Simple Test-Time Refinement for Plot-to-Code Generation via Visual-Code Diagnostics

Peiming Guo[†], **Jinpeng Huang**[†], **Yangbin Sun**[†], **Gaoxiong Cao**[‡], **Ensheng Shi**[‡], **Yuchi Ma**[‡], **Yu Zhang**[§], **Meishan Zhang**[†*], **Baotian Hu**[†], **Min Zhang**[†]
[†] Institute of Computing and Intelligence, Harbin Institute of Technology (Shenzhen),
[‡] Huawei CodeArts Model Team, HUAWEI,
[§] School of Computer Science and Technology, Soochow University
guopeiming.gpm@gmail.com, mason.zms@gmail.com

Abstract

Plot-to-code generation has been greatly advanced by recent multimodal large language models (MLLMs). However, existing studies mainly focus on one-pass generation or iterative sampling guided by verifier-based filtering, which does not align with the typical human programming workflow of generate first, then iteratively error-location and fix. This naturally motivates a test-time refinement paradigm for plot-to-code generation. In this work, we present the first study on plot-to-code generation via iterative test-time refinement. We propose a Visual-Code Diagnostics framework that enables refinement through visual feedback from source code. Specifically, we first categorize generation errors into four coarse-grained types, and then train lightweight discriminators to recognize and localize these errors based on the rendered visual outputs and corresponding code. Unlike traditional compiler- or debugger-based verification, our approach performs iterative refinement guided by one diagnostics system, enabling more effective semantic and visual correction during test time. Extensive experiments demonstrate that our method consistently improves generation quality across diverse generators and significantly outperforms strong test-time scaling baselines.

1 Introduction

Plot-to-code generation [1–5] aims to synthesize executable plotting programs that faithfully reproduce a given reference chart. This task has attracted growing attention due to its applications in visual chart understanding [6, 7], data visualization [1, 8], and code generation [9–11]. Recent advances in large multimodal language models (MLLMs) [12–18] have substantially improved performance by enabling joint visual and textual reasoning. Nevertheless, generating visually accurate plotting code remains challenging, especially for complex charts requiring precise structural layouts, data mappings, and stylistic consistency.

Most existing approaches [19, 4, 20, 3, 5] formulate plot-to-code generation as a direct program synthesis problem, where the model generates plotting code from the input chart in a single pass. Although recent MLLMs [21, 22, 18, 23–25] demonstrate strong generation capabilities, such methods remain highly sensitive to visual-semantic errors, including incorrect data encodings, missing visual elements, layout and styling mismatches. In practice, we observe that these failures exhibit recurring visual patterns and can be grouped into several coarse-grained categories of visual errors. Once such errors appear in generated code, they are often difficult to correct through single-pass decoding alone.

* Corresponding author: Meishan Zhang.

40th Conference on Neural Information Processing Systems (NeurIPS 2026).

To alleviate generation failures, recent works have begun exploring test-time scaling strategies such as iterative repeated sampling [26]. Repeated sampling methods improve performance by generating multiple candidate programs and selecting outputs using execution-based or verifier-based filtering. However, these approaches repeatedly regenerate the entire program, leading to substantial inference cost and token consumption, especially for long and structurally complex code. More importantly, existing verification signals are typically coarse-grained, only indicating whether a generated result is acceptable without identifying the underlying causes of visual discrepancies. Such paradigms do not align with the typical human programming workflow, where developers usually generate an initial code first, then iteratively inspect outputs, localize errors, and revise only problematic code regions. This naturally motivates a test-time refinement paradigm for plot-to-code generation.

Iterative refinement provides a more promising direction by progressively correcting generation errors through multiple rounds of revision [27]. However, effective refinement for plot-to-code generation requires more than holistic correctness evaluation. Existing approaches still lack fine-grained visual-semantic diagnostics that can explicitly connect rendering errors with suspicious code regions. As a result, refinement often remains inefficient, requiring redundant regeneration and suffering from accumulated error propagation. This limitation becomes particularly pronounced in multi-agent refinement settings, where the absence of structured diagnostic feedback leads to ineffective interactions between generation and revision agents.

To address these challenges, we propose a Visual-Code Diagnostics framework for efficient test-time refinement in plot-to-code generation. As shown in Figure 1, instead of repeatedly regenerating the full plotting program, our framework iteratively diagnoses visual-code inconsistencies from rendered charts and performs localized code patch refinement. Specifically, we introduce a lightweight visual-code diagnostic verifier that identifies visual discrepancy categories and localizes suspicious code regions associated with rendering errors. The resulting diagnostic feedback serves as a structured interaction bridge for iterative refinement, enabling targeted code modification with substantially lower token consumption and improved correction efficiency.

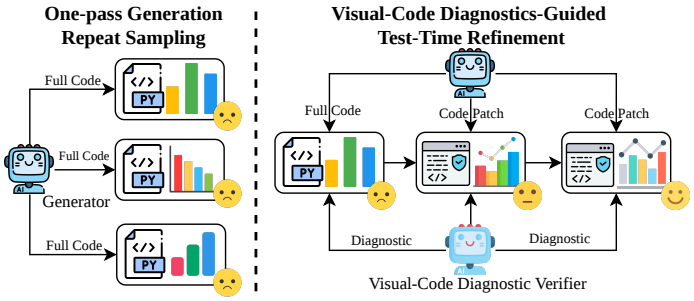


Figure 1: Comparison of one-pass generation-based repeat sampling and our visual-code diagnostic-guided test-time refinement.

To train the verifier, we construct a visual-code error discrimination dataset using controlled code mutations generated by MLLMs to simulate realistic plotting errors. We further employ an improved line-level Levenshtein distance algorithm [28, 29] to localize modified code regions and build structured supervision signals that associate visual discrepancies with corresponding code edits, enabling the verifier to learn visual-semantic diagnostic reasoning over chart-code pairs. At inference time, our framework follows an iterative refinement pipeline: generate, render, diagnose, and patch-refine. Unlike conventional test-time scaling approaches that repeatedly regenerate complete programs, our method only revises localized suspicious code segments during refinement, significantly improving inference efficiency while maintaining strong correction capability.

We evaluate our approach on Chart2Code-160k [3], ChartMimic [1], and Plot2Code [2]. Experimental results demonstrate that our method consistently improves generation quality across multiple generators and substantially outperforms strong test-time scaling baselines. Furthermore, compared with existing iterative refinement methods, our framework achieves superior performance with considerably lower inference cost.

Our contributions can be summarized as follows:

- We present a simple test-time refinement for plot-to-code generation, creating a diagnostic system to guide the iterative refinement while reducing inference cost.
- We build a lightweight visual-code diagnostic verifier that is able to detect visual discrepancies and localize them to specific code regions based on an automatically built dataset.

- We show that our method is effective on plot-to-code benchmarks, outperforming strong baselines, including one-pass generation and other test-time scaling strategies at low cost. [2]

2 Related Work

Plot-to-Code Generation. With the rapid development of multimodal large language models (MLLMs) [21, 22, 18, 23, 24], plot-to-code generation has attracted increasing attention. Recent works have constructed various benchmarks and training datasets for this task, including ChartMimic [1], Plot2Code [2], ChartEdit [30], ChartX [31], ReChartPrompt-240K [4], Chart2Code-160K [3], and Chart2Code53 [5]. VisCodex [32] and JanusCode [33] further extend multimodal code generation to broader tasks such as webpage and UI generation.

Beyond dataset construction, researchers have explored different training paradigms for improving plot-to-code generation. Zhang et al. [19] optimize generation using preference learning and DPO, while ChartMaster [4] and MSRL [20] employ reinforcement learning with visual and attribute-level rewards. ChartCoder [3] and Chart2Code53 [5] perform supervised fine-tuning on large-scale chart-code datasets. Most existing methods formulate plot-to-code as direct program synthesis, where the entire plotting program is generated in a single pass. In contrast, our work focuses on efficient test-time refinement through visual-code diagnostics.

