CodeGrad: Integrating Multi-Step Verification with Gradient-Based LLM Refinement

Anonymous ACL submission

Abstract

LLMs have shown strong code generation ability, but their outputs often lack guarantees of correctness, robustness, and efficiency in constraint-heavy tasks. CODEGRAD is a principled, iterative generation loop that turns structured checks and task constraints into a textual pseudo-gradient to steer an LLM toward better code. Treating code as a differentiable-like text variable, CODEGRAD converts reviewer feedback and constraint signals into ordered edit directions. This guidance yields implementations that are not only functional but also robust and consistent with the specification and efficiency requirements. We evaluate CodeGrad on the HumanEval, HumanEval+, and LiveCodeBench benchmarks. Our implementation outperforms baselines, achieving an absolute improvement of up to 27% on HumanEval and a 41% relative improvement on the challenging LiveCodeBench V6.

1 Introduction

Automated code generation is a transformative technology in software engineering and AI, accelerating development, reducing human error, and making programming more accessible (Svyatkovskiy et al., 2020; Becker et al., 2023; Li et al., 2022). Large Language Models (LLMs) have demonstrated capabilities in generating code from natural language descriptions, solving competitive programming problems (Gu, 2023; Wang and Chen, 2023; Liu et al., 2023).

Early explorations in LLM-based code synthesis centered on one-shot or few-shot generation (Xiong et al., 2024; Ye et al., 2025; Ságodi et al., 2024), the recent research has shifted to augmentation with feedback mechanisms like test-driven development and RLHF (Shen and Zhang, 2024; Gehring et al., 2024). Despite these methods can produce syntactically correct and functional code, the resulting code, while often functional, frequently struggles with edge cases, fails to generalize beyond seen examples, and lacks mechanisms for systematic improvement (Wang and Chen, 2023; Liu et al., 2023; Li et al., 2022; Liu et al., 2024; Zhang et al., 2025; Ouyang et al., 2023).

Rule- and spec-driven checking offers rigor but is hard to scale to open-ended, natural-language tasks where LLMs excel; closing this gap requires loops that translate checks into actionable guidance for models (Liu and Adams, 1995; Woodcock et al., 2009). The challenge is to combine the flexibility of LLMs with constraint-aware guidance so that models improve systematically under explicit requirements.(Agarwal et al., 2024; Fakhoury et al., 2024). Encouragingly, recent studies have demonstrated the potential of using natural language as a proxy for formal reasoning to guide and constrain LLM behavior, suggesting a viable path toward this integration (Cao et al., 2025; Jha et al., 2023; Morishita et al., 2024).

We introduce CODEGRAD, which treats code revision as an optimization guided by textual pseudo-gradients derived from structured evaluation. As illustrated in Figure 1, our approach treats code as a differentiable variable. This allows us to iteratively refine code by generating a textual pseudo-gradient—a structured set of corrections—derived from rigorous formal analysis. Rather than a numeric vector, this gradient consists of specific, textual edits that steer the code toward an optimal state. At each iteration, a backward reviewer analyzes the draft against correctness, I/O format, efficiency, and coverage, returning a structured critique that is parsed into edit directives. The resulting critique is converted into the textual pseudo-gradient, which guides a forward engine to produce an improved candidate. An update module then applies these directives while enforcing explicit invariants such as input contracts and complexity targets. Each revision includes a concise, natural-language rationale that explains why the

update satisfies the stated requirements.

2 Related Work

Self-critique and iterative refinement. Research has evolved from improving step-by-step reasoning to multi-round refinement with explicit feedback (Hao et al., 2024; Zhang et al., 2024). *Reflexion* established agent self-critique to steer subsequent attempts (Shinn et al., 2023), and recent systems generalize this loop for long-horizon tasks and program synthesis (Zhou et al., 2024; Abbasi Yadkori et al., 2024). TextGrad conceptualizes natural-language critiques as pseudo-gradients that prioritize edits in discrete code space (Yuksekgonul et al., 2024).

Test- and execution-driven feedback. In code generation, unit tests and sandboxed execution provide actionable signals for repair. Methods such as CODET and CODERL leverage test outcomes, I/O conformance, and traces to guide revisions, while noting brittleness when tests are weak or incomplete (Le et al., 2022; Chen et al., 2022).

3 Methodology

3.1 Problem Definition

We seek to automatically generate a program that satisfies a natural-language task description while meeting a set of explicit constraints. Formally, given a natural language task description T, our objective is to discover a program P from the space of all possible Python programs $\mathcal{S}$ that satisfies two primary conditions.

First, the program must be *functionally correct*. This means that for every valid input x from the domain of possible inputs $\mathcal{D}$, the output of the generated program $P(x)$ must match the output of the ideal target function $f_T(x)$ induced by the description T. This can be expressed as:

$$\forall x \in \mathcal{D} : P(x) = f_T(x), \tag{1}$$

where f_T is the unknown ground-truth function.

Second, the program must satisfy a set of *formal invariants*, which represent non-negotiable properties required for the solution to be considered valid.Constraints $I = \{I_1, \ldots, I_k\}$ cover I/O format, edge-case coverage, time/space targets, and syntactic validity.

