S^*: Test Time Scaling for Code Generation

Dacheng Li[*†] **Shiyi Cao**[*†] **Chengkun Cao**[†] **Xiuyu Li**[†]

Shangyin Tan **Kurt Keutzer** **Jiarong Xing** **Joseph Gonzalez**

Ion Stoica

University of California, Berkeley

Abstract

Increasing test-time compute for LLMs shows promise across domains but remains underexplored in code generation, despite extensive study in math. In this paper, we propose S^*, the first hybrid test-time scaling framework that substantially improves the coverage and selection accuracy of generated code. S^* augments the existing parallel scaling approach with sequential scaling to further increase the performance. It further leverages a novel selection mechanism that adaptively generates distinguishing inputs for pairwise comparison, combined with execution-grounded information to robustly identify correct solutions.

We evaluate S^* across 12 Large Language Models and Large Reasoning Models and show that: (1) S^* consistently improves performance across model families and sizes, enabling a 3B model to outperform GPT-4o-mini; (2) S^* enables non-reasoning models to surpass reasoning models—GPT-4o-mini with S^* outperforms o1-preview by 3.7% on LiveCodeBench; (3) S^* further boosts state-of-the-art reasoning models—DeepSeek-R1-Distill-Qwen-32B with S^* achieves 85.7% on LiveCodeBench, approaching o1 (high) at 88.5%. Codes, model generations and intermediate experiments results are available under https://github.com/NovaSky-AI/SkyThought.

1 Introduction

Increasing test-time compute has emerged as a powerful approach for improving the performance of large language models (LLMs) across diverse tasks (OpenAI, 2024; Guo et al., 2025; Qwen, 2024; Muennighoff et al., 2025; Team, 2025; Brown et al., 2024; Snell et al., 2024). In particular, test-time scaling has been extensively explored in mathematical reasoning, where parallel sampling increases solution coverage, sequential refinement improves individual samples through rethinking and revising, and reward models guide the search process more effectively (Ehrlich et al., 2025; Snell et al., 2024; Li et al., 2024b). These methods collectively push the performance boundaries of LLMs by leveraging additional compute during inference.

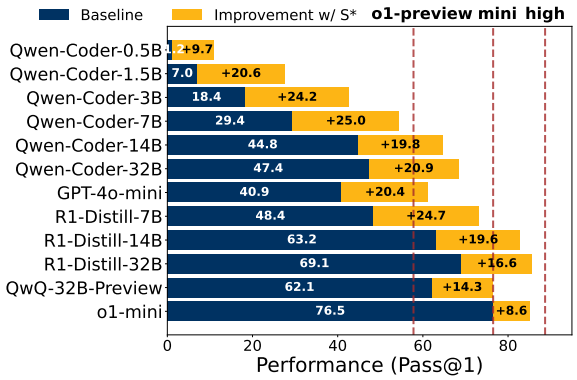

Figure 1: **Performance improvement with S^* in LiveCodeBench (v2)** (Jain et al., 2024). S^* consistently improves models across different sizes, allowing non-reasoning models to surpass reasoning models and open models to be competitive with o1 (high reasoning effort). "Qwen-Coder" denotes "Qwen2.5-Coder-Instruct," (Hui et al., 2024) and "R1-Distill" denotes "DeepSeek-R1-Distill-Qwen." (Guo et al., 2025).

Despite these advancements in the math domain, the potential of test-time scaling for code generation—a domain with both fundamental importance and widespread practical applications—remains under-explored. Code generation introduces unique challenges compared to math reasoning. Correctness in math can often be verified through rule-based string matching with reference answers (Guo et al., 2025; Zeng et al., 2025), whereas validating code requires executing a large set of test cases to accurately check functional correctness (Liu et al., 2023). This dependence on execution increases the complexity of test-time scaling and complicates the design of reward models (Zeng et al., 2025). However, code generation also offers a distinct advantage: The ability to execute

[*]Equal Contribution.
[†]Major Contributor.

Findings of the Association for Computational Linguistics: EMNLP 2025, pages 15964–15978
November 4-9, 2025

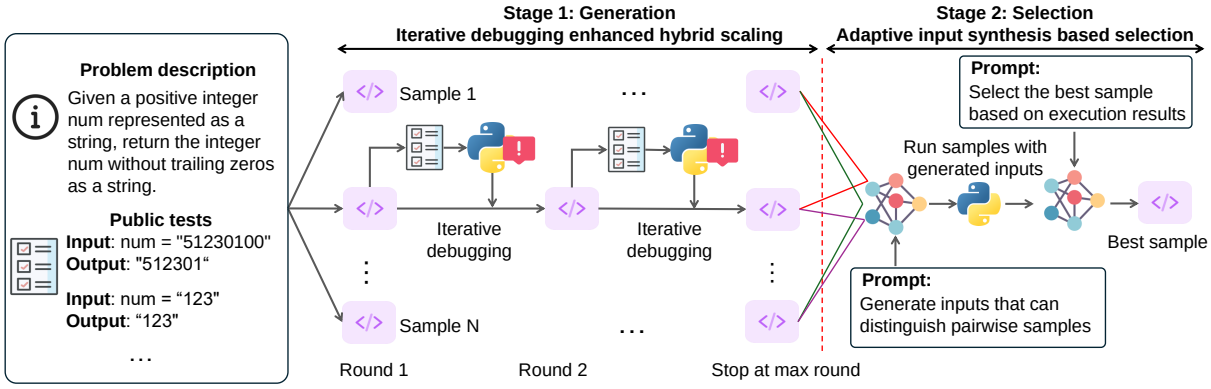

Figure 2: **Overview of S^*. Stage 1: Generation**—S^* enhances parallel samples through iterative debugging. Each sample is tested using public test cases executed via an interpreter, with outputs and/or error messages used to guide the next round of sample generation. **Stage 2: Selection**—S^* selects the best sample by prompting an LLM to generate inputs that differentiate between paired samples, then leveraging actual execution results to inform the LLM to determine the optimal choice.

programs allows us to obtain precise outputs and error messages, which provide a reliable grounding mechanism for improving generation and selection (Chen et al., 2023; Li et al., 2022).

In this paper, we propose S^*, the first hybrid test-time scaling framework for code generation, which substantially improves both coverage[1] and selection accuracy. The framework operates in two key stages (Fig. 2): (1) solution generation, which aimes to generate at least one correct solution, and (2) solution selection, which aimes to select a correct solution from generated ones.

First, in the generation stage, S^* augments parallel sampling (Brown et al., 2024; Li et al., 2022), which samples multiple solutions independently, with sequential scaling via iterative debugging. Each generated sample is executed on public test cases to obtain outputs and/or error messages, which are fed back into the model to iteratively refine the code. **Second**, in the selection stage, existing methods often rely on generating test inputs indiscriminately, which can fail to effectively differentiate between candidate solutions (Chen et al., 2022; Zeng et al., 2025). To overcome this limitation, S^* introduces *adaptive input synthesis*: for each pair of samples, an LLM is prompted to generate distinguishing test inputs. These inputs are executed, where the outputs are further provided to ground the LLM to select the best sample. This adaptive, execution-grounded approach ensures robust identification of correct solutions (§5.4).