Test-Time Scaling and Iterative Refinement. Test-time scaling [26, 34] has recently become an effective strategy for improving code generation performance. A common approach is repeated sampling [26], where multiple candidate programs are generated and filtered using execution feedback, reward models, or verifier-based selection [35]. However, such methods repeatedly regenerate complete programs, resulting in high inference cost and suboptimal results.

Recent works have also explored iterative and multi-agent refinement for plot-co-code generation. METAL [27] introduces a multi-agent framework for chart-to-code generation with iterative collaboration between multiple GPT-4o-based agents. While effective, existing refinement frameworks still rely on holistic conversational feedback and repeated full-program generation, making them vulnerable to inefficient revisions and error propagation.

Different from prior work, our framework performs localized patch-based refinement through visual-code diagnostics. By identifying visual discrepancy categories and suspicious code regions, our method enables targeted iterative correction with substantially improved inference efficiency.

3 Method

Given a reference chart image I, plot-to-code generation aims to synthesize an executable plotting program C using an MLLM generator D, such that the rendered chart $\widehat{I}$ faithfully matches I. Instead of treating this task as one-pass generation, we propose the test-time refinement strategy, where an initially generated program is iteratively corrected according to visual-code diagnostic feedback. Specifically, we introduce visual-code diagnostic verification: $V(I, \widehat{I}, C) \rightarrow (e, L)$, where e denotes the visual error category and L denotes the suspicious code regions associated with the discrepancy. Based on (e, L), the generator performs localized patch-based refinement instead of full-program regeneration, enabling more efficient and targeted correction.

This section introduces our framework (Figure 2), including visual error taxonomy and dataset construction, diagnostic verifier training, and visual-code diagnostics-guided test-time refinement.

3.1 Visual-Code Diagnostic Verification

We model visual-code diagnostics as structured errors that can be detected from rendered outputs and localized to specific code segments. This subsection describes how we define error structures, construct supervision, and train the verifier.

[2] Our code is publicly available at `https://github.com/guopeiming/plot_to_code_refine`.

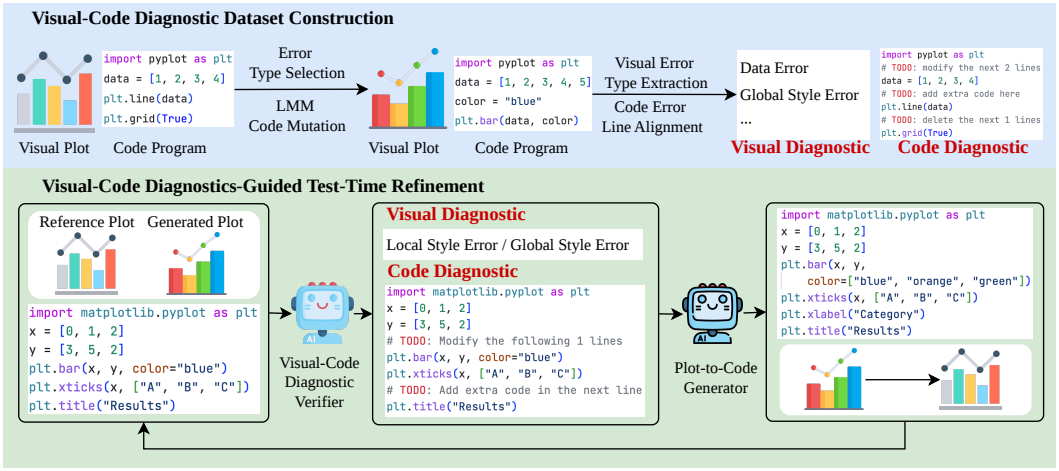


Figure 2: Overview of test-time refinement for plot-to-code generation via visual-code diagnostics.

3.1.1 Error Modeling and Diagnostic Categories

To better understand failure patterns in plot-to-code generation, we conduct a systematic analysis of real-world errors produced by several representative MLLMs, including ChartMaster-7B [4], Qwen-VL-2B-Instruct, Qwen-VL-32B-Instruct [12], and GPT-4o [13]. Our analysis shows that visual discrepancies typically exhibit structured and localized patterns, rather than arbitrary mismatches. Instead of defining a rigid taxonomy, we adopt a set of coarse-grained diagnostic categories that capture common failure modes in Matplotlib-based [36, 37] plot generation. These categories broadly include: 1). compiler failures that prevent successful rendering, 2). data-related error in plotted shapes or values, 3). local styling errors affecting specific visual elements, and 4). global layout or styling inconsistencies involving plot-level components.

Importantly, these errors are often associated with localized code regions, suggesting that effective refinement requires not only detecting whether a chart is incorrect, but also identifying where the error likely originates in the code. This observation motivates our diagnostic verifier, which produces structured feedback grounded in code structure and visual discrepancies. Detailed error definitions, examples, and distribution statistics of real-world errors are provided in Appendix A.

3.1.2 Structured Diagnostic Dataset Construction

To train the visual-code diagnostic verifier, we construct a structured dataset that provides supervision for both visual error detection and code localization. Each sample consists of a generated plot, a reference plot, and the code that produces the generated plot, with supervision signals including the visual error type and code-grounded diagnostic annotations.

We build this dataset from correct chart-code pairs in Chart2Code-160K [3]. To simulate realistic errors, we perform error-aware code mutation using GPT-4o [13]. Specifically, for each seed program, we generate three mutated variants corresponding to data error, local style error, and global style error, while keeping other parts of the code unchanged. Compiler errors are excluded from mutation, as they can be directly identified via execution feedback. To improve diversity, mutation types are applied in random sequential orders across samples.

Given one seed program and its mutated variants, we construct pairwise diagnostic samples by selecting two programs (C_1, C_2), where C_1 is treated as the generated code and C_2 as the reference. The corresponding plots (I_1, I_2) are rendered, and the visual error label is determined by the mutation type that differentiates C_1 from C_2. This pairwise construction yields diverse visual-code discrepancies and expands the dataset size (up to $12\times$ per seed example).

To obtain code-region supervision, we introduce a line-level alignment algorithm based on an improved Levenshtein distance algorithm [28]. The method identifies inserted, deleted, and modified code segments between C_1 and C_2, and maps them to candidate error regions. The algorithm details are placed in Appendix B.2. Instead of predicting explicit line indices, we represent localization

signals by inserting `TODO` comments into the code near the identified regions, forming structured diagnostic annotations that are more robust for the generation paradigm of MLLMs.

This process converts synthetic visual-code differences into structured supervision and enables the scalable construction of structured diagnostic supervision without manual annotation, allowing the verifier to learn both error classification and fine-grained code localization. Detailed construction procedures are provided in Appendix B.

3.1.3 Visual-Code Diagnostic Verifier Training

We train a compact visual-code diagnostic verifier that takes the reference chart I, rendered chart $\widehat{I}$, and generated code C as input, and outputs structured visual-code diagnostics (e, L), where visual diagnostic e is the visual error type from our taxonomy and code diagnostic L is localized code feedback. Specifically, instead of predicting line indices, it generates the code with `TODO` comments inserted near erroneous regions, providing generation-friendly and context-aware localization.

We train the verifier using a combination of supervised fine-tuning and reinforcement learning. While supervised training teaches the model the basic diagnostic format, reinforcement learning further improves the accuracy and usefulness of the diagnostic outputs. To this end, we design a reward function for GRPO training [38] that evaluates both visual error prediction and code localization quality: $r = r_{\text{visual}} + r_{\text{code}}$. Both components are computed using Jaccard similarity [39]: $r(X, Y) = \frac{|X \cap Y|}{|X \cup Y|}$, where X and Y denote the predicted and ground-truth sets, respectively.