3.2 Verification-Gradient Code Generation Pipeline

Conventional optimization uses a numerical gradient $\partial\mathcal{L}/\partial\theta$ to steer parameters θ toward lower loss $\mathcal{L}$. In code generation, the surface is discrete, so we approximate a gradient over discrete code by mapping reviewer feedback into an ordered list of edits (a textual pseudo-gradient). (Yuksekgonul et al., 2024):

$$g^{(t)} = \mathrm{Parse}\big(F^{(t)}\big) \longrightarrow \{(\text{location, action})\}_j. \tag{2}$$

Each tuple pinpoints a fragment of $P^{(t)}$ together with the type of modification (e.g., "replace $\mathcal{O}(n^3)$ loop with linear scan"). Although expressed in natural language, $g^{(t)}$ functions exactly like a real gradient: following its direction increases the likelihood that the next candidate will satisfy the loss template defined below. As figure 1, at each iteration $t \in \{0, \ldots, N\}$ the system maintains a differentiable text variable $P^{(t)}$ and executes the following four phases:

1. **Forward Generation** (LM_{fwd}). We prompt a code-centric LLM LM_{fwd} to produce an initial or revised program

   $$P^{(t)} = \mathrm{LM}_{\text{fwd}}\big(\mathcal{T}, P^{(t-1)}, \nabla P^{(t-1)}\big),$$

   where $\nabla P^{(t-1)}$—if available—is textual gradient feedback from the previous step. Conditioned on the task and the previous pseudo-gradient, the forward model proposes a revised candidate $P^{(t)}$. Intuitively, this component plays the role of a software engineer drafting a solution while being aware of earlier review comments.

2. **Backward Evaluation** (LM_{bwd}). A backward model returns structured feedback $F^{(t)} = [F_{\text{corr}}, F_{\text{io}}, F_{\text{eff}}, F_{\text{comp}}]$ covering correctness, I/O format, efficiency, and completeness.

3. **Gradient Propagation**. We parse the feedback into a pseudo-gradient $\nabla P^{(t)}$ that specifies where and how to edit the code.

4. **Constraint-Aware Update**. We accept an updated program $P^{(t+1)}$ that follows $\nabla P^{(t)}$ and is accompanied by a brief rationale $J^{(t+1)}$ showing that stated invariants hold.

$$\Pi^{(t+1)} = \mathrm{LM}_{\text{fwd}}\big(P^{(t)}, \nabla P^{(t)}, \mathcal{I}\big). \tag{3}$$

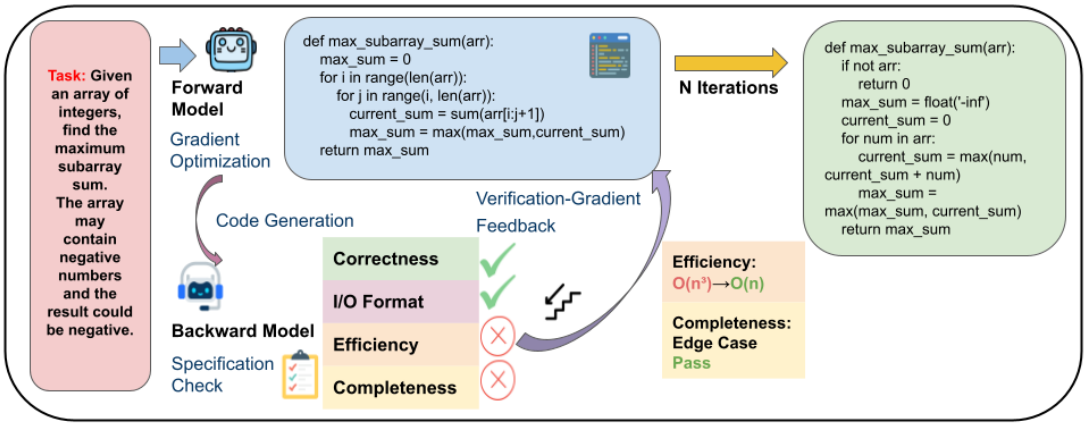

Figure 1: Overview of the gradient-style code refinement loop on the "maximum-subarray" task. First, the forward model proposes an $O(n^3)$ triple-loop solution. The backward reviewer flags inefficiency and missing edge-case handling and returns a structured critique, which is converted into a textual pseudo-gradient. This structured critique is converted into a textual pseudo-gradient that directs refinement. Guided by this gradient and a formal efficiency invariant, the optimizer produces a new, correct, and highly efficient $O(n)$ solution.

The update is accepted if The update is accepted if all invariant checks succeed; otherwise it is discarded.