S^* is a general approach that outperforms zero-shot generation and existing test-time scaling methods. We evaluate S^* on 12 models, spanning a wide range of sizes, both open and closed, instruction-based and reasoning models. S^* consistently enhances performance across these diverse settings. Notably, S^* (1) enables small models to surpass larger models within the same family: Qwen2.5-7B-Instruct + S^* outperforms Qwen2.5-32B-Instruct on LiveCodeBench by 10.7%; (2) enables instruction-based models to outperform reasoning models: GPT-4o-mini + S^* surpasses o1-preview by 3.7%; and (3) enables open reasoning models to achieve performance competitive with state-of-the-art closed models: DeepSeek-R1-Distill-Qwen-32B + S^* achieves 85.7% on LiveCodeBench, approaching the state-of-the-art performance of o1-high at 88.7%. Fig. 3 provides an overview of the performance improvements enabled by our techniques. In summary, our contributions are:

1. We present the first hybrid test-time scaling framework for code generation, combining the strengths of parallel and sequential scaling to improve the overall effectiveness of test-time scaling in code generation.

2. We introduce adaptive input synthesis, that employs LLMs to generate test cases capable of distinguishing between code samples to ground sample selections.

3. We conduct extensive evaluations on LiveCodeBench and CodeContests, demonstrating that S^* consistently improves performance across diverse model families and sizes.

4. We released all software artifacts, model gen-

[1]The fraction of problems that are solved by any generated sample (Brown et al., 2024).

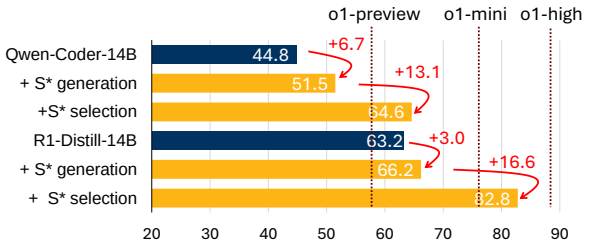

Figure 3: **Ablation of S^* performance benefits**: Qwen2.5-Coder-14B-Instruct (denoted as Qwen-Coder-14B) (Hui et al., 2024) with S^* can surpass o1-preview without S^*. DeepSeek-R1-Distill-Qwen-14B (denoted as R1-Distill-14B) (Guo et al., 2025) with S^* outperforms o1-mini without S^*. +S* generation denotes the setting where we perform parallel sampling and iterative debugging, where the additional +S* selection denotes to additionally apply the selection method (i.e., the full S* method). The accuracy is from LiveCodeBench v2.

erations, and intermediate results to support and accelerate future research in this area.

2 Related work

Test Time Scaling for LLMs. Existing approaches to increase test-time compute can be broadly categorized into two paradigms: parallel scaling and sequential scaling (Muennighoff et al., 2025). Parallel scaling (also known as repeated sampling) involves generating multiple solutions simultaneously and selecting the best one, a strategy commonly known as Best-of-N. Coverage—the fraction of problems solved by any of these N samples—continues to improve as N increases (Chollet, 2019; Irvine et al., 2023), even at the scale of 10^4 to 10^6 (Brown et al., 2024; Li et al., 2022). However, common selection strategies, such as (weighted) majority voting (Wang et al., 2022) and reward model scoring (Christiano et al., 2017; Lightman et al., 2023; Wang et al., 2024a; Wu et al., 2024; Beeching et al.; Pan et al., 2024), often struggle to select the best sample in parallel scaling (Brown et al., 2024; Hassid et al., 2024; Stroebl et al., 2024). We propose a novel method that improves selection for coding tasks.

Sequential scaling, on the other hand, encourages the model to refine its reasoning over multiple steps. This includes methods like chain-of-thought (CoT) prompting (Wei et al., 2022; Nye et al., 2021), and iterative rethinking and revision (Madaan et al., 2024; Lee et al., 2025; Hou et al., 2025; Huang et al., 2022; Min et al., 2024; Team, 2025; Muennighoff et al., 2025; Wang et al., 2024b; Li et al., 2025). Noticeably, OpenAI o1, DeepSeek R1, Qwen QwQ, and Kimi employ in-context long CoT with revision and backtracking to find the best solution (OpenAI, 2024; Guo et al., 2025; Qwen, 2024; Team et al., 2025). In this paper, we leverage iterative debugging from test execution feedback for sequential scaling code generation performance (Chen et al., 2023).

Test Time Scaling for Code Generation. Chen et al. (2022); Huang et al. (2023); Jiao et al. (2024) use models to generate code samples and test cases, selecting the final sample in a self-consistency manner (Wang et al., 2022; Zeng et al., 2025). However, these approaches often suffer from model hallucination, where the model fails to accurately predict the output of a test input (Jain et al., 2024; Zeng et al., 2025; Gu et al., 2024). AlphaCode explores large-scale parallel sampling with a trained model to generate test cases for filtering and selection (Li et al., 2022). AlphaCodium uses a series of self-revision on both public demonstration and model-generated tests to improve solutions (Ridnik et al., 2024). Saad-Falcon et al. (2024) searches over various inference techniques and finds that parallel sampling with model-generated tests works well for CodeContests problems (Li et al., 2022). Unlike methods relying solely on parallel sampling or sequential scaling, we use a hybrid approach that combines their advantages.

Hybrid Test-Time Scaling. Many works in the math domain study hybrid approaches that combine parallel and sequential scaling, often leveraging reward-model-guided tree search algorithms, such as Monte-Carlo Tree Search (MCTS), to effectively navigate the solution space (Gao et al., 2024; Li et al., 2024b; Silver et al., 2016; Snell et al., 2024; Hendrycks et al., 2021b). S1 (Muennighoff et al., 2025) primarily focuses on sequential scaling but observes diminishing returns and thus incorporates parallel-based approaches like majority voting and tree search to further enhance performance.

In contrast, our work applies hybrid scaling to code generation tasks without relying on tree search methods, as developing a general and effective reward model for the code generation domain remains challenging (Zeng et al., 2025). Instead, S^* augments parallel scaling with sequential scaling via execution-grounded iterative debugging to improve coverage and introduces adaptive input synthesis to enhance selection accuracy. (Ehrlich et al., 2025) generates parallel samples and revises samples, but it focuses on a different domain (soft-

ware engineering (Jimenez et al., 2023)).

3 Method

S^* takes as input a coding problem $\mathbf{P}$ and a code generation model $\mathbf{M}$. The model $\mathbf{M}$ aims to generate a program solution $\mathbf{X}(\cdot)$ that maps inputs to outputs according to the problem specification.

