequence generation faster. *arXiv preprint arXiv:2106.04718*, 2021.

[30] Kechi Zhang, Ge Li, Yihong Dong, Jingjing Xu, Jun Zhang, Jing Su, Yongfei Liu, and Zhi Jin. Codedpo: Aligning code models with self generated and verified source code. *arXiv preprint arXiv:2410.05605*, 2024.

[31] Chunting Zhou, Pengfei Liu, Puxin Xu, Srinivasan Iyer, Jiao Sun, Yuning Mao, Xuezhe Ma, Avia Efrat, Ping Yu, Lili Yu, et al. Lima: Less is more for alignment. *Advances in Neural Information Processing Systems*, 36, 2024.

A Prompt Template

Prompt for Generating Problem Description

[Code Snippet]
```
for (first_p, second_p) in zip_longest(diag1, diag2):
    assert first_p[0] == pytest.approx(second_p[0])
    assert first_p[1] == pytest.approx(second_p[1])
```

[Template]
Polycarp is reading a book consisting of n pages numbered from 1 to n. Every time he finishes the page with the number divisible by m, he writes down the last digit of this page number. For example, if $n = 15$ and $m = 5$, pages divisible by m are $5, 10, 15$. Their last digits are $5, 0, 5$ correspondingly, their sum is 10.
Your task is to calculate the sum of all digits Polycarp has written down.
You have to answer q independent queries.

[Instruction]
Please gain inspiration from the previous random code snippet and template to create a high-quality python programming problem.
Rules:
- Never mention the "code snippet".
- Don't write the solution.
- Do not specify constraints nor example inputs/outputs.
- The inspiration is just an inspiration. You can deviate from it.
- The solution of the problem should be only one function, not an entire program.
- The problem should be self-contained.

Prompt for Generating Function Signature

<Problem1>
Write a function to find the similar elements from the given two tuple lists.
</Problem1>
<Signature1>
similar_elements(test_tup1: list, test_tup2: list) ->list
</Signature1>
<Problem2>
Write a python function to identify non-prime numbers.
</Problem2>
<Signature2>

Prompt for Generating Test Input

<Q1>
Write a function to find the longest string in a list of strings. If the strings are not comparable (due to being of different lengths), the function should return None. function signature: longest_string(strings: list[str])
</Q1>
<ANALYSIS1>
- Case 1: 'strings' is a list of strings.
- Case 2: 'strings' is empty.
</ANALYSIS1>
<INPUTS1>
```
longest_string(['dog', 'cat', 'elephant']) # Consider case 1.
longest_string([]) # Consider case 2.
```
</INPUTS1>
<Q2>
Write a function to check if the given string represents a sequence of ASCII characters. The function should be able to handle different types of sequences, such as lists, tuples, and NumPy arrays. The function should return True if the sequence contains only ASCII characters, and False otherwise.
function signature: is_ascii(seq: list) ->bool
</Q2>
<ANALYSIS2>

Prompt for Generating Test Output

<Q1 >
Write a function to find the similar elements from the given two tuple lists.
function signature: `similar_elements(test_tup1: Tuple, test_tup2: Tuple)`
</Q1 >
<INPUT1 >
`similar_elements((3, 4, 5, 6),(5, 7, 4, 10))`
</INPUT1 >
<ANALYSIS1 >
- In the first tuple (3, 4, 5, 6), the elements 4 and 5 are present.
- In the second tuple (5, 7, 4, 10), the elements 4 and 5 are also present.
- Since 4 and 5 are present in both tuples, they should be included in the output.
- The other elements in the tuples (3, 6, 7, and 10) are not present in both tuples, so they should not be included in the output.
- So the expected output is (4, 5).
</ANALYSIS1 >
<OUTPUT1 >
`assert similar_elements((3, 4, 5, 6),(5, 7, 4, 10)) == (4, 5)`
</OUTPUT1 >
<Q2 >
Write a python function to identify non-prime numbers.
function signature: `is_not_prime(n: int)`
</Q2 >
<INPUT2 >
`is_not_prime(2)`
</INPUT2 >
<ANALYSIS2 >
- One of the fundamental properties of prime numbers is that they can only be divided evenly by 1 and themselves.
- 2 is considered a prime number because it can only be divided evenly by 1 and itself.
- So 2 is a prime, and the expected output is False.
</ANALYSIS2 >
<OUTPUT2 >
`assert is_not_prime(2) == False`
</OUTPUT2 >
<Q3 >
Write a function to find all words which are at least 4 characters long in a string by using regex.
function signature: `find_char_long(text: str)`
</Q3 >
<INPUT3 >
`find_char_long('Jing Eco and Tech')`
</INPUT3 >
<ANALYSIS3 >
- For the first word 'Jing', it's 4 characters long, and 4 >= 4, so it should be included.
- For the second word 'Eco', it's 3 characters long, and 3 < 4, so it should NOT be included.
- For the third word 'and', it's 3 characters long, and 3 < 4, so it should NOT be included.
- For the fourth word 'Tech', it's 4 characters long, and 4 >= 4, so it should be included.
- To sum up, the output is ['Jing', 'Tech'].
</ANALYSIS3 >
<OUTPUT3 >
`assert find_char_long('Jing Eco and Tech') == ['Jing', 'Tech']`
</OUTPUT3 >
<Q4 >
function signature:
</Q4 >
<INPUT4 >
</INPUT4 >
<ANALYSIS4 >