For visual diagnostics, r_{visual} measures the overlap between predicted and ground-truth error types. For code diagnostics, we represent each predicted diagnostic action as a tuple (i, a, j), where i denotes the line position of the inserted `TODO` comment, $a \in \{\texttt{add}, \texttt{modify}, \texttt{delete}\}$ denotes the repair action type, and j denotes the number of affected lines. The reward r_{code} is computed as the Jaccard similarity between predicted and ground-truth sets of such tuples.

This training formulation aligns the verifier with its diagnostic role: it encourages accurate identification of visual discrepancies while producing precise, code-grounded localization signals. Notably, a compact 2B model is sufficient to perform joint visual-code diagnostics, eliminating the need for complex multi-agent reasoning while enabling efficient test-time refinement with significantly reduced inference cost.

3.2 Visual-Code Diagnostics-Guided Test-Time Refinement

Given a reference chart I, the generator D first produces an initial program C_0, which is rendered into $\widehat{I}_0$. The diagnostic verifier then analyzes the visual discrepancy between I and $\widehat{I}_t$, together with the associated code C_t, and produces structured diagnostic feedback (e_t, L_t), where e_t denotes a coarse-grained error category and L_t indicates the suspicious code region.

Based on this feedback, the generator produces a localized code patch ΔC_t targeting the identified region L_t. The patch is then merged back into the original program to obtain the refined code:

$$(e_t, L_t) = V(I, \widehat{I}_t, C_t), \qquad \Delta C_t = D(C_t, e_t, L_t), \qquad C_{t+1} = \texttt{Merge}(C_t, \Delta C_t). \tag{1}$$

This process can be iteratively repeated to progressively improve chart correctness. Unlike one-pass regeneration methods, our approach performs localized refinement while preserving correct code regions, reducing unnecessary modifications and mitigating error propagation. Moreover, compared to multi-agent refinement frameworks that require repeated interactions among multiple large models, our method relies on a lightweight diagnostic verifier to provide explicit and structured feedback, resulting in a substantially more efficient test-time refinement process.

4 Experiments

We conduct experiments to verify the effectiveness and efficiency of our proposed visual-code diagnostic verifier and test-time refinement strategy. In addition, we also perform experimental analyses to gain a deeper understanding of plot code diagnostics and repair.

Table 1: Main results of plot-to-code generation across different generators, visual-code diagnostic verifiers, and test-time scaling methods. Vanilla represents one-pass generation.

Models	ChartMimic							Plot2Code						
	E	TT	LO	TP	CL	L	H	E	TT	LO	TP	CL	L	H
Qwen3-VL-8B-Instruct As Generator														
Vanilla	77.8	63.2	73.6	75.7	52.1	66.2	64.4	76.5	41.5	62.8	54.3	37.9	49.2	48.7
Best-of-N	79.0	64.2	74.2	76.3	53.0	66.9	65.2	77.3	42.8	63.8	55.5	39.0	50.3	49.5
Self-debug	79.8	64.8	74.8	76.8	53.8	67.6	65.8	78.0	43.5	64.5	56.2	39.8	51.0	50.2
Multi-agent	83.0	70.0	80.5	74.0	56.8	70.3	67.2	86.4	49.5	76.5	60.8	44.0	57.7	57.8
Qwen3-VL-2B	85.6	69.0	81.5	74.1	55.3	70.0	65.7	92.4	47.9	78.5	59.3	43.3	57.2	58.1
Qwen3-VL-32B	83.6	70.7	81.1	74.5	57.2	70.9	67.5	88.6	50.4	77.3	61.5	44.8	58.5	58.3
Ours-SFT-2B	86.3	71.0	83.6	76.0	58.0	72.1	69.2	**96.2**	50.9	78.6	60.8	43.8	58.5	59.0
Ours-RL-2B	**90.0**	**72.4**	**85.3**	**77.2**	**58.8**	**73.4**	**71.8**	93.2	**51.6**	**79.6**	**63.5**	**45.3**	**60.0**	**60.4**
Qwen3-VL-32B-Instruct As Generator														
Vanilla	75.5	66.7	73.9	76.0	52.5	67.3	65.9	78.6	47.8	69.7	59.3	46.4	55.8	56.3
Best-of-N	77.0	68.0	75.0	76.2	53.0	68.1	67.0	79.5	48.8	70.2	60.0	46.8	56.5	56.8
Self-debug	77.8	68.6	75.6	76.5	53.5	68.6	67.5	79.5	49.3	70.8	60.5	47.1	56.9	57.0
Multi-agent	79.6	72.0	79.2	76.3	54.0	70.4	69.8	80.3	50.2	66.5	61.2	47.2	56.3	57.2
Qwen3-VL-2B	82.5	72.8	81.0	76.9	55.7	71.6	70.6	83.3	52.5	72.4	62.6	49.3	59.2	59.8
Qwen3-VL-32B	80.3	72.7	79.9	76.6	54.3	70.9	70.2	75.0	50.7	66.8	61.8	47.5	56.7	57.5
Ours-SFT-2B	84.8	74.5	83.3	77.8	57.6	73.3	73.6	87.9	55.4	76.0	64.7	52.3	62.1	61.7
Ours-RL-2B	**86.7**	**76.5**	**85.5**	**78.6**	**59.8**	**75.1**	**75.9**	**90.9**	**57.0**	**78.7**	**65.9**	**54.2**	**64.0**	**63.4**
GPT-4o As Generator														
Vanilla	92.4	78.1	92.7	79.9	75.8	81.6	82.8	89.4	57.0	78.3	71.1	57.8	66.1	66.6
Best-of-N	92.8	79.5	93.5	81.0	76.8	82.7	83.5	90.9	58.5	79.5	72.5	58.8	67.3	67.5
Self-debug	93.0	80.2	94.0	81.8	77.5	83.4	84.0	90.9	59.2	80.2	73.2	59.5	68.0	68.2
Multi-agent [27]	–	86.3	95.5	84.2	79.9	86.5	–	–	–	–	–	–	–	–
Qwen3-VL-2B	92.7	82.3	93.8	82.5	77.0	83.9	83.8	91.7	59.3	79.7	72.9	60.1	68.0	68.3
Qwen3-VL-32B	92.9	84.6	94.5	83.2	78.1	85.1	84.7	92.4	60.8	81.0	74.2	61.6	69.4	69.5
Ours-SFT-2B	93.1	87.1	95.0	84.3	79.6	86.5	85.8	92.4	61.9	82.2	75.6	62.7	70.6	71.1
Ours-RL-2B	**93.3**	**88.8**	**95.5**	**85.1**	**81.0**	**87.6**	**86.6**	**93.2**	**62.8**	**83.4**	**76.7**	**64.3**	**71.8**	**73.1**

4.1 Experiment Setting

Datasets. We sample 1,000 examples from Chart2Code-160k [3] to build our visual-code diagnostic verification dataset that consists of 12,000 examples. ChartMimic [1] and Plot2Code [2] are used as test datasets to verify our visual-code diagnostic verifier and test-time plot-to-code refinement.

Evaluation. The code execution success rate (E) is the basic evaluation metric. Following [1], we calculate Text score (TT), Layout score (LO), Type score (TP), Color score (CL) and their average score (L) as fine-grained visual metrics, which are the F1 scores of these elements between the generated and the reference plot. Besides, we employ GPT-4o to evaluate high-level (H) scores [1].

Baselines. To evaluate the generalization of test-time refinement methods, we conduct experiments across multiple generators of different scales and architectures, including Qwen3-VL (8B/32B) and GPT-4o. This setup allows us to assess whether test-time refinement strategies can consistently improve performance across diverse plot-to-code generators.