3.3 Experimental Setup

To evaluate the effectiveness of CodeGrad, we establish a comprehensive experimental setup. We employ **Qwen 2.5-Coder-3B** as our *Forward Model* (Hui et al., 2024). For the *Backward Model*, we evaluate two distinct models to analyze the impact of critic capability: the lightweight, open-source **Qwen3-1.7B** (Yang et al., 2025) and the more powerful, proprietary **GPT4.1-mini** (OpenAI, 2025). All open-source models were deployed using the vLLM inference engine on an NVIDIA A6000 GPU (Kwon et al., 2023). For all experiments, we fixed the number of max refinement iterations at two. We assess performance on two complementary benchmarks:

- **HumanEval** (HE) (Chen et al., 2021) is a canonical benchmark for functional correctness, comprising 164 human-authored Python problems with rigorous unit tests. We also use its extension, **HumanEval+** (HE+) (Liu et al., 2023), which provides a more comprehensive test suite to better assess solution robustness.

- **LiveCodeBench** (LCB V6) (Naman Jain, 2024) is a continuously updated benchmark for competitive programming, including problems sourced from active platforms like LeetCode, AtCoder, and Codeforces. We use V6 in its release version, which include 175 questions from Jan 2025 to May 2025.

4 Result

Table 1: Pass@1 performance across benchmarks. The '+Code' suffix indicates code-enabled backward models. Percentages show increase over the Null backward model baseline.

Benchmark	Null	Qwen3-1.7B		GPT4.1-mini	
		Base	+Code	Base	+Code
HE	0.695	**0.787**	**0.787**	**0.811**	**0.884**
HE+	0.671	**0.713**	**0.726**	**0.756**	**0.817**
LCB V6	0.143	**0.166**	**0.160**	**0.251**	**0.201**

4.1 RQ1: To what extent does the CodeGrad framework improve code generation performance?

We first establish a baseline using a forward-only configuration with the Qwen2.5Coder-3B model and no backward optimization loop (the "Null" model in Table 1). This baseline achieves a pass@1 score of 69.5% on HumanEval, 67.1% on HumanEval+, and 14.3% on the more challenging LiveCodeBench V6. All subsequent results are evaluated relative to this starting point. Introducing the CodeGrad framework yields substantial improvements. Augmenting the forward model with a lightweight backward model (Qwen3-1.7B) boosts performance by up to 16% on LiveCodeBench. Employing a more powerful, albeit code-blind, reviewer (GPT4.1-mini) further elevates the scores, with a 17% absolute increase on HumanEval and a 76% relative increase on LiveCodeBench.

Table 2: Pass@1 by Algorithmic Category, for Forward Model with different Backward model.

Model		**Array**	**DP**	**Graph**	**Greedy**	**Hash Map**	**Math**	**Simulation**	**String**
Null		0.412	0.000	0.000	0.000	0.231	0.119	0.071	0.333
GPT4.1-mini	Base	**0.471**↑14%	**0.083**	**0.062**	**0.154**	**0.385**↑67%	**0.262**↑120%	**0.179**↑152%	**0.444**↑33%
	+Code	0.412	**0.083**	**0.062**	**0.154**	**0.385**↑67%	**0.167**↑40%	**0.143**↑101%	**0.444**↑33%
Qwen3-1.7B	Base	0.353↓14%	0.000	**0.062**	0.000	**0.308**↑33%	**0.214**↑80%	**0.107**↑51%	0.278↓17%
	+Code	0.294↓29%	0.000	**0.125**	0.000	0.154↓33%	**0.167**↑40%	**0.143**↑101%	0.333

Table 3: Pass@1 by Difficulty Level for Forward model with different backward model

Difficulty	**Null**	**Qwen3-1.7B**		**GPT4.1-mini**	
		Base	+Code	Base	+Code
Easy	0.558	0.558	0.558	**0.791**	**0.628**
Medium	0.019	**0.077**	**0.058**	**0.115**	**0.135**
Hard	0.000	**0.013**	**0.013**	**0.050**	**0.025**

4.2 RQ2: What is the impact of enabling dynamic code execution within the backward model?

Our approach also adapts partial execution for self-debugging, as demonstrated in prior work (Le et al., 2022), which demonstrates that even minimal, sandboxed probes enhance reviewer effectiveness. This research question examines the performance difference when the backward model is granted the ability to execute code probes. We compare pairs of architecturally identical reviewers—with and without the sandboxed execution capability. The results presented in Table 1 reveal that the benefit of code execution is highly contingent on the capability of the backward model and the nature of the evaluation benchmark. For the more powerful GPT4.1-mini reviewer, enabling execution on the HumanEval benchmarks yields improvements. The pass@1 score on HumanEval rises from 0.811 to 0.884, a gain of 7.3 percentage points, and on HumanEval+, it increases from 0.756 to 0.817. This demonstrates that a sophisticated reviewer can effectively leverage execution to identify and correct flaws in standard, function-based programming tasks. The effect is less pronounced for the smaller Qwen3-1.7B model, which sees a modest 1.3% gain on HumanEval+ and no change on the base HumanEval benchmark.

Conversely, the data from Table 1 shows a performance degradation on LiveCodeBench V6 when execution is enabled. The GPT4.1-mini (Code) configuration scores 5.0% lower than its non-executing counterpart, and the Qwen3-1.7B (Code) model shows a slight decrease of 0.6%.