We adopt the standard setup widely used in existing coding benchmarks (Chen et al., 2021; Li et al., 2022, 2023; Jain et al., 2024; Hendrycks et al., 2021a; Gulwani et al.). Each coding problem $\mathbf{P}$ consists of a natural language description and a set of public and private test cases, each represented as input-output pairs.

Private tests evaluate the correctness of $\mathbf{X}$ but remain inaccessible to $\mathbf{M}$ during code generation. A solution is considered correct if it passes all private tests. In contrast, public tests are provided to clarify the problem's intent and are typically included in the prompt. Public tests are usually far fewer than private tests; for instance, in CodeContests (Li et al., 2022), there are, on average, 2.0 public tests and 202.1 private tests per problem. This contrasts with mathematical reasoning tasks, where evaluation typically relies on exact string matching of the final solution without additional test information (Li et al., 2024a).

3.1 The S^* Framework

S^* is a two-stage hybrid test-time scaling framework consisting of *Generation* and *Selection* stages, as demonstrated in Fig. 2. It extends parallel sampling with sequential sampling via iterative debugging to *improve coverage* and employs adaptive input synthesis during selection to *enhance selection accuracy*, leveraging execution results throughout the process. An example of effect for different stages can be found in Fig. 3.

Stage 1: Generation. In the generation stage, S^* improves coverage by extending parallel scaling with sequential scaling through *iterative debugging grounded with execution feedback*. Specifically, S^* first generates $\mathbf{N}$ initial samples independently, leveraging parallel sampling techniques (Chen et al., 2023). Each sample is then refined through up to $\mathbf{R}$ rounds of sequential revision, informed by execution results on public test cases. The revision process halts once a sample passes all public tests or reaches the maximum number of revision attempts.

Algorithm 1: Best Sample Selection in S^*

Input: Problem description: P
Input: Candidate samples: X
Output: The best selected sample: x^*

1 $\mathcal{T} \leftarrow$ `llm_test_input_gen`(P)
2 $\mathcal{O} \leftarrow$ `sample_execution`$(X, \mathcal{T})$
3 $\mathcal{C} \leftarrow$ `sample_clustering`$(\mathcal{O})$
4 Scores $\leftarrow \mathbf{0}$
5 **for** ***each*** *pair* $(C_i, C_j) \in \mathcal{C}$ **do**
6 Sample x_i, x_j from C_i, C_j
7 $\mathcal{T}_a \leftarrow$ `adaptive_input_gen`(x_i, x_j)
8 better_idx = `exec_and_llm_select`$(x_i, x_j, \mathcal{T}_a)$
9 Scores[better_idx] += 1
10 **end**
11 $C^* \leftarrow \arg\max$(Scores)
12 $x^* \leftarrow$ `random_pick`(C^*)
13 **return** x^*

Stage 2: Selection. After generating $\mathbf{N}$ candidate solutions, the next step is to identify the best one. Since the public tests are already used during the generation stage, additional evaluation is needed for reliable selection. We investigate two existing approaches: (1) LLM-as-a-judge (Zheng et al., 2023), which relies on pre-trained knowledge to compare candidate solutions, and (2) generated test cases (Li et al., 2022; Chen et al., 2022) which uses synthesized test cases to guide selection.

Unfortunately, we find that LLM-based judging alone often struggles to predict program behavior accurately, while generated tests frequently fail to provide reliable outputs for grounding the selection or to produce high-quality inputs that effectively distinguish samples (see Tab. 3).

To overcome these limitations, S^* introduces *adaptive input synthesis*, a hybrid selection approach that integrates LLM evaluation with execution-grounded verification, as illustrated in Algorithm 1. First, we prompt an LLM to synthesize a set of test inputs. We execute these inputs and cluster the $\mathbf{N}$ samples based on their execution outputs (Line 2 to Line 3) (Li et al., 2022). We then perform pairwise comparisons across clusters: for each comparison, we prompt the LLM to generate distinguishing inputs, execute both samples using these inputs, and select the superior one based on the execution results (Line 7 to Line 9). This adaptive selection process grounds LLM evaluations in concrete execution feedback, resulting in more reliable and accurate sample selection compared to naive LLM judging or generated tests-based methods (see §4).

Method	Qwen2.5-Coder-Instruct						4o-mini	R1-Distill			QwQ	o1-mini
	0.5B	1.5B	3B	7B	14B	32B		7B	14B	32B		
Zero-Shot	1.2	7.0	18.4	29.4	44.8	47.4	40.9	48.4	63.2	69.1	62.1	76.5
Majority Vote	2.5	11.0	25.2	40.5	50.8	55.9	46.6	58.7	68.1	75.8	67.3	81.6
Self-Debugging	2.4	9.4	27.8	39.9	51.5	59.5	51.7	58.4	66.2	70.1	59.3	79.9
S* (Select by GPT-4o-mini)	**10.9**	**27.6**	**42.7**	**54.4**	**64.6**	**70.1**	**61.3**	**73.2**	**82.8**	**85.7**	**76.3**	**85.3**
S* (Select by the same model)	10.5	24.8	39.7	52.8	64.0	69.7	-	-	-	-	-	-

Table 1: **Pass@1 of zero-shot, majority voting (Wang et al., 2022; Li et al., 2022), self-debugging (Chen et al., 2023), and S^* on LiveCodeBench (v2)**. Bold text denotes the best performance. "R1-Distill", "QwQ", "4o-mini" is short for "DeepSeek-R1-Distill-Qwen" (Guo et al., 2025), "QwQ-32B-Preview" (Qwen, 2024), and "GPT-4o-mini" (Achiam et al., 2023) respectively. We include two variants for a fair assessment of S*: (1) selection made by GPT-4o-mini, and (2) selection made by the model itself, specifically for models weaker than GPT-4o-mini, to avoid inflated performance due to a stronger selection model. *S^* consistently outperforms other baselines.*

4 Evaluation

In this section, we evaluate S^* across a diverse set of instruction-based and reasoning models, various model families, sizes, and access types (open and closed), as well as multiple benchmarks (Jain et al., 2024; Li et al., 2022). Our key findings demonstrate the generality and effectiveness of S^*:

1. S^* consistently improves model performance across different families, sizes, and types, and generalizes effectively to multiple code generation benchmarks, including LiveCodeBench (§4.2) and CodeContests (§4.4), showcasing its robustness and broad applicability.

2. S^* outperforms existing test-time scaling methods, including self-debugging (Chen et al., 2023) and majority voting (Wang et al., 2022; Li et al., 2022), by enhancing both coverage and selection accuracy (§4.3).