We compare our method with several representative baselines, including vanilla one-pass generation, multi-agent cooperative generation, and representative test-time refinement methods: Best-of-N sampling and self-debugging. Best-of-N selects the best output from multiple candidates, while self-debug iteratively refines outputs using self-feedback.

For multi-agent methods, we follow the standard setting in [27]. Under the GPT-4o setting, we directly report the original results from [27]. For other generators, we reproduce the multi-agent framework using the same backbone model as the generator, where all agents share the same base model, ensuring a fair and controlled comparison.

We further include our methods, Ours-SFT-2B and Ours-RL-2B, where a compact 2B visual-code verifier is used to guide test-time refinement. To evaluate the effectiveness of learned diagnostic

verification, we also compare with untrained verifiers of different scales (2B and 32B), highlighting the importance of training for accurate visual-code diagnostics.

4.2 Main Results

Table1 reports the main results across different generators and test-time scaling methods.

Overall improvements across generators. Our method consistently improves over vanilla generation across all generators and benchmarks. Both Ours-SFT-2B and Ours-RL-2B achieve notable gains on all evaluation metrics, demonstrating that the test-time refinement approach via structured visual-code diagnostics effectively corrects generation errors.

Comparison with test-time scaling baselines. Compared with representative test-time scaling methods such as Best-of-N and Self-debug, our approach achieves stronger and more consistent improvements. While these methods rely on heuristic sampling or implicit self-feedback, our method provides explicit diagnostic signals, leading to more stable refinement behavior across different generators. The results verify the effectiveness of the proposed simple test-time refinement strategy based on visual-code diagnostics.

Comparison with multi-agent methods. Multi-agent approaches improve performance by coordinating multiple large models (e.g., Qwen3-VL-8B/32B or GPT-4o) for iterative generation and critique. In contrast, our method uses a single compact 2B verifier to guide refinement. Despite this significantly smaller model size, Ours-RL-2B achieves competitive or superior performance across most settings, indicating that learned visual-code diagnostics can effectively replace complex multi-agent reasoning for error correction.

Effect of verifier training. Comparing Ours-SFT-2B and Ours-RL-2B, we observe that reinforcement learning further improves diagnostic quality and downstream generation performance. This highlights the importance of optimizing diagnostic signals beyond supervised learning.

Generalization across model scales. Our method demonstrates consistent gains when applied to generators of different scales, from 8B to 32B models and GPT-4o. This suggests that diagnostic verification captures general error patterns that are complementary to model scaling, and can be effectively integrated with diverse base generators.

Inference Token Cost. Table 2 compares the inference token consumption of different refinement strategies. Compared with Best-of-N sampling and multi-agent refinement based on GPT-4o, our method uses a lightweight Qwen3-VL-2B diagnostic verifier to guide localized refinement.

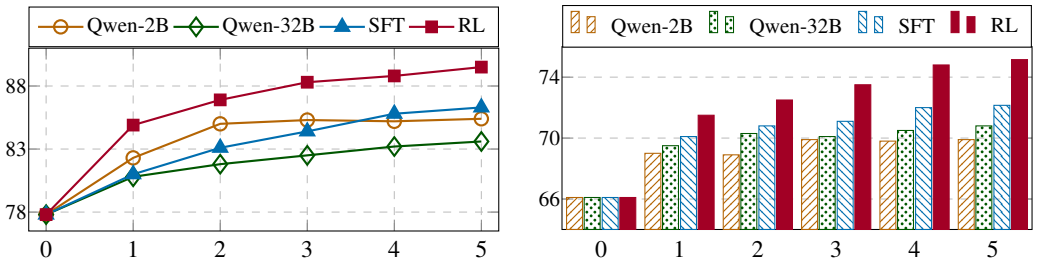
Table 2: Comparison of inference token consumption.

Method	Model	L	Tokens	Times
Best-of-N	GPT-4o	82.7	3047.6	3.7x
Multi-agent	GPT-4o	86.5	11264.0	1.0x
Ours	Qwen3-VL-2B	**87.6**	**1859.3**	**6.1x**

Despite using substantially fewer tokens, our method achieves the best visual score (L: 87.6). Compared with multi-agent refinement, our approach reduces inference tokens from 11,264.0 to 1,859.3, corresponding to a $6.1\times$ reduction in inference cost. These results show that structured visual-code diagnostics enable efficient and effective test-time refinement.

4.3 Analyses

Performances over Refinement Iterations. We analyze the effectiveness of visual-code diagnostics across refinement iterations on ChartMimic. Figure 3 compares untrained models (Qwen-2B, Qwen-32B) and our trained verifiers (SFT and RL). Curves denote execution success rate (E), while bars report the visual metric (L). Starting from one-pass generation (iteration 0), all methods improve with iterative refinement, confirming that test-time refinement can progressively correct both execution and visual errors. However, the quality of diagnostic signals plays a critical role. While larger untrained models such as Qwen-32B provide moderate gains, both trained 2B-scale verifiers consistently



(a) E score over refinement iterations.

(b) L score over refinement iterations.

Figure 3: Plot-to-code generation results of refinement iterations across different visual-code diagnostic verifiers.

achieve stronger refinement performance, showing that learned visual-code diagnostics are more effective than model scale alone.

We further observe different refinement behaviors across verifiers. Qwen-2B achieves rapid early gains in execution success rate but provides limited improvement in visual quality, suggesting that it mainly resolves simple execution issues while struggling with more complex visual discrepancies. In contrast, trained verifiers produce more informative diagnostic feedback, enabling progressively better visual refinement over iterations. Among all methods, the RL-trained verifier achieves the best execution and visual performance, indicating that reinforcement learning further improves diagnostic quality and refinement effectiveness.

Table 3: Main results of plot-to-code generation based on ChartMaster-7B [4] as Generator.

Models	**ChartMimic**							**Plot2Code**						
	E	**TT**	**LO**	**TP**	**CL**	**L**	**H**	**E**	**TT**	**LO**	**TP**	**CL**	**L**	**H**
Vanilla	93.7	**80.0**	91.6	79.1	**67.0**	79.4	**85.1**	90.1	**57.2**	**82.4**	**62.9**	51.0	**63.4**	**64.5**
Best-of-N	**94.2**	79.4	**92.2**	79.8	66.5	**79.5**	84.7	**91.7**	56.8	81.5	63.5	50.8	63.2	64.0
Self-debug	91.0	77.0	89.5	77.5	64.0	77.0	82.5	86.4	55.0	79.0	61.5	49.5	61.3	62.0
Multi-agent	88.5	74.5	86.5	75.5	62.5	74.8	80.5	84.8	53.0	76.5	60.0	48.5	59.5	60.5
Qwen3-VL-2B	78.3	63.3	74.3	**80.0**	58.9	69.1	71.2	83.3	48.3	71.6	61.9	**52.9**	58.7	60.3
Qwen3-VL-32B	67.0	59.9	70.9	**80.0**	56.7	66.9	69.9	66.6	43.7	70.2	62.5	52.5	57.2	58.4
Ours-SFT-2B	73.3	58.0	69.6	76.0	54.1	64.4	67.6	82.6	50.2	73.2	60.0	52.0	58.9	59.9
Ours-RL-2B	71.9	56.6	67.9	76.2	52.9	63.4	65.5	87.1	51.2	70.0	60.7	53.5	58.9	59.3

Analysis on Specialized Plot-to-Code Generators. We further evaluate ChartMaster-7B [4], a model specialized for direct plot-to-code generation. As shown in Table 3, strong one-pass generation does not necessarily translate into effective refinement. On ChartMimic, Ours-RL-2B reduces execution success from 93.7 to 71.9 and visual similarity (L) from 79.4 to 63.4; performance also declines on Plot2Code. This failure extends across different feedback strategies: self-debug, multi-agent refinement, and untrained verifiers also degrade performance, while Best-of-N provides almost no visual improvement.