4.3 RQ3: How does the performance of CodeGrad vary across different problem categories and difficulty levels?

To understand the conditions under which CodeGrad is most beneficial, we analyze its performance on problems grouped by algorithmic category (Table 2) and difficulty (Table 3). We analyze the problems from the LiveCodeBench dataset along two dimensions: their predefined difficulty level and their algorithmic category, which we assign based on the classification from LeetCode (LeetCode, 2025). The breakdown by category reveals that the framework's advantages are most pronounced in complex algorithmic domains. For categories like Dynamic Programming (DP), Graph, and Greedy, the baseline model fails to produce a single correct solution (0.0% pass@1). In contrast, the full CodeGrad approach successfully solves problems in these areas, demonstrating its ability to navigate complex logic and constraints. The largest relative gains are observed in Simulation (+101%), Math (+40%), and Hash map (+66%), where iterative refinement and formal checks are critical for correctness.

The analysis by difficulty level in Table III reinforces this finding. While CodeGrad offers a respectable 12.5% improvement on "Easy" problems, its impact is transformative for more challenging tasks. It boosts performance on "Medium" difficulty problems by over 610% relative to the baseline and successfully generates correct solutions for "Hard" problems where the baseline fails completely.

5 Conclusion

This work introduced CodeGrad, a framework that integrates gradient-based iterative refinement with multi-step verification for LLM-based code generation. Our results demonstrate that this hybrid approach enhances performance, achieving gains on benchmarks like LiveCodeBench. The framework proves particularly effective for complex problems requiring strict constraint adherence.

6 Limitations

Verifier dependence and specification drift. The pseudo-gradient quality depends on the backward model's critique and the evaluation prompt. When the critique omits corner cases or overstates correctness, refinements can converge to solutions that satisfy the critique but not the full specification. Future work can integrate static analysis, SMT-assisted checks, property-based testing, and fuzzing to produce more comprehensive, grounded critiques.

Generality and portability. Our experiments focus on Python code generation tasks. Future work can extend to other languages, richer ecosystems, and non-code tasks that may require new invariants, tooling, and I/O protocols.

References

Yasin Abbasi Yadkori, Ilja Kuzborskij, András György, and Csaba Szepesvari. 2024. To believe or not to believe your llm: Iterative prompting for estimating epistemic uncertainty. *Advances in Neural Information Processing Systems*, 37:58077–58117.

Anisha Agarwal, Aaron Chan, Shubham Chandel, Jinu Jang, Shaun Miller, Roshanak Zilouchian Moghaddam, Yevhen Mohylevskyy, Neel Sundaresan, and Michele Tufano. 2024. Copilot evaluation harness: Evaluating llm-guided software programming. *arXiv preprint arXiv:2402.14261*.

Brett A Becker, Paul Denny, James Finnie-Ansley, Andrew Luxton-Reilly, James Prather, and Eddie Antonio Santos. 2023. Programming is hard-or at least it used to be: Educational opportunities and challenges of ai code generation. In *Proceedings of the 54th ACM Technical Symposium on Computer Science Education V. 1*, pages 500–506.

Jialun Cao, Yaojie Lu, Meiziniu Li, Haoyang Ma, Haokun Li, Mengda He, Cheng Wen, Le Sun, Hongyu Zhang, Shengchao Qin, and 1 others. 2025. From informal to formal–incorporating and evaluating llms on natural language requirements to verifiable formal proofs. *arXiv preprint arXiv:2501.16207*.

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, Alex Ray, Raul Puri, Gretchen Krueger, Michael Petrov, Heidy Khlaaf, Girish Sastry, Pamela Mishkin, Brooke Chan, Scott Gray, and 39 others. 2021. Evaluating large language models trained on code.

Sarah Fakhoury, Aaditya Naik, Georgios Sakkas, Saikat Chakraborty, and Shuvendu K Lahiri. 2024. Llm-based test-driven interactive code generation: User study and empirical evaluation. *IEEE Transactions on Software Engineering*.

Jonas Gehring, Kunhao Zheng, Jade Copet, Vegard Mella, Quentin Carbonneaux, Taco Cohen, and Gabriel Synnaeve. 2024. Rlef: Grounding code llms in execution feedback with reinforcement learning. *arXiv preprint arXiv:2410.02089*.

Qiuhan Gu. 2023. Llm-based code generation method for golang compiler testing. In *Proceedings of the 31st ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, pages 2201–2203.

Shibo Hao, Yi Gu, Haotian Luo, Tianyang Liu, Xiyan Shao, Xinyuan Wang, Shuhua Xie, Haodi Ma, Adithya Samavedhi, Qiyue Gao, and 1 others. 2024. Llm reasoners: New evaluation, library, and analysis of step-by-step reasoning with large language models. *arXiv preprint arXiv:2404.05221*.

Binyuan Hui, Jian Yang, Zeyu Cui, Jiaxi Yang, Dayiheng Liu, Lei Zhang, Tianyu Liu, Jiajun Zhang, Bowen Yu, Keming Lu, and 1 others. 2024. Qwen2. 5-coder technical report. *arXiv preprint arXiv:2409.12186*.