4.1 Experimental Setup

Models. We consider instruction-based and reasoning-based models. To compare performance across models sizes using S^*, we select a series of models within the same family. We experiment with 12 models: (1) Instruction-based models: Qwen2.5-Coder-Instruct series (0.5B, 1.5B, 3B, 7B, 14B, 32B), GPT-4o mini (0718 version) (Hui et al., 2024; Achiam et al., 2023); (2) Reasoning models: QwQ-32B-Preview, DeepSeek-R1-Distill-Qwen (7B, 14B, 32B), and o1-mini (Qwen, 2024; Guo et al., 2025; OpenAI, 2024).

Benchmarks. We primarily use LiveCodeBench (MIT License) as our main evaluation benchmark, given its extensive usage by recent reasoning models and its inclusion of difficulty levels, which help analyze the behavior of different techniques (Jain et al., 2024; DeepSeek, 2024; Qwen, 2024). We use its v4 version for development (e.g., selecting hyper-parameters), which contains problems from August 2024 to November 2024. For final evaluation, we use v2 version that is non-overlapping to v4, and contain more problems. LiveCodeBench (v2) contains 511 problems, ranging from easy (182 problems), medium (206 problems), to hard (123 problems). In addition, we evaluate S^* on CodeContests (Li et al., 2022), a collection of 165 challenging coding problems. We use Pass@1 as our primary metric (Chen et al., 2021). Some experiments report Pass@N with N samples (often referred to as the 'oracle' settings) (Stroebl et al., 2024; Brown et al., 2024).

Baselines. Our evaluation considers two categories of baselines. First, we assess our method's improvement over the original model (without test-time scaling), using three leading OpenAI reasoning models—o1-preview, o1-high, and o1-mini (OpenAI, 2024)—as performance benchmarks. Second, we evaluate different test-time scaling methods applied to the same models, encompassing both parallel (i.e., majority voting) and sequential (i.e., self-debugging) approaches.

Implementation Details. We configure S^* to generate 16 samples in parallel with a temperature of 0.7 (without top-p sampling) and perform 2 rounds of iterative debugging on public tests. We justify our choice of hyper-parameters in §5. Prompts are automatically generated by a prompting framework, DSPy, where detailed prompts can be found in Appendix A.2 (Khattab et al., 2023). We run codes in a sandbox to avoid maliciously generated code, according to (Chen

et al., 2021). The main results and the baseline of our largest model (DeepSeek-R1-Distill-Qwen32B) take 8xH100 compute for 1 day (numbers of the 11th column in Table 1), on the Livecodebench v2 dataset. Experiments are done once.

4.2 S^* Main Results

Fig. 1 presents a performance comparison on LiveCodeBench with and without S^*, alongside the o1-series reasoning models for reference. Our results demonstrate that S^* consistently enhances model performance. When applied to models within the same family, S^* allows small models to surpass large ones: Qwen2.5-7B-Coder-Instruct integrated with S^* outperforms Qwen2.5-32B-Coder-Instruct without S^* by 10.1%. Additionally, S^* enables instruction-based models to surpass reasoning models, as evidenced by GPT-4o mini (0718) with S^* outperforming o1-Preview. Moreover, S^* further improves strong reasoning models: DeepSeek-R1-Distill-Qwen-32B, when enhanced with S^*, surpasses o1-mini and achieves near state-of-the-art results comparable to o1 (high reasoning efforts).

To provide an alternative comparison, we also let each model perform its own selection. This avoids potential inflation from using the stronger GPT-4o-mini as the judge. Notably, the performance remains similar. These results highlight that S^* serves as a powerful test-time scaling technique to improve model performance across different scales, architectures, and reasoning capabilities.

4.3 Comparison to Other Test-Time Methods

We evaluate S^* against two popular test-time scaling methods: majority voting (Li et al., 2022) and self-debugging (Chen et al., 2023). Majority voting employs parallel scaling: the model generates N samples, clusters them based on execution results (Li et al., 2022), selects the largest cluster, and randomly picks a final sample from it. Self-debugging follows a sequential scaling approach: the model generates a single sample, iteratively refines it using public tests (Chen et al., 2023), and selects the final revised version.

We use consistent hyperparameters: 16 parallel samples for majority voting and 2 debugging rounds for self-debugging. GPT-4o mini generates inputs for majority voting clustering and refines code samples for reasoning models. We use the model itself to refine code for non-reasoning models. As shown in Tab. 1, S^* consistently outperforms both methods.For Qwen-2.5-Coder series, S^* improves 8.4% to 18.2% to baselines. DeepSeek-R1-Distill-Qwen-32B, S^* outperforms the majority vote baseline by 9.9%, and the self debugging baseline by 15.6%. These results demonstrating the effectiveness of our hybrid approach.

Model	Zero Shot	Majority Vote	Self Debugging	S*
Qwen-Coder-7B	1.8	3.6	6.7	**10.9**
Qwen-Coder-14B	9.7	10.3	12.1	**21.8**
Qwen-Coder-32B	10.1	10.9	10.9	**21.8**
GPT-4o-mini	9.1	12.1	13.3	**23.0**
o1-mini	32.7	42.4	32.7	**48.5**

Table 2: **Performance comparison on CodeContests**. "Qwen-Coder" and "R1-Distill" are short for "Qwen2.5-Coder-Instruct", and "DeepSeek-R1-Distill-Qwen".

From another perspective, the self-debugging baseline can be seen as the S* method without the selection module. By comparing the zero-shot, self-debugging, and S* (selection by GPT-4o-mini) rows, we can isolate the contributions of the self-debugging and selection components. Both significantly improve performance, with selection typically yielding greater gains. The remaining gap between S* and the oracle selection highlights the need for continued advancement in selection methods for test-time scaling in code generation.

4.4 Results on Other Benchmark

We further validate S^* on CodeContests (Li et al., 2022). Tab. 2 summarizes results, where S^* consistently improves both instruction-based and reasoning models significantly. In particular, Qwen-2.5-Coder-7B-Instruct with S^* improves 9.1% from its zero-shot peformance of 1.8%. It further outperforms GPT-4o mini without S^* by 1.8%. It also outperforms the majority vote and self-debugging baselines consistently.

5 Ablation Studies

We conduct further ablation studies to analyze the key components of S^* on LiveCodeBench v4:

Parallel Scaling We analyze the impact of different hyper-parameter choices, such as the temperature setting and the number of samples, on parallel sampling performance (§5.1), and incorporating in-context example retrieval into the parallel sampling process (§5.2). We find that moderate temperatures improve performance, and adding ICL example can potential further improve performance.

Sequential Scaling We explore variations of the iterative debugging process, e.g. with model-generated test cases (§5.3). We find that iteratively debugging from test execution feedback improve

performance, even for reasoning models. We find that simply appending execution results from public tests for every iteration works the best.

Selection Policy We assess the performance of different selection policies, comparing our adaptive input synthesis approach with alternative selection strategies (§5.4). We find that adaptive input synthesis selection is consistently more reliable than the generated tests and the LLM judge method.