These results suggest that the generator's ability to revise code is a key bottleneck. ChartMaster-7B is trained for direct generation but is not explicitly trained for instruction-following code revision, whereas general-purpose MLLMs can use their broader instruction-following capabilities to act on diagnostic feedback. To further investigate this explanation, we conduct a manual case analysis in which ChartMaster-7B receives human-annotated gold visual-code diagnostics specifying the correct error types and erroneous code regions. Even with this accurate feedback, the model still fails to correct the identified errors in the inspected cases. This qualitative evidence indicates that inaccurate diagnostics alone cannot explain the failures and supports insufficient code-revision ability as an important contributing factor.

ChartMaster-7B therefore illustrates an applicability boundary: effective test-time refinement requires a generator that can reliably translate diagnostic feedback into correct code edits. This behavior suggests a trade-off in ChartMaster-7B's domain-specific fine-tuning: gains in one-pass plot-to-code generation may come at the cost of general instruction-following and code-editing capabilities.

Improving verifier quality alone may be insufficient when this capability is lacking, even for a generator with strong one-pass performance. Training generators to follow diagnostic instructions and perform reliable code revision is an important direction for future work.

Ablation Study on Visual-Code Diagnostic Signals. We conduct ablation experiments on the ChartMimic dataset using Qwen3-VL-8B as the generator to analyze the contribution of visual and code diagnostics. As shown in Table 4, both visual-only and code-only settings improve over vanilla generation, indicating that each type of signal provides useful guidance for refinement.

Table 4: Ablation experiments for visual-code diagnostics.

Visual	Code	E	TT	LO	TP	CL	L	H
✓	✗	84.2	69.9	80.1	75.7	56.9	70.7	66.6
✗	✓	86.5	71.3	83.4	76.0	58.0	72.2	70.4
✓	✓	**90.0**	**72.4**	**85.3**	**77.2**	**58.8**	**73.4**	**71.8**

Comparing the two, code localization leads to stronger gains than visual-only signals, suggesting that precise, code-grounded feedback is particularly important for correcting errors. However, the best performance is achieved when combining both signals. This demonstrates that visual and code diagnostics are complementary: visual signals identify what is wrong, while code localization specifies where to fix. Their joint modeling enables more accurate and effective refinement, supporting our formulation of structured visual-code diagnostics.

Ablation Study on Reward Function. We examine different reward functions by training verifiers from the same SFT checkpoint using exact match, set F1, or Jaccard similarity for both the visual and code rewards, while retaining their uniform weighting. Table 5 reports downstream execution success (E) and fine-grained visual similarity (L) on ChartMimic using Qwen3-VL-8B as the generator.

Table 5: Ablation experiments for reward functions.

Reward function	E	L
Exact match	87.2	71.9
F1	89.8	**73.4**
Jaccard	**90.0**	**73.4**

Jaccard similarity provides partial credit for overlap between variable-size sets of error types or localization actions, while normalizing for set size and penalizing both missing and spurious predictions. In contrast, exact match assigns zero reward to a component whenever its predicted set differs from the reference, even if some predictions are correct. Set F1 is a monotonic transformation of Jaccard similarity for each prediction/reference pair and captures similar overlap information.

Jaccard and F1 achieve the same L score of 73.4 and nearly identical execution success rates (90.0 and 89.8, respectively), suggesting limited sensitivity to the choice between these two overlap metrics in this setting. Exact match yields lower scores (E: 87.2; L: 71.9), consistent with its sparser reward discarding useful partial matches and providing less informative feedback during training.

Diagnostic Verification Performance. In addition to evaluating end-to-end plot-to-code generation, we directly assess the quality of visual-code diagnostic verification on the error discrimination dataset. We use 6,000 and 5,000 samples for SFT and RL training, respectively, and reserve 1,000 samples for evaluation. Table 6 reports the results across different verifiers. We evaluate both visual error prediction (PEF) and code localization quality (CEP, CER, CEF), where code errors are defined based on the structured tuples (i, a, j) described in Section 3.1.3.

Table 6: Visual-code diagnostic verification results across different verifiers.

Model	PEF	CEP	CER	CEF
Qwen3-VL-2B	70.5	50.2	20.1	28.7
Qwen3-VL-32B	74.7	59.4	40.9	48.4
Ours-SFT-2B	80.8	65.8	62.1	63.9
Ours-RL-2B	**87.2**	**73.2**	**80.7**	**76.8**

Our trained verifiers significantly outperform untrained models of both smaller (Qwen3-2B) and larger (Qwen3-32B) scales. Notably, Ours-RL-2B achieves the best performance across all metrics, demonstrating that a compact model can effectively learn structured visual-code diagnostics when trained with our dataset and objectives. We further observe that untrained models exhibit imbalanced precision and recall in code localization, indicating unreliable and incomplete diagnostic signals. In contrast, our trained models achieve more balanced performance, suggesting more accurate and comprehensive identification of error regions. These results validate the effectiveness of our

diagnostic dataset and training strategy, and explain the strong gains observed in test-time refinement: improved diagnostic signals directly translate into more precise and effective code updates.

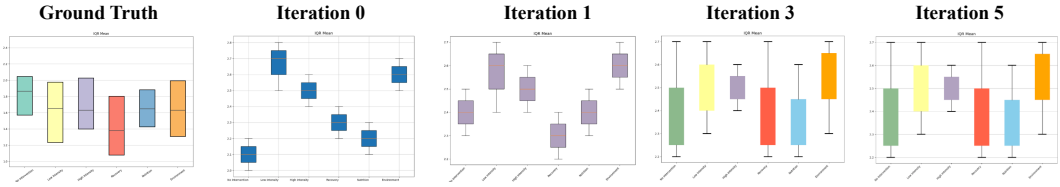


Figure 4: A case of our test-time refinement for plot-to-code generation.

Case Study. Figure 4 illustrates a representative example of our visual-code diagnostics-guided test-time refinement. Iteration 0 corresponds to direct generation, which exhibits noticeable discrepancies compared to the reference chart. With iterative refinement, the generated plot progressively aligns with the reference. Early iterations correct prominent errors such as axis configuration and styling attributes, while later iterations refine more detailed visual elements. This demonstrates that the verifier provides structured diagnostic signals that enable the generator to perform targeted updates rather than full regeneration.

We observe that some corrections may not succeed in early iterations but are resolved in subsequent steps, indicating that refinement benefits from accumulating diagnostic feedback over iterations. Meanwhile, later iterations tend to introduce fewer changes, suggesting convergence as major errors are resolved. Despite these improvements, certain fine-grained discrepancies remain (e.g., subtle styling differences), even after multiple iterations. This highlights the remaining challenges in precisely aligning low-level visual details and suggests that further improving diagnostic granularity or generator editing capability could yield additional gains.

5 Conclusion

We study plot-to-code generation from the perspective of test-time refinement and identify diagnostic verification as a key component for improving generation quality. We propose a visual-code diagnostic framework that produces structured signals to detect visual discrepancies and localize code errors. By constructing a scalable diagnostic dataset and training a compact verifier, our method enables effective and targeted code refinement. Experiments show that a 2B verifier consistently improves generation performance, outperforming larger untrained models and achieving competitive results compared to multi-agent approaches with substantially lower cost. We further analyze limitations on specialized plot-to-code generators, highlighting the importance of revision capability for effective refinement. Our results suggest that learned diagnostic verification is a promising direction for efficient multimodal code generation.

Acknowledgments and Disclosure of Funding

This work was supported by the National Natural Science Foundation of China under Grant 62336008, the Guangdong Provincial Key Laboratory of Artificial General Intelligence and Large Models under Grant 2026B1212010005, and the CCF-Huawei Populus Grove Fund under Grant CCF-HuaweiSE2025021.