Susmit Jha, Sumit Kumar Jha, Patrick Lincoln, Nathaniel D Bastian, Alvaro Velasquez, and Sandeep Neema. 2023. Dehallucinating large language models using formal methods guided iterative prompting. In *2023 IEEE International Conference on Assured Autonomy (ICAA)*, pages 149–152. IEEE.

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

Hung Le, Yue Wang, Akhilesh Deepak Gotmare, Silvio Savarese, and Steven Chu Hong Hoi. 2022. Coderl: Mastering code generation through pretrained models and deep reinforcement learning. *Advances in Neural Information Processing Systems*, 35:21314–21328.

LeetCode. 2025. Problemset. `https://leetcode.com/problemset/`. Accessed: 2025-07-11.

Yujia Li, David Choi, Junyoung Chung, Nate Kushman, Julian Schrittwieser, Rémi Leblond, Tom Eccles, James Keeling, Felix Gimeno, Agustin Dal Lago, and 1 others. 2022. Competition-level code generation with alphacode. *Science*, 378(6624):1092–1097.

Fang Liu, Yang Liu, Lin Shi, Houkun Huang, Ruifeng Wang, Zhen Yang, Li Zhang, Zhongqi Li, and Yuchi Ma. 2024. Exploring and evaluating hallucinations in llm-powered code generation. *arXiv preprint arXiv:2404.00971*.

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.

Shaoying Liu and Rolf Adams. 1995. Limitations of formal methods and an approach to improvement. In *Proceedings 1995 Asia Pacific Software Engineering Conference*, pages 498–507. IEEE.

Terufumi Morishita, Gaku Morio, Atsuki Yamaguchi, and Yasuhiro Sogawa. 2024. Enhancing reasoning capabilities of llms via principled synthetic logic corpus. *Advances in Neural Information Processing Systems*, 37:73572–73604.

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

OpenAI. 2025. Introducing GPT-4.1, GPT-4.1 Mini, and GPT-4.1 Nano. https://openai.com/index/gpt-4-1/. Accessed: July 7, 2025.

Shuyin Ouyang, Jie M Zhang, Mark Harman, and Meng Wang. 2023. Llm is like a box of chocolates: the nondeterminism of chatgpt in code generation. *arXiv e-prints*, pages arXiv–2308.

Zoltán Ságodi, István Siket, and Rudolf Ferenc. 2024. Methodology for code synthesis evaluation of llms presented by a case study of chatgpt and copilot. *Ieee Access*.

Wei Shen and Chuheng Zhang. 2024. Policy filtration in rlhf to fine-tune llm for code generation. *arXiv preprint arXiv:2409.06957*.

Noah Shinn, Federico Cassano, Ashwin Gopinath, Karthik Narasimhan, and Shunyu Yao. 2023. Reflexion: Language agents with verbal reinforcement learning. *Advances in Neural Information Processing Systems*, 36:8634–8652.

Alexey Svyatkovskiy, Shao Kun Deng, Shengyu Fu, and Neel Sundaresan. 2020. Intellicode compose: Code generation using transformer. In *Proceedings of the 28th ACM joint meeting on European software engineering conference and symposium on the foundations of software engineering*, pages 1433–1443.

Jianxun Wang and Yixiang Chen. 2023. A review on code generation with llms: Application and evaluation. In *2023 IEEE International Conference on Medical Artificial Intelligence (MedAI)*, pages 284–289. IEEE.

Jim Woodcock, Peter Gorm Larsen, Juan Bicarregui, and John Fitzgerald. 2009. Formal methods: Practice and experience. *ACM computing surveys (CSUR)*, 41(4):1–36.

Chenwei Xiong, Cheng Liu, Huawei Li, and Xiaowei Li. 2024. Hlspilot: Llm-based high-level synthesis. In *Proceedings of the 43rd IEEE/ACM International Conference on Computer-Aided Design*, pages 1–9.

An Yang, Anfeng Li, Baosong Yang, Beichen Zhang, Binyuan Hui, Bo Zheng, Bowen Yu, Chang Gao, Chengen Huang, Chenxu Lv, and 1 others. 2025. Qwen3 technical report. *arXiv preprint arXiv:2505.09388*.

Tong Ye, Yangkai Du, Tengfei Ma, Lingfei Wu, Xuhong Zhang, Shouling Ji, and Wenhai Wang. 2025. Uncovering llm-generated code: A zero-shot synthetic code detector via code rewriting. In *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 39, pages 968–976.

Mert Yuksekgonul, Federico Bianchi, Joseph Boen, Sheng Liu, Zhi Huang, Carlos Guestrin, and James Zou. 2024. Textgrad: Automatic" differentiation" via text. *arXiv preprint arXiv:2406.07496*.

Yadong Zhang, Shaoguang Mao, Tao Ge, Xun Wang, Adrian de Wynter, Yan Xia, Wenshan Wu, Ting Song, Man Lan, and Furu Wei. 2024. Llm as a mastermind: A survey of strategic reasoning with large language models. *arXiv preprint arXiv:2404.01230*.