5.1 Parallel Sampling Hyper-Parameters

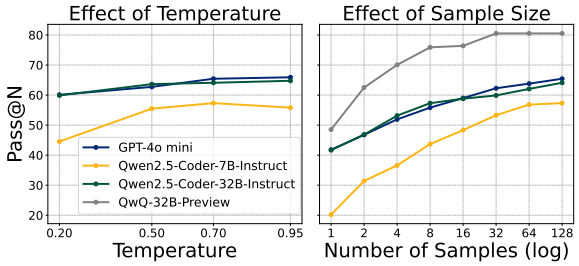

Figure 4: **The effect of hyper-parameters**. Left: The impact of temperature. Right: The effect of the number of samples. Performance improves log-linearly.

We examine the impact of two key factors in parallel sampling: temperature and the number of parallel samples. Understanding their influence is essential for optimizing test-time scaling strategies.

Moderate temperatures improve performance, but high temperatures degrade it. Fig. 4 (left) shows that moderate temperatures (0.2–0.7) enhance performance by balancing exploration and sample diversity. However, beyond 0.7, performance plateaus or declines, likely due to excessive randomness introducing noise. Qwen2.5-Coder-7B-Instruct exhibit performance regression at higher temperatures, emphasizing the trade-off between diversity and solution consistency.

Repeated sampling improves performance, even for reasoning models. As shown in Fig. 4 (right), increasing the number of parallel samples significantly improves performance across all models. Notably, Qwen2.5-Coder-7B-Instruct, the weakest performer at $N = 1$, shows the largest gain, exceeding 35% at $N = 64$. Similarly, the more capable reasoning-model (QwQ-32B-Preview) follows the same trend, though its gains plateau beyond $N = 32$. Nevertheless, it improves substantially, rising from 50% at $N = 1$ to 80% at $N = 32$. These results confirm that increasing the number of parallel samples is a simple yet effective strategy for enhancing performance in both instruction-following and reasoning-based models.

5.2 Impact of In-Context Examples

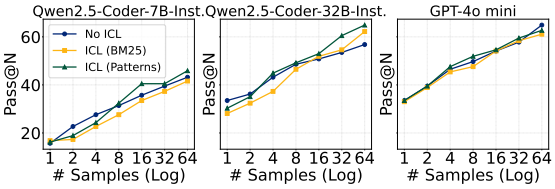

Figure 5: **Performance with in-context examples versus numbers of parallel samples** (N).

While S^* primarily focuses on repeated sampling for parallel scaling, it can be integrated with more advanced parallel scaling techniques. For instance, varying input prompts can create more diverse responses (Lambert et al., 2024), which lead to better coverage. Thus, we investigate whether augmenting prompts with in-context examples can further improve parallel scaling performance.

We construct an example set from LiveCodeBench (v2) containing correct solutions and reasoning traces generated by GPT-4o mini. We explore two retrieval approaches for selecting in-context examples. *ICL (BM25)* retrieves the top-k similar prompts using a BM25 retriever and prepends each to a different sample when $n = k$ (Robertson et al., 2009). This approach is simple but may overlook solution-level similarities. *ICL (Pattern)* groups problems by techniques (e.g., dynamic programming) and retrieves examples from the same technique, aiming to provide more relevant and structurally similar examples.

We evaluate medium-difficulty problems from LiveCodeBench (v4) with oracle selection. As shown in Fig. 5, performance is highly sensitive to in-context example quality. ICL (BM25) performs similarly to or worse than the zero-shot baseline in most cases, except for $n = 64$ with Qwen2.5-Coder-32B-Instruct. In contrast, ICL (Pattern) outperforms the baseline when $n \geq 8$ for Qwen2.5-Coder-7B-Instruct and $n \geq 4$ for Qwen2.5-Coder-32B-Instruct, while showing comparable performance with GPT-4o mini. These results highlight that selecting high-quality examples is crucial, and naive retrieval methods often fall short. Although ICL itself is promising, its performance is sensitive to example quality and retrieval effectiveness. We regard it as future work to develop robust ICL techniques that can be integrated into S^* to further enhance parallel scaling performance.

5.3 Impact of Iterative Debugging Variants

We examine the effectiveness of three variants of iterative debugging: (1) **Public Tests**: The model

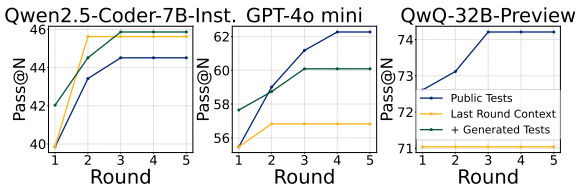

Figure 6: **Comparison of three iterative debugging approaches**: *Public Tests*, *+ Generated Tests* and *Last Round Context*. Results are obtained with $N = 8$, temperature $= 0.7$ and up to four rounds of debugging.

iteratively debugs using public tests and stops once the sample passes all of them. (2) **+Generated Tests**: In addition to public tests, the model continues debugging on model-generated tests following the algorithm in (Ridnik et al., 2024). (3) **Last Round Context**: The model iteratively debugs using only public tests, but instead of using code samples from all previous rounds for debugging, it uses only the last round of code sample as context. This is motivated by observations that LLMs may perform sub-optimally when handling large context windows (Liu et al., 2024).

Fig. 6 summarizes the result. We find that: (1) *Even though reasoning models implicitly perform self-reflection and revising, they benefit from explicit debugging through test execution feedback*: the performance of QwQ-32B-Preview model improves from 72.6 to 74.2 with 2 rounds of debugging. (2) *Reducing the context window or considering more model-generated tests does not show consistent improvement*: while using only the last round of context improves performance for the Qwen2.5-Coder-7B-Instruct model, it results in worse performance for the other two models. Similarly, incorporating additional model-generated tests does not enhance performance for GPT-4o mini. (3) *The benefits of iterative debugging tend to plateau, typically after 2–3 rounds*: this finding aligns with the observation that the benefit of sequential scaling flattens out (Muennighoff et al., 2025). Motivated by these findings, we choose to use 2 round of debugging, only on public tests for simplicity, and apply iterative debugging for reasoning models in §4.2.

5.4 Impact of Different Selection Policies

We compare different policies for selecting the best sample after iterative debugging. We evaluate four approaches: (1) **Public Only**: using only public test cases for selection and randomly selecting a sample if it passes all tests; (2) **Generated Tests**: applying public test filtering followed by additional test case generation using GPT-4o mini, selecting the sample that passes the most test cases; (3) **LLM Judge**: applying public test filtering and then using LLMs for pairwise selection among code samples; and (4) **Adaptive Input Synthesis** —applying the selection algorithm described in § 3.1 with GPT-4o mini after public test filtering.