References

[1] Cheng Yang, Chufan Shi, Yaxin Liu, Bo Shui, Junjie Wang, Mohan Jing, Linran XU, Xinyu Zhu, Siheng Li, Yuxiang Zhang, Gongye Liu, Xiaomei Nie, Deng Cai, and Yujiu Yang. Chartmimic: Evaluating LMM's cross-modal reasoning capability via chart-to-code generation. In *The Thirteenth International Conference on Learning Representations*, 2025. URL `https://openreview.net/forum?id=sGpCzsfd1K`.

[2] Chengyue Wu, Zhixuan Liang, Yixiao Ge, Qiushan Guo, Zeyu Lu, Jiahao Wang, Ying Shan, and Ping Luo. Plot2Code: A comprehensive benchmark for evaluating multi-modal large

language models in code generation from scientific plots. In *Findings of the Association for Computational Linguistics: NAACL 2025*, pages 3006–3028, 2025. ISBN 979-8-89176-195-7. doi: 10.18653/v1/2025.findings-naacl.164. URL https://aclanthology.org/2025.findings-naacl.164/.

[3] Xuanle Zhao, Xianzhen Luo, Qi Shi, Chi Chen, Shuo Wang, Zhiyuan Liu, and Maosong Sun. ChartCoder: Advancing multimodal large language model for chart-to-code generation. In *Proceedings of the 63rd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pages 7333–7348, 2025. ISBN 979-8-89176-251-0. doi: 10.18653/v1/2025.acl-long.363. URL https://aclanthology.org/2025.acl-long.363/.

[4] Wentao Tan, Qiong Cao, Chao Xue, Yibing Zhan, Changxing Ding, and Xiaodong He. Chartmaster: Advancing chart-to-code generation with real-world charts and chart similarity reinforcement learning. *arXiv preprint arXiv:2508.17608*, 2025. URL https://arxiv.org/abs/2508.17608.

[5] Tianhao Niu, Yiming Cui, Baoxin Wang, Xiao Xu, Xin Yao, Qingfu Zhu, Dayong Wu, Shijin Wang, and Wanxiang Che. Chart2Code53: A large-scale diverse and complex dataset for enhancing chart-to-code generation. In *Proceedings of the 2025 Conference on Empirical Methods in Natural Language Processing*, pages 15839–15855, 2025. doi: 10.18653/v1/2025.emnlp-main.799. URL https://aclanthology.org/2025.emnlp-main.799/.

[6] Zhengzhuo Xu, Bowen Qu, Yiyan Qi, SiNan Du, Chengjin Xu, Chun Yuan, and Jian Guo. Chartmoe: Mixture of diversely aligned expert connector for chart understanding. In *The Thirteenth International Conference on Learning Representations*, 2025. URL https://openreview.net/forum?id=o5TsWTUSeF.

[7] Yuwei Yang, Zeyu Zhang, Yunzhong Hou, Zhuowan Li, Gaowen Liu, Ali Payani, Yuan-Sen Ting, and Liang Zheng. Effective training data synthesis for improving mllm chart understanding. In *Proceedings of the IEEE/CVF International Conference on Computer Vision (ICCV)*, pages 2653–2663, October 2025. URL https://openaccess.thecvf.com/content/ICCV2025/html/Yang_Effective_Training_Data_Synthesis_for_Improving_MLLM_Chart_Understanding_ICCV_2025_paper.html.

[8] James Ford, Xingmeng Zhao, Dan Schumacher, and Anthony Rios. Charting the future: Using chart question-answering for scalable evaluation of LLM-driven data visualizations. In *Proceedings of the 31st International Conference on Computational Linguistics*, pages 7497–7510, 2025. URL https://aclanthology.org/2025.coling-main.501/.

[9] Chengqi Duan, Kaiyue Sun, Rongyao Fang, Manyuan Zhang, Yan Feng, Ying Luo, Yufang Liu, Ke Wang, Peng Pei, Xunliang Cai, et al. Codeplot-cot: Mathematical visual reasoning by thinking with code-driven images. *arXiv preprint arXiv:2510.11718*, 2025. URL https://arxiv.org/abs/2510.11718.

[10] Ahmed Masry, Megh Thakkar, Aayush Bajaj, Aaryaman Kartha, Enamul Hoque, and Shafiq Joty. ChartGemma: Visual instruction-tuning for chart reasoning in the wild. In *Proceedings of the 31st International Conference on Computational Linguistics: Industry Track*, pages 625–643, 2025. URL https://aclanthology.org/2025.coling-industry.54/.

[11] Victor Carbune, Hassan Mansoor, Fangyu Liu, Rahul Aralikatte, Gilles Baechler, Jindong Chen, and Abhanshu Sharma. Chart-based reasoning: Transferring capabilities from LLMs to VLMs. In *Findings of the Association for Computational Linguistics: NAACL 2024*, pages 989–1004, Mexico City, Mexico, 2024. doi: 10.18653/v1/2024.findings-naacl.62. URL https://aclanthology.org/2024.findings-naacl.62/.

[12] Qwen Team. Qwen3-vl technical report. *arXiv preprint arXiv:2511.21631*, 2025. URL https://arxiv.org/pdf/2511.21631.

[13] Aaron Hurst, Adam Lerer, Adam P Goucher, Adam Perelman, Aditya Ramesh, Aidan Clark, AJ Ostrow, Akila Welihinda, Alan Hayes, Alec Radford, et al. Gpt-4o system card. *arXiv preprint arXiv:2410.21276*, 2024. URL http://arxiv.org/abs/2410.21276.

[14] Yuan Yao, Tianyu Yu, Ao Zhang, Chongyi Wang, Junbo Cui, Hongji Zhu, Tianchi Cai, Chi Chen, Haoyu Li, Weilin Zhao, et al. Efficient gpt-4v level multimodal large language model for deployment on edge devices. *Nature Communications*, 16(1):5509, 2025. URL https://www.nature.com/articles/s41467-025-61040-5.

[15] Weiyun Wang, Zhangwei Gao, Lixin Gu, Hengjun Pu, Long Cui, Xingguang Wei, Zhaoyang Liu, Linglin Jing, Shenglong Ye, Jie Shao, et al. Internvl3. 5: Advancing open-source multimodal models in versatility, reasoning, and efficiency. *arXiv preprint arXiv:2508.18265*, 2025. URL https://arxiv.org/abs/2508.18265.

[16] Wenyi Hong, Wenmeng Yu, Xiaotao Gu, Guo Wang, Guobing Gan, Haomiao Tang, Jiale Cheng, Ji Qi, Junhui Ji, Lihang Pan, et al. Glm-4.1 v-thinking: Towards versatile multimodal reasoning with scalable reinforcement learning. *arXiv preprint arXiv:2507.01006*, 2025. URL https://arxiv.org/abs/2507.01006.

[17] Dong Guo, Faming Wu, Feida Zhu, Fuxing Leng, Guang Shi, Haobin Chen, Haoqi Fan, Jian Wang, Jianyu Jiang, Jiawei Wang, et al. Seed1. 5-vl technical report. *arXiv preprint arXiv:2505.07062*, 2025. URL https://arxiv.org/abs/2505.07062.

[18] Zhiyu Wu, Xiaokang Chen, Zizheng Pan, Xingchao Liu, Wen Liu, Damai Dai, Huazuo Gao, Yiyang Ma, Chengyue Wu, Bingxuan Wang, et al. Deepseek-vl2: Mixture-of-experts vision-language models for advanced multimodal understanding. *arXiv preprint arXiv:2412.10302*, 2024. URL https://arxiv.org/abs/2412.10302.

[19] Zhihan Zhang, Yixin Cao, and Lizi Liao. Boosting chart-to-code generation in mllm via dual preference-guided refinement. In *Proceedings of the 33rd ACM International Conference on Multimedia*, MM '25, 2025. URL https://arxiv.org/pdf/2504.02906.