Ziyao Zhang, Chong Wang, Yanlin Wang, Ensheng Shi, Yuchi Ma, Wanjun Zhong, Jiachi Chen, Mingzhi Mao, and Zibin Zheng. 2025. Llm hallucinations in practical code generation: Phenomena, mechanism, and mitigation. *Proceedings of the ACM on Software Engineering*, 2(ISSTA):481–503.

Zhehua Zhou, Jiayang Song, Kunpeng Yao, Zhan Shu, and Lei Ma. 2024. Isr-llm: Iterative self-refined large language model for long-horizon sequential task planning. In *2024 IEEE International Conference on Robotics and Automation (ICRA)*, pages 2081–2088. IEEE.

A Appendix

A. System Overview

CodeGrad couples a forward code-generation model with a backward verification model inside a differentiable prompt-optimization loop powered by TextGrad. The forward model proposes code; the backward model provides structured critique that is treated as a loss signal. Gradients of this loss with respect to the code text are used to steer the next generation step under a set of formal invariants. We demonstrate this in two evaluation settings: HumanEval (function completion) and

LiveCodeBench (competitive programming with stdin/stdout I/O).

Concretely, we use:

- A forward engine (code generation) to produce candidate programs.
- A backward engine (verifier/critic) to evaluate correctness and provide actionable feedback.
- TextGrad primitives: differentiable textual variables, a black-box LLM wrapper, and a formal optimizer enforcing invariants while updating the solution.

B. Prompt Structure

B.1 Forward (Code Generation) Prompts

HumanEval Forward Prompt

```
You are a Python code generation
    assistant. You will receive a Python
     function signature and docstring,
    potentially with feedback on
    previous attempts. Do not catch and
    raise the error when try to fixing
    to code. Your SOLE task is to output
     the improved or completed Python
    function code. Do NOT output any
    other text, explanation, preamble,
    markdown formatting, or any wrapper
    around the code except for these
    specific tags. You can use <execute>
     tags to run Python code and see the
     output, for example: <execute>print
    ('Hello World!')</execute>
```

LiveCodeBench Forward Prompt

```
You are a competitive programming expert
     specializing in Python code
    generation. You will receive
    competitive programming problems and
     generate efficient, correct Python
    solutions. Your task is to output
    only the Python code solution that
    reads from stdin and writes to
    stdout. Focus on: correct logic,
    efficient algorithms, proper input/
    output handling, edge case coverage.
     Use appropriate data structures (
    lists, sets, dictionaries, heaps,
    etc.) and algorithms (sorting,
    searching, dynamic programming, etc
    .). Ensure your solution handles all
     edge cases and follows competitive
    programming best practices. Do NOT
    output any explanations, markdown
    formatting, or any wrapper around
    the code except for these specific
    tags. You can use <execute> tags to
    run Python code and see the output,
    for example: <execute>print('Hello
    World!')</execute>
```

B.2 Backward (Verifier/Critic) Prompts

HumanEval Backward Prompt

```
You are an expert Python code reviewer
    evaluating code for the HumanEval
    benchmark with python code format.
    You will receive Python code and the
     original problem description (
    docstring and signature). Evaluate
    if the code correctly and completely
     implements the requirements
    described in the problem description
    . Provide concise, specific, and
    actionable feedback for improvement.
     Focus *only* on correctness
    according to the description. You
    can use <execute> tags to run Python
     code and see the output, for
    example: <execute>print('Hello World
    !')</execute> Otherwise, provide *
    only* the feedback
```

LiveCodeBench Backward Prompt

```
You are an expert competitive
    programming code reviewer evaluating
     Python solutions. You will receive
    Python code and the original
    competitive programming problem
    description. Evaluate if the code
    correctly implements the competitive
     programming problem requirements.
    Consider: correctness, efficiency,
    input/output format compliance (
    stdin/stdout), edge case handling.
    You are encouraged to use <execute
    >...</execute> tags to run code
    snippets, test edge cases, or check
    outputs. For example: <execute>print
    ('Test output')</execute> or <
    execute>len([1,2,3])</execute>. Use
    code execution to support your
    reasoning and provide more concrete
    feedback. Check for common
    competitive programming issues:
    incorrect algorithm, wrong data
    structures, inefficient solutions,
    missing edge cases, wrong input/
    output format. Provide concise,
    specific, and actionable feedback
    for improvement. Focus *only* on
    competitive programming correctness
    and completeness according to the
    problem description. If the code is
    correct, state clearly that it meets
     all requirements. Otherwise,
    provide *only* the feedback
```

B.3 Evaluation (Loss) Prompt Templates

HumanEval Evaluation Prompt

```
Evaluate this code solution for the
    following problem:

Problem:
[problem text]

Solution:
[candidate code with any <execute>
    blocks resolved to include outputs]
```

```
Provide specific feedback on correctness
     and completeness.
```