B Performance Change Across Iterations

Performance of Solver-Verifier Framework are shown in Figure 4

C Prompt Coverage Analysis

Prompt distribution comparison is shown in Figure 5

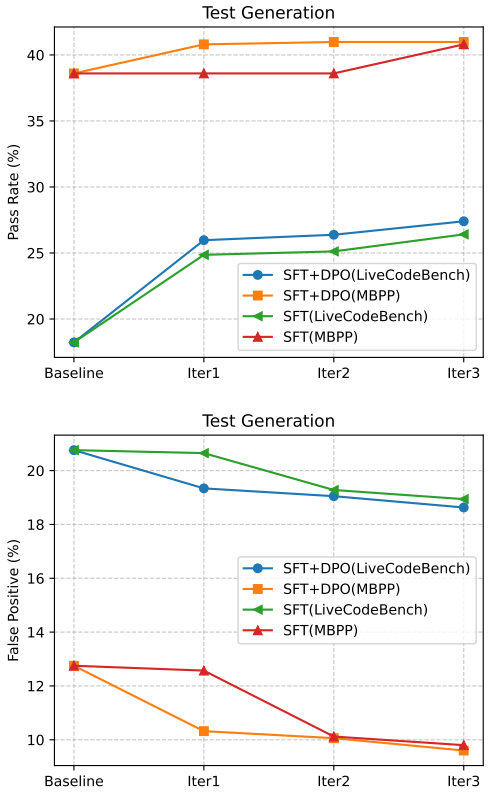

Figure 4: Performance of Solver-Verifier Framework.

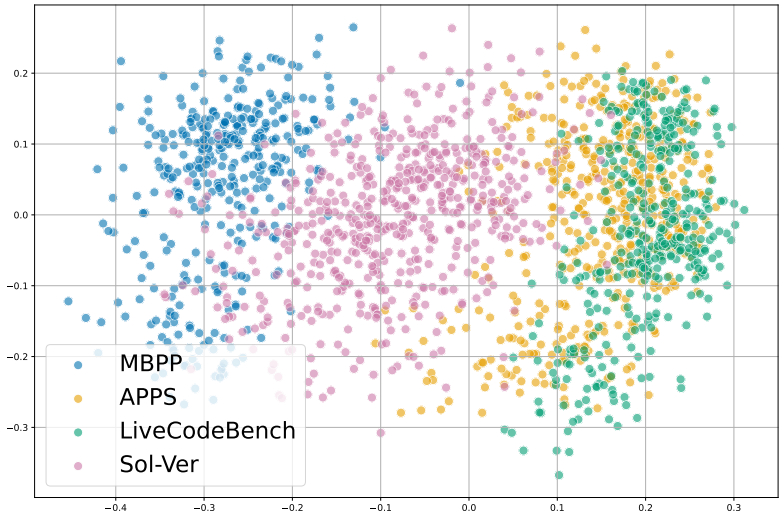

Figure 5: Prompt distribution comparison with other standard coding benchmarks. We use Principal Component Analysis (PCA) for embedding dimension reduction.