[20] Lei Chen, Xuanle Zhao, Zhixiong Zeng, Jing Huang, Liming Zheng, Yufeng Zhong, and Lin Ma. Breaking the SFT plateau: Multimodal structured reinforcement learning for chart-to-code generation. In *The Fourteenth International Conference on Learning Representations*, 2026. URL https://openreview.net/forum?id=14qYxv2jki.

[21] Kimi Team, Angang Du, Bofei Gao, Bowei Xing, Changjiu Jiang, Cheng Chen, Cheng Li, Chenjun Xiao, Chenzhuang Du, Chonghua Liao, et al. Kimi k1. 5: Scaling reinforcement learning with llms. *arXiv preprint arXiv:2501.12599*, 2025. URL https://arxiv.org/abs/2501.12599.

[22] Jingyi Zhang, Jiaxing Huang, Sheng Jin, and Shijian Lu. Vision-language models for vision tasks: A survey. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 46(8): 5625–5644, 2024. doi: 10.1109/TPAMI.2024.3369699.

[23] Jiaxing Huang, Jingyi Zhang, Kai Jiang, Han Qiu, Xiaoqin Zhang, Ling Shao, Shijian Lu, and Dacheng Tao. Visual instruction tuning towards general-purpose multimodal large language model: A survey. *International Journal of Computer Vision*, 133(11):8151–8189, 2025. URL https://link.springer.com/article/10.1007/s11263-025-02572-7.

[24] Siyu Cao, Hangting Chen, Peng Chen, Yiji Cheng, Yutao Cui, Xinchi Deng, Ying Dong, Kipper Gong, Tianpeng Gu, Xiusen Gu, et al. Hunyuanimage 3.0 technical report. *arXiv preprint arXiv:2509.23951*, 2025. URL https://arxiv.org/abs/2509.23951.

[25] Peiming Guo, Sinuo Liu, Yanzhao Zhang, Dingkun Long, Pengju Xie, Meishan Zhang, and Min Zhang. An end-to-end model for photo-sharing multi-modal dialogue generation. In *2025 IEEE International Conference on Multimedia and Expo (ICME)*, pages 1–7, 2025. doi: 10.1109/ICME59968.2025.11208999. URL https://ieeexplore.ieee.org/abstract/document/11208999.

[26] Bradley Brown, Jordan Juravsky, Ryan Ehrlich, Ronald Clark, Quoc V Le, Christopher Ré, and Azalia Mirhoseini. Large language monkeys: Scaling inference compute with repeated sampling. *arXiv preprint arXiv:2407.21787*, 2024. URL https://arxiv.org/abs/2407.21787.

[27] Bingxuan Li, Yiwei Wang, Jiuxiang Gu, Kai-Wei Chang, and Nanyun Peng. METAL: A multi-agent framework for chart generation with test-time scaling. In *Proceedings of the 63rd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pages 30054–30069, 2025. ISBN 979-8-89176-251-0. doi: 10.18653/v1/2025.acl-long.1452. URL https://aclanthology.org/2025.acl-long.1452/.

[28] VI Levenshtein. Binary codes capable of correcting deletions, insertions, and reversals. In *Soviet physics doklady*, volume 10, pages 707–710, 1966. URL https://nymity.ch/sybilhunting/pdf/Levenshtein1966a.pdf.

[29] Robert A. Wagner and Michael J. Fischer. The string-to-string correction problem. *J. ACM*, 21 (1):168–173, 1974. ISSN 0004-5411. doi: 10.1145/321796.321811. URL https://doi.org/10.1145/321796.321811.

[30] Xuanle Zhao, Xuexin Liu, Yang Haoyue, Xianzhen Luo, Fanhu Zeng, Jianling Li, Qi Shi, and Chi Chen. ChartEdit: How far are MLLMs from automating chart analysis? evaluating MLLMs' capability via chart editing. In *Findings of the Association for Computational Linguistics: ACL 2025*, pages 3616–3630, 2025. ISBN 979-8-89176-256-5. doi: 10.18653/v1/2025.findings-acl.185. URL https://aclanthology.org/2025.findings-acl.185/.

[31] Renqiu Xia, Hancheng Ye, Xiangchao Yan, Qi Liu, Hongbin Zhou, Zijun Chen, Botian Shi, Junchi Yan, and Bo Zhang. Chartx and chartvlm: A versatile benchmark and foundation model for complicated chart reasoning. *IEEE Transactions on Image Processing*, 34:7436–7447, 2025. doi: 10.1109/TIP.2025.3607618. URL https://ieeexplore.ieee.org/abstract/document/11202357.

[32] Lingjie Jiang, Shaohan Huang, Xun Wu, Yixia Li, Guanhua Chen, Dongdong Zhang, and Furu Wei. Viscodex: Unified multimodal code generation via merging vision and coding models. In *The Fourteenth International Conference on Learning Representations*, 2026. URL https://openreview.net/forum?id=hUXzPauNEM.

[33] Qiushi Sun, Jingyang Gong, Yang Liu, Qiaosheng Chen, Lei Li, Kai Chen, Qipeng Guo, Ben Kao, and Fei Yuan. Januscoder: Towards a foundational visual-programmatic interface for code intelligence. In *The Fourteenth International Conference on Learning Representations*, 2026. URL https://openreview.net/forum?id=N4BB09TXad.

[34] Charlie Snell, Jaehoon Lee, Kelvin Xu, and Aviral Kumar. Scaling llm test-time compute optimally can be more effective than scaling model parameters. *arXiv preprint arXiv:2408.03314*, 2024. URL https://arxiv.org/abs/2408.03314.

[35] Hunter Lightman, Vineet Kosaraju, Yuri Burda, Harrison Edwards, Bowen Baker, Teddy Lee, Jan Leike, John Schulman, Ilya Sutskever, and Karl Cobbe. Let's verify step by step. In *The Twelfth International Conference on Learning Representations*, 2024. URL https://openreview.net/forum?id=v8L0pN6EOi.

[36] J. D. Hunter. Matplotlib: A 2d graphics environment. *Computing in Science & Engineering*, 9 (3):90–95, 2007. doi: 10.1109/MCSE.2007.55.

[37] Charles R Harris, K Jarrod Millman, Stéfan J Van Der Walt, Ralf Gommers, Pauli Virtanen, David Cournapeau, Eric Wieser, Julian Taylor, Sebastian Berg, Nathaniel J Smith, et al. Array programming with numpy. *nature*, 585(7825):357–362, 2020. URL https://www.nature.com/articles/s41586-020-2649-2.

[38] Daya Guo, Dejian Yang, Haowei Zhang, Junxiao Song, Peiyi Wang, Qihao Zhu, Runxin Xu, Ruoyu Zhang, Shirong Ma, Xiao Bi, et al. Deepseek-r1 incentivizes reasoning in llms through reinforcement learning. *Nature*, 645(8081):633–638, 2025. URL https://www.nature.com/articles/s41586-025-09422-z.

[39] Paul Jaccard. Étude comparative de la distribution florale dans une portion des alpes et des jura. *Bull Soc Vaudoise Sci Nat*, 37:547–579, 1901. URL https://cir.nii.ac.jp/crid/1573387450552842240.

[40] Yaowei Zheng, Richong Zhang, Junhao Zhang, Yanhan Ye, and Zheyan Luo. LlamaFactory: Unified efficient fine-tuning of 100+ language models. In *Proceedings of the 62nd Annual Meeting of the Association for Computational Linguistics (Volume 3: System Demonstrations)*, pages 400–410, 2024. doi: 10.18653/v1/2024.acl-demos.38. URL https://aclanthology.org/2024.acl-demos.38/.