Model	Public Only	Generated Tests	LLM Judge	Adaptive Input Synthesis (Ours)
Qwen-Coder-7B	42.3	42.3	42.3	**44.5**
Qwen-Coder-32B	54.6	57.8	55.5	**58.3**
GPT-4o mini	54.1	55.2	56.3	**57.3**
QwQ-32B-Preview	64.3	65.9	68.6	**69.7**
Avg.	53.8	53.1	55.6	**57.5**

Table 3: **Pass@1 Performance comparison between different selection methods.** Bold text denotes the best performance of the same model. "Qwen-Coder", "R1-Distill" short for "Qwen2.5-Coder-Instruct" and "DeepSeek-R1-Distill-Qwen".

Tab. 3 summarizes the results. Notably, the Generated Tests approach does not yield improvements over the Public Only baseline. This is due to errors in model-generated outputs, which, when applied to poorly chosen inputs, introduce significant noise in the selection process, ultimately degrading performance. In contrast, our Adaptive Selection method enables the LLM to strategically select an input that best differentiates samples while avoiding the need to predict outputs. By leveraging real execution outputs rather than model predicttions, the LLM makes more reliable decisions, leading to improved selection accuracy.

6 Conclusion

We propose S^*, the first hybrid test-time scaling framework for code generation that substantially improves both coverage and selection accuracy. S^* extends the existing parallel scaling paradigm with sequential scaling and incorporates *adaptive input synthesis*, a novel mechanism that synthesizes distinguishing test inputs to differentiate candidates and identify correct solutions via execution results. S^* consistently improves code generation performance across benchmarks. Notably, S^* enables a 3B model to outperform GPT-4o mini, GPT-4o mini to surpass o1-preview by 3.7% on LiveCodeBench, and DeepSeek-R1-Distill-Qwen-32B to achieve 86.7% on LiveCodeBench, approaching o1-high at 88.5%.

7 Limitations

This work primarily focuses on competition-level code generation, where it does not studies tasks such as software engineering tasks, e.g., SWE-BENCH (Jimenez et al., 2023). The method primarily focuses on improving accuracy, while it does not aim for minimizing costs.

References

Josh Achiam, Steven Adler, Sandhini Agarwal, Lama Ahmad, Ilge Akkaya, Florencia Leoni Aleman, Diogo Almeida, Janko Altenschmidt, Sam Altman, Shyamal Anadkat, et al. 2023. Gpt-4 technical report. *arXiv preprint arXiv:2303.08774*.

Edward Beeching, Lewis Tunstall, and Sasha Rush. Scaling test-time compute with open models.

Bradley Brown, Jordan Juravsky, Ryan Ehrlich, Ronald Clark, Quoc V Le, Christopher Ré, and Azalia Mirhoseini. 2024. Large language monkeys: Scaling inference compute with repeated sampling. *arXiv preprint arXiv:2407.21787*.

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

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. 2021. Evaluating large language models trained on code. *arXiv preprint arXiv:2107.03374*.

Xinyun Chen, Maxwell Lin, Nathanael Schärli, and Denny Zhou. 2023. Teaching large language models to self-debug. *arXiv preprint arXiv:2304.05128*.

François Chollet. 2019. On the measure of intelligence. *arXiv preprint arXiv:1911.01547*.

Paul F Christiano, Jan Leike, Tom Brown, Miljan Martic, Shane Legg, and Dario Amodei. 2017. Deep reinforcement learning from human preferences. *Advances in neural information processing systems*, 30.

DeepSeek. 2024. Deepseek-r1-lite-preview release. https://api-docs.deepseek.com/news/news1120. Accessed: 2024-11-20.

Ryan Ehrlich, Bradley Brown, Jordan Juravsky, Ronald Clark, Christopher Ré, and Azalia Mirhoseini. 2025. Codemonkeys: Scaling test-time compute for software engineering. *arXiv preprint arXiv:2501.14723*.

Zitian Gao, Boye Niu, Xuzheng He, Haotian Xu, Hongzhang Liu, Aiwei Liu, Xuming Hu, and Lijie Wen. 2024. Interpretable contrastive monte carlo tree search reasoning. *arXiv preprint arXiv:2410.01707*.

Alex Gu, Baptiste Rozière, Hugh Leather, Armando Solar-Lezama, Gabriel Synnaeve, and Sida I Wang. 2024. Cruxeval: A benchmark for code reasoning, understanding and execution. *arXiv preprint arXiv:2401.03065*.

Sumit Gulwani, Oleksandr Polozov, and Rishabh Singh. Foundations and trends in programming languages. *Bd*, 4:1–119.

Daya Guo, Dejian Yang, Haowei Zhang, Junxiao Song, Ruoyu Zhang, Runxin Xu, Qihao Zhu, Shirong Ma, Peiyi Wang, Xiao Bi, et al. 2025. Deepseek-r1: Incentivizing reasoning capability in llms via reinforcement learning. *arXiv preprint arXiv:2501.12948*.

Michael Hassid, Tal Remez, Jonas Gehring, Roy Schwartz, and Yossi Adi. 2024. The larger the better? improved llm code-generation via budget reallocation. *arXiv preprint arXiv:2404.00725*.

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

Dan Hendrycks, Steven Basart, Saurav Kadavath, Mantas Mazeika, Akul Arora, Ethan Guo, Collin Burns, Samir Puranik, Horace He, Dawn Song, et al. 2021b. Measuring coding challenge competence with apps. *arXiv preprint arXiv:2105.09938*.

Zhenyu Hou, Xin Lv, Rui Lu, Jiajie Zhang, Yujiang Li, Zijun Yao, Juanzi Li, Jie Tang, and Yuxiao Dong. 2025. Advancing language model reasoning through reinforcement learning and inference scaling. *arXiv preprint arXiv:2501.11651*.

Baizhou Huang, Shuai Lu, Weizhu Chen, Xiaojun Wan, and Nan Duan. 2023. Enhancing large language models in coding through multi-perspective self-consistency. *arXiv preprint arXiv:2309.17272*.

Jiaxin Huang, Shixiang Shane Gu, Le Hou, Yuexin Wu, Xuezhi Wang, Hongkun Yu, and Jiawei Han. 2022. Large language models can self-improve. *arXiv preprint arXiv:2210.11610*.

Binyuan Hui, Jian Yang, Zeyu Cui, Jiaxi Yang, Dayiheng Liu, Lei Zhang, Tianyu Liu, Jiajun Zhang, Bowen Yu, Kai Dang, et al. 2024. Qwen2. 5-coder technical report. *arXiv preprint arXiv:2409.12186*.

Robert Irvine, Douglas Boubert, Vyas Raina, Adian Liusie, Ziyi Zhu, Vineet Mudupalli, Aliaksei Korshuk, Zongyi Liu, Fritz Cremer, Valentin Assassi, et al. 2023. Rewarding chatbots for real-world engagement with millions of users. *arXiv preprint arXiv:2303.06135*.

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

Fangkai Jiao, Geyang Guo, Xingxing Zhang, Nancy F Chen, Shafiq Joty, and Furu Wei. 2024. Preference optimization for reasoning with pseudo feedback. *arXiv preprint arXiv:2411.16345*.