LiveCodeBench Evaluation Prompt

```
Evaluate this competitive programming
    solution:

Problem: [problem text]

Solution:
‘‘‘python
[candidate code with any <execute>
    blocks resolved]
‘‘‘

Evaluate the code for:
1. Correctness: Does the logic correctly
     solve the problem?
2. Input/Output: Does it properly read
    from stdin and write to stdout?
3. Efficiency: Is the solution efficient
     enough for competitive programming
    constraints?
4. Edge Cases: Does it handle edge cases
     mentioned in the problem?
5. Completeness: Does it implement all
    requirements?

You can use <execute> tags to test code
    snippets or check specific aspects.
    For example:
- <execute>print("Testing input format")
    </execute>
- <execute>len([1,2,3])</execute>

Provide specific, actionable feedback.
    Focus on competitive programming
    correctness.
```

C. Multi-Step Verification

CodeGrad augments the backward model with lightweight execution to ground feedback:

- The pipeline resolves <execute>.../execute> blocks found in candidate code or in the backward model feedback with a constrained interpreter, returning captured outputs inline.
- Early stopping: if the critic asserts correctness with strong indicators (e.g., “code is correct”, “meets all requirements”), the loop terminates.
- Safety/robustness: code blocks are length-limited, filtered for risky constructs, and executed under strict timeouts.

D. Gradient-Based Refinement

Let x denote the candidate program text. The backward model produces textual feedback $\ell(x)$. TextGrad differentiates this loss signal with respect to x and performs a constrained update using a forward LLM under explicit invariants.

Core loop:

1. Initialize a differentiable text variable from the forward engine output.
2. Compute loss by prompting the backward engine with the problem and current code.
3. Backpropagate through TextGrad to obtain gradients.
4. Apply a constrained update with a formal optimizer, providing invariants and the forward engine to propose an improved candidate.
5. Repeat up to a maximum number of iterations or until an early-stopping trigger.

Invariants. Invariants encode domain constraints that must be preserved during refinement. Representative sets:

- HumanEval: maintain the original function signature; implement all described requirements; ensure syntactically valid Python; handle specified edge cases.
- LiveCodeBench: correct stdin/stdout I/O; edge-case coverage; efficiency appropriate for constraints; avoid infinite loops; use appropriate data structures/algorithms.

Listing 1: Constrained update call

```
formal_optimizer =
    FormalMethodsOptimizerCode(
  parameters=[optimizable_code],
  engine=forward_engine,
  invariants=invariants,
  verbose=1,
  new_variable_tags=("<IMPROVED_VARIABLE
      >", "</IMPROVED_VARIABLE>"),
  loss_function=loss_function
)
formal_optimizer.step()
// CodeGrad constrained update
```

E. Task Formatting

HumanEval. Problems provide a function signature and docstring. The forward model returns the function body; post-processing removes auxiliary tags/markdown and repairs unterminated triple quotes to ensure parsability.

LiveCodeBench. Each problem is normalized to a single prompt comprising the question text and, if present, starter code; otherwise, solutions must read stdin and write stdout. The evaluation prompt enumerates correctness, I/O, efficiency, and edge-case criteria.

F. Engineering Notes

- Timeouts and concurrency: generation, verification, and optimization steps use conservative timeouts to avoid pipeline stalls.
- Caching: simple memoization for `<execute>` blocks speeds repeated checks.
- Post-processing: removal of helper tags and code fences; balancing triple quotes; ensuring strict code-only outputs for downstream evaluators.
- Early-stop heuristic: string-based positive/negative indicators trade recall/precision to cap iteration count while preserving quality.

G. Hyperparameters and Defaults

- Iterations: small (e.g., 2–3) for efficient improvement.
- Temperature: typically `0.0` for determinism in competitive programming.
- Model endpoints: OpenAI-compatible servers for code generation and verification; may be hosted separately.
- Execution limits: `<execute>` lines and characters constrained; snippet execution timeouts on the order of sub-seconds.

H. Pseudocode

Listing 2: CodeGrad loop pseudocode

```
Inputs: problem P, forward LLM F,
    backward LLM B, invariants I, steps
    T
x0 <- F(P)                          #
    initial code
for t in 1..T do
  y  <- resolve_execute_tags(xt)  #
    safe, timed execution
  L  <- B( format_eval_prompt(P, y) )
     # textual feedback
  if is_correct(L): break
  g  <- textgrad_backward(L, xt)
  xt+1 <- formal_optimize(xt, g; F, I
    )
return sanitize(xt)
```

J. LiveCodeBench Problem Categories

This appendix reports categories for LiveCodeBench tasks as produced by our evaluation aggregation script that performs LLM-based categorization (see the analysis utility for LiveCodeBench aggregation). The categorizer prompts an expert model to choose a single primary algorithmic category per problem from a fixed, standardized list.

J.1 Category List (Fixed Choices)

The LLM is instructed to select exactly one of the following categories: array, string, math, dp (dynamic programming), graph, tree, greedy, simulation, binary_search, stack_queue , heap, hash_map, linked_list, backtracking, bit_manipulation, geometry, sorting.

J.2 Categorization Procedure

For each problem, the aggregation utility provides the title and statement to the LLM and requests a single best-fit category from the fixed list above, with temperature set to zero for determinism. The chosen category is cached per question title and reused across groups to ensure a consistent mapping. If the LLM returns an out-of-vocabulary label, it is mapped to `other`.