Carlos E Jimenez, John Yang, Alexander Wettig, Shunyu Yao, Kexin Pei, Ofir Press, and Karthik Narasimhan. 2023. Swe-bench: Can language models resolve real-world github issues? *arXiv preprint arXiv:2310.06770*.

Omar Khattab, Arnav Singhvi, Paridhi Maheshwari, Zhiyuan Zhang, Keshav Santhanam, Sri Vardhamanan, Saiful Haq, Ashutosh Sharma, Thomas T Joshi, Hanna Moazam, et al. 2023. Dspy: Compiling declarative language model calls into self-improving pipelines. *arXiv preprint arXiv:2310.03714*.

Nathan Lambert, Jacob Morrison, Valentina Pyatkin, Shengyi Huang, Hamish Ivison, Faeze Brahman, Lester James V Miranda, Alisa Liu, Nouha Dziri, Shane Lyu, et al. 2024. T\" ulu 3: Pushing frontiers in open language model post-training. *arXiv preprint arXiv:2411.15124*.

Kuang-Huei Lee, Ian Fischer, Yueh-Hua Wu, Dave Marwood, Shumeet Baluja, Dale Schuurmans, and Xinyun Chen. 2025. Evolving deeper llm thinking. *arXiv preprint arXiv:2501.09891*.

Dacheng Li, Shiyi Cao, Tyler Griggs, Shu Liu, Xiangxi Mo, Shishir G Patil, Matei Zaharia, Joseph E Gonzalez, and Ion Stoica. 2025. Llms can easily learn to reason from demonstrations structure, not content, is what matters! *arXiv preprint arXiv:2502.07374*.

Jia Li, Edward Beeching, Lewis Tunstall, Ben Lipkin, Roman Soletskyi, Shengyi Huang, Kashif Rasul, Longhui Yu, Albert Q Jiang, Ziju Shen, et al. 2024a. Numinamath: The largest public dataset in ai4maths with 860k pairs of competition math problems and solutions. *Hugging Face repository*, 13:9.

Qingyao Li, Wei Xia, Kounianhua Du, Xinyi Dai, Ruiming Tang, Yasheng Wang, Yong Yu, and Weinan Zhang. 2024b. Rethinkmcts: Refining erroneous thoughts in monte carlo tree search for code generation. *arXiv preprint arXiv:2409.09584*.

Rongao Li, Jie Fu, Bo-Wen Zhang, Tao Huang, Zhihong Sun, Chen Lyu, Guang Liu, Zhi Jin, and Ge Li. 2023. Taco: Topics in algorithmic code generation dataset. *arXiv preprint arXiv:2312.14852*.

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

Hunter Lightman, Vineet Kosaraju, Yura Burda, Harri Edwards, Bowen Baker, Teddy Lee, Jan Leike, John Schulman, Ilya Sutskever, and Karl Cobbe. 2023. Let's verify step by step. *arXiv preprint arXiv:2305.20050*.

Jiawei Liu, Chunqiu Steven Xia, Yuyao Wang, and Lingming Zhang. 2023. Is your code generated by chatgpt really correct? rigorous evaluation of large language models for code generation. *Advances in Neural Information Processing Systems*, 36:21558–21572.

Nelson F Liu, Kevin Lin, John Hewitt, Ashwin Paranjape, Michele Bevilacqua, Fabio Petroni, and Percy Liang. 2024. Lost in the middle: How language models use long contexts. *Transactions of the Association for Computational Linguistics*, 12:157–173.

Aman Madaan, Niket Tandon, Prakhar Gupta, Skyler Hallinan, Luyu Gao, Sarah Wiegreffe, Uri Alon, Nouha Dziri, Shrimai Prabhumoye, Yiming Yang, et al. 2024. Self-refine: Iterative refinement with self-feedback. *Advances in Neural Information Processing Systems*, 36.

Yingqian Min, Zhipeng Chen, Jinhao Jiang, Jie Chen, Jia Deng, Yiwen Hu, Yiru Tang, Jiapeng Wang, Xiaoxue Cheng, Huatong Song, et al. 2024. Imitate, explore, and self-improve: A reproduction report on slow-thinking reasoning systems. *arXiv preprint arXiv:2412.09413*.

Niklas Muennighoff, Zitong Yang, Weijia Shi, Xiang Lisa Li, Li Fei-Fei, Hannaneh Hajishirzi, Luke Zettlemoyer, Percy Liang, Emmanuel Candès, and Tatsunori Hashimoto. 2025. s1: Simple test-time scaling. *arXiv preprint arXiv:2501.19393*.

Maxwell Nye, Anders Johan Andreassen, Guy Gur-Ari, Henryk Michalewski, Jacob Austin, David Bieber, David Dohan, Aitor Lewkowycz, Maarten Bosma, David Luan, et al. 2021. Show your work: Scratchpads for intermediate computation with language models. *arXiv preprint arXiv:2112.00114*.

OpenAI. 2024. Learning to reason with llms. https://openai.com/index/learning-to-reason-with-llms/. Accessed: 2024-11-20.

Jiayi Pan, Xingyao Wang, Graham Neubig, Navdeep Jaitly, Heng Ji, Alane Suhr, and Yizhe Zhang. 2024. Training software engineering agents and verifiers with swe-gym. *arXiv preprint arXiv: 2412.21139*.

Qwen. 2024. Qwq: Reflect deeply on the boundaries of the unknown. https://qwenlm.github.io/blog/qwq-32b-preview/.

Tal Ridnik, Dedy Kredo, and Itamar Friedman. 2024. Code generation with alphacodium: From prompt engineering to flow engineering. *arXiv preprint arXiv:2401.08500*.

Stephen Robertson, Hugo Zaragoza, et al. 2009. The probabilistic relevance framework: Bm25 and beyond. *Foundations and Trends® in Information Retrieval*, 3(4):333–389.

Jon Saad-Falcon, Adrian Gamarra Lafuente, Shlok Natarajan, Nahum Maru, Hristo Todorov, Etash Guha, E Kelly Buchanan, Mayee Chen, Neel Guha, Christopher Ré, et al. 2024. Archon: An architecture search framework for inference-time techniques. *arXiv preprint arXiv:2409.15254*.

David Silver, Aja Huang, Chris J Maddison, Arthur Guez, Laurent Sifre, George Van Den Driessche, Julian Schrittwieser, Ioannis Antonoglou, Veda Panneershelvam, Marc Lanctot, et al. 2016. Mastering the game of go with deep neural networks and tree search. *nature*, 529(7587):484–489.

Charlie Snell, Jaehoon Lee, Kelvin Xu, and Aviral Kumar. 2024. Scaling llm test-time compute optimally can be more effective than scaling model parameters. *arXiv preprint arXiv:2408.03314*.

Benedikt Stroebl, Sayash Kapoor, and Arvind Narayanan. 2024. Inference scaling flaws: The limits of llm resampling with imperfect verifiers. *arXiv preprint arXiv:2411.17501*.

Kimi Team, Angang Du, Bofei Gao, Bowei Xing, Changjiu Jiang, Cheng Chen, Cheng Li, Chenjun Xiao, Chenzhuang Du, Chonghua Liao, et al. 2025. Kimi k1. 5: Scaling reinforcement learning with llms. *arXiv preprint arXiv:2501.12599*.

NovaSky Team. 2025. Sky-t1: Train your own o1 preview model within 450 dollars. https://novasky-ai.github.io/posts/sky-t1. Accessed: 2025-01-09.

Peiyi Wang, Lei Li, Zhihong Shao, Runxin Xu, Damai Dai, Yifei Li, Deli Chen, Yu Wu, and Zhifang Sui. 2024a. Math-shepherd: Verify and reinforce llms step-by-step without human annotations. In *Proceedings of the 62nd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pages 9426–9439.

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

Yifei Wang, Yuyang Wu, Zeming Wei, Stefanie Jegelka, and Yisen Wang. 2024b. A theoretical understanding of self-correction through in-context alignment. *arXiv preprint arXiv:2405.18634*.

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

Yangzhen Wu, Zhiqing Sun, Shanda Li, Sean Welleck, and Yiming Yang. 2024. Inference scaling laws: An empirical analysis of compute-optimal inference for problem-solving with language models. *arXiv preprint arXiv:2408.00724*.

Huaye Zeng, Dongfu Jiang, Haozhe Wang, Ping Nie, Xiaotong Chen, and Wenhu Chen. 2025. Acecoder: Acing coder rl via automated test-case synthesis. *ArXiv*, 2502.01718.

Lianmin Zheng, Wei-Lin Chiang, Ying Sheng, Siyuan Zhuang, Zhanghao Wu, Yonghao Zhuang, Zi Lin, Zhuohan Li, Dacheng Li, Eric Xing, et al. 2023.

Judging llm-as-a-judge with mt-bench and chatbot arena. *Advances in Neural Information Processing Systems*, 36:46595–46623.

A Appendix

A.1 Example of Coding Problem

In the method section (§3), we introduce our problem setup, which includes unambiguous configuration with a small amount of demonstrations. In this section, we provide one such example to better illustrate how typically dataset provide questions. In particular, we show one sample from the hard subset of LiveCodeBench (Jain et al., 2024).

Question

You are given a string word and an array of strings forbidden. A string is called valid if none of its substrings are present in forbidden. Return the length of the longest valid substring of the string word. A substring is a contiguous sequence of characters in a string, possibly empty.

Example 1:
Input: word = "cbaaaabc", forbidden = ["aaa","cb"]
Output: 4
Explanation: There are 11 valid substrings in word: "c", "b", "a", "ba", "aa", "bc", "baa", "aab", "ab", "abc" and "aabc". The length of the longest valid substring is 4. It can be shown that all other substrings contain either "aaa" or "cb" as a substring.

Example 2:
Input: word = "leetcode", forbidden = ["de","le","e"]
Output: 4
Explanation: There are 11 valid substrings in word: "l", "t", "c", "o", "d", "tc", "co", "od", "tco", "cod", and "tcod". The length of the longest valid substring is 4. It can be shown that all other substrings contain either "de", "le", or "e" as a substring.

Constraints:
$1 \leq$ word. length $\leq 10^5$ word consists only of lowercase English letters. $1 \leq$ forbidden. length $\leq 10^5$. $1 \leq$ forbidden$[i]$. length $\leq$ 10. forbidden[i] consists only of lowercase English letters.

A.2 Prompt templates

We also provide detailed prompts used in our experiments in Fig. 7 to Fig. 9. These prompts are generated automatically by DSPy (Khattab et al., 2023).

System message:
Your input fields are:
1. `prompt` (str)

Your output fields are:
1. `reasoning` (str)
2. `code` (str): Here is the past history of your code and the test case feedback. Please reason why your code failed in the last round, and correct the code. Do not write non-code content in the code field.

All interactions will be structured in the following way, with the appropriate values filled in.
[[## prompt ##]]
{prompt}

[[## reasoning ##]]
{reasoning}

[[## code ##]]
{code}

[[## completed ##]]

In adhering to this structure, your objective is: Given the fields `prompt`, produce the fields `code`.

User message:
[[## prompt ##]]
{Question Prompt}
Code:
[Round 0 Reasoning]: {Round 0 Reasoning}
[Round 0 Generated code]: {Round 0 Generated Code}
[Round 0 Test Feedback]: {Round 0 Test Feedback}

Respond with the corresponding output fields, starting with the field `[[ ## reasoning ## ]]`, then `[[ ## code ## ]]`, and then ending with the marker for `[[ ## completed ## ]]`.

Figure 7: The prompt for iterative debugging.

System message:
Your input fields are:
1. `prompt` (str)
Your output fields are:
1. `reasoning` (str)
2. `tests` (str): Generate a complete set of potential inputs to test an AI-generated solution to the coding problem. Cover: (i) Edge cases, such as empty string or arrays, (ii) Complex and difficult inputs, but do not include very long inputs. (iii) Other ones that can maximize the chance of catching a bug. Provide the input and output in JSON format as follows: {"input": <example_input>, "output": <expected_output>} Ensure the input and output match the types and structure expected for the problem. Do not include any additional text or explanations, just the JSON object.

All interactions will be structured in the following way, with the appropriate values filled in.
[[## prompt ##]] {prompt}
[[## reasoning ##]] {reasoning}
[[## tests ##]] {tests}
[[## completed ##]]

In adhering to this structure, your objective is: Given the fields `prompt`, produce the fields `tests`.

User message:
[[## prompt ##]] {Question Prompt}

Respond with the corresponding output fields, starting with the field `[[ ## reasoning ## ]]`, then `[[ ## tests ## ]]`, and then ending with the marker for `[[ ## completed ## ]]`.

Figure 8: The prompt for generating test cases.

System message:

Your input fields are:
1. `prompt` (str)

Your output fields are:
1. `reasoning` (str)
2. `code` (str): Executable Python function generated from the given prompt.
   DO NOT include anything other than function body! Give me only the function itself!

All interactions will be structured in the following way, with the appropriate values filled in.

[[## prompt ##]]
{prompt}

[[## reasoning ##]]
{reasoning}

[[## code ##]]
{code}

[[## completed ##]]

In adhering to this structure, your objective is:
Given the fields `prompt`, produce the fields `code`.

User message:

[[## prompt ##]]
{Question Prompt}

Code:

Respond with the corresponding output fields, starting with the field `[[ ## reasoning ## ]]`, then `[[ ## code ## ]]`, and then ending with the marker for `[[ ## completed ## ]]`.

Figure 9: The prompt for code generation.