J.3 Categorization Prompt

We include the exact categorization prompt used by the aggregation utility. The model is instructed to respond with only one category name from the list.

```
You are an expert in competitive
    programming and algorithmic problem
    classification.

Given the following programming problem,
     categorize it by the primary
    algorithmic approach or data
    structure needed to solve it.

Title: {question_title}
Problem Description: {question_content}

Choose ONE category from the following
    list that best describes the primary
     algorithmic approach:

1. array - Array manipulation, sliding
    window, two pointers
2. string - String processing, pattern
    matching, parsing
3. math - Mathematical problems, number
    theory, combinatorics
4. dp - Dynamic programming, memoization
5. graph - Graph algorithms, DFS, BFS,
    shortest path
6. tree - Tree data structures, binary
    trees, tree traversal
7. greedy - Greedy algorithms,
    optimization
8. simulation - Simulation problems,
    step-by-step execution
9. binary_search - Binary search, divide
     and conquer
10. stack_queue - Stack, queue,
    monotonic stack/queue
11. heap - Priority queue, heap
    operations
```

```
12. hash_map - Hash table, frequency
    counting
13. linked_list - Linked list
    manipulation
14. backtracking - Backtracking,
    recursive search
15. bit_manipulation - Bit operations,
    bitwise algorithms
16. geometry - Geometric algorithms,
    computational geometry
17. sorting - Sorting algorithms, custom
     comparators
18. other - Other algorithms or complex
    combinations

Respond with ONLY the category name (e.g
    ., "array", "dp", "graph") without
    any explanation or additional text.
```

J.3 Reporting in the Paper

We recommend summarizing per-category counts and performance, including cross-tabs by difficulty when available. For example:

- Distribution: fraction of problems per primary category (Strings, Graphs, DP, Greedy, etc.).
- Gains: pass-rate deltas of CodeGrad vs. baseline per category, highlighting domains where multi-step verification helps most (e.g., Simulation, Math, Hash Map).
- Error profiles: common failure modes per category (e.g., I/O mishandling in Simulation; off-by-one in Arrays; complexity blow-ups in Graph/DP).

J.4 Notes

This categorization is intentionally lightweight and reproducible without external annotations. If official labels are available in a given LiveCodeBench release, they can replace or complement our heuristic labels directly.

J.5 Category Listing (Grouped by Category)

From the existing aggregation outputs, we derive a grouped listing that shows each category followed by the identifiers of all problems assigned to it. This format emphasizes category coverage and makes it easy to locate families of problems.

Category	Problems
array	3674, 3705, 3717, 3723, 3725, 3731, 3747, 3751, 3778, 3788, 3789, abc388_c, abc390_a, abc391_b, abc393_d, abc395_a, abc396_a
backtracking	3799
binary_search	3762
bit_manipulation	abc390_d, abc396_e
dp	3677, 3722, 3737, 3765, 3777, 3786, 3832, abc387_f, abc390_e, abc393_f, abc396_c, arc194_d
geometry	3763, 3781, abc389_d
graph	3720, 3809, abc388_f, abc394_e, abc394_f, abc394_g, abc395_e, abc396_d, abc397_e, abc397_g, abc398_g, abc399_c, abc400_d, arc191_d, arc193_a, arc193_b
greedy	3743, 3776, 3779, 3791, abc388_e, abc388_g, abc389_e, abc396_g, abc400_g, arc191_a, arc194_b, arc194_c, arc196_a
hash_map	3736, 3750, 3753, abc390_f, abc392_b, abc392_c, abc392_g, abc395_c, abc397_c, abc397_f, abc398_b, abc398_c, arc195_a
heap	3759, 3764, abc391_f
math	3696, 3697, 3704, 3714, 3739, 3754, 3760, 3783, 3785, 3801, 3811, 3817, abc387_a, abc387_b, abc387_c, abc389_b, abc389_g, abc390_b, abc390_g, abc391_g, abc392_a, abc392_d, abc393_a, abc393_e, abc395_f, abc396_f, abc397_a, abc397_d, abc399_d, abc399_f, abc400_a, abc400_b, abc400_c, abc400_e, arc190_d, arc191_c, arc192_d, arc192_e, arc195_b, arc195_c, arc196_b, arc196_c
simulation	3708, 3721, 3733, 3744, 3768, 3773, 3794, 3795, abc388_b, abc388_d, abc389_f, abc390_c, abc391_d, abc391_e, abc392_f, abc394_c, abc395_b, abc398_d, abc399_b, arc190_a, arc190_c, arc192_a, arc192_b, arc193_d, arc194_a, arc194_e, arc195_d, arc196_d
sorting	3748, abc394_b
stack_queue	abc394_d, abc396_b
string	3692, 3701, 3709, 3770, 3771, 3784, 3793, 3805, abc388_a, abc389_a, abc391_a, abc393_b, abc394_a, abc397_b, abc398_a, abc398_f, abc399_a, abc399_e
tree	3687, arc195_e