[41] Guangming Sheng, Chi Zhang, Zilingfeng Ye, Xibin Wu, Wang Zhang, Ru Zhang, Yanghua Peng, Haibin Lin, and Chuan Wu. Hybridflow: A flexible and efficient rlhf framework. In *Proceedings of the Twentieth European Conference on Computer Systems*, EuroSys '25, page 1279–1297, 2025. ISBN 9798400711961. doi: 10.1145/3689031.3696075. URL https://doi.org/10.1145/3689031.3696075.

A Plot Error Taxonomy and Distribution

A.1 Plot Error Types

Examining all results and considering the programming logic and coding style of the Matplotlib [36, 37] code, we categorize generation errors into four coarse-grained types:

- Compiler error, where the generated code fails to execute and render the plot.
- Data error, where the number and values of shapes and data points are incorrect.
- Local style error, where styles of specific graphic shapes and elements are wrong, such as incorrect colors, markers, or line styles.
- Global style error, where styles of global elements for the plot are misaligned, like axes, grid, titles, or extra texts.

Figure 5 demonstrates examples of four error types. The compiler error does not render the reference plot image, while the data error contains an extra box and incorrect lower and upper quartiles. There are wrong shapes and colors for the lines, dots, and boxes in the local style error, while extra x tick label and title in the global style error. In particular, the categorization basis of the latter three error types is primarily derived from the Matplotlib [36] code. A standard Matplotlib program consists of three components. The first component is data loading, the second involves invoking the core plotting APIs, where the local properties of plot elements can be configured, and the third sets global elements such as the title, axes, and additional annotations. Accordingly, these three categories of plot errors directly correspond to errors in the code, enabling efficient localization of code errors from plot errors. This correspondence reduces the burden on the subsequent Plot&Code discriminator, allowing even relatively small-scale MLLMs to effectively handle this task and thereby improve the iterative refinement performance of the plot-to-code generator.

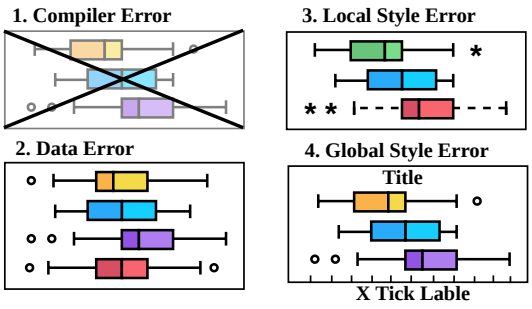


Figure 5: Error types for plot-to-code generation.

A.2 Plot Error Distribution

We sample 100 instances from the ChartMimic dataset and use four representative plot-to-code models to directly generate code outputs. The three authors of this paper first independently inspect the code and rendered plots produced by the large language models, and then jointly discuss and establish the error taxonomy by synthesizing their individual judgments. Subsequently, all samples are annotated by the three authors. For cases with annotation disagreements, the final labels are determined through group discussion. Figure 6 presents the distribution of error types across different models: Qwen3-VL-8B, Qwen3-VL-32B, ChartMaster and GPT-4o.

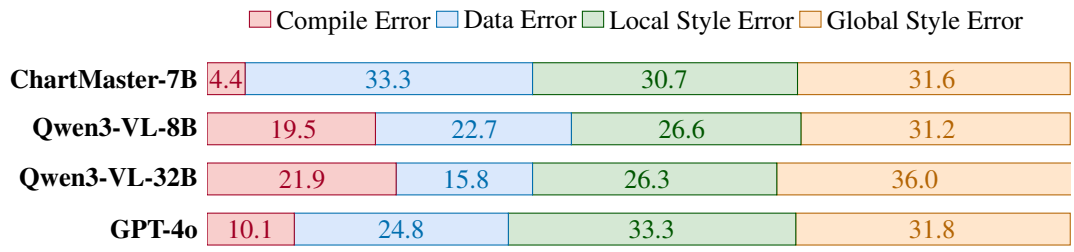


Figure 6: Distribution of plot error types.

B Dataset Construction

B.1 Process

B.1.1 Error-Aware Code Mutation

Given a correct chart-code pair (I, C), we construct synthetic erroneous samples by applying controlled code mutations. We focus on three types of visual errors: data errors, local style errors, and global style errors. Compiler errors are excluded from mutation, as they can be directly identified from execution feedback. For each seed program, we employ GPT-4o to sequentially generate three mutated variants, each introducing one type of error while preserving other parts of the code. The prompts of GPT-4o for code mutation are listed in Appendix B.3. The order of mutation types is randomly permuted across samples to improve diversity and avoid positional bias.

B.1.2 Pairwise Sample Construction

Given one seed program and its three mutated variants, we construct pairwise diagnostic samples by selecting two programs (C_1, C_2). The corresponding rendered plots (I_1, I_2) are obtained via execution. We treat C_1 as the generated code and C_2 as the reference. The visual error label is determined by the mutation type that differentiates C_1 from C_2. This pairwise construction produces multiple combinations per seed example, resulting in up to 12 samples per original chart-code pair.

B.1.3 Code Region Alignment

To obtain code localization supervision, we align C_1 and C_2 using a line-level variant of the Levenshtein distance algorithm (in Appendix B.2). The alignment identifies inserted, deleted, and modified code segments between the two programs. We further refine the alignment by applying code-aware constraints to ensure valid line matching and reduce noise. The resulting regions are treated as candidate error locations.

B.1.4 Diagnostic Annotation Format

Instead of predicting explicit line indices, we convert localization signals into a generation-friendly format by inserting `TODO` comments near the identified code regions in C_1. This produces structured diagnostic annotations that preserve code context and are more robust for language models. Each final training instance consists of: 1. Generated plot I_1, 2. Reference plot I_2, 3. Generated code C_1 with diagnostic annotations, 4. Visual error label. This construction provides scalable supervision for both error classification and fine-grained code localization without requiring manual annotation of code regions.

B.2 Postprocessing Algorithm

We adopt line-level Levenshtein distance algorithm to compare two codes and extract the inserted/deleted/modified code line numbers, which is detailed in algorithm 1. In concrete terms, the condition control statement in lines 20-23 for replace_distance is different from the conventional Levenshtein distance algorithm.

B.3 Prompts

Dataset Construction Prompt for Data Issue

You are an expert Python developer who specializes in writing matplotlib code. Please modify the given code snippet according to the requirements.
Requirements:
1. Change the data used in the code snippet to generate a different image.
2. The changed data should have enough variation to ensure the noticeable difference in the generated image.
3. Add or reduce the number of graphic shapes, and change the values of the data.
4. Keep the color and the style of graphic shapes of the image unchanged.
5. Ensure that the overall structure and style of the code remain unchanged.
Response Format:
Return the modified complete code surrounded by ```python and ```.

Dataset Construction Prompt for Local Style Issue

You are an expert Python developer who specializes in writing matplotlib code. Please modify the given code snippet according to the requirements.
Requirements:
1. Change the style of graphic shapes and elements in the chart to generate a different image.
2. Change, add or reduce various parameters in the graphic shape rendering function.
3. The change should have enough variation to ensure the noticeable difference in the generated image.
4. Only when the original code uses the pie function, add or change or reduce parameters wedgeprops or explode with one-half probability.
5. Keep the data and the style of axes of the image unchanged.
6. Ensure that the overall structure and style of the code remain unchanged.
Response Format:
Return the modified complete code surrounded by ```python and ```.

Dataset Construction Prompt for Global Style Issue

You are an expert Python developer who specializes in writing matplotlib code. Please modify the given code snippet according to the requirements.
Requirements:
1. Change the global styles used in the code snippet to generate a different image.
2. Gloabl styles include: Axis ranges, Axis scales, Axis labels, Axis ticks, Title, Annotations, Grid, and Extra text.
3. The change should have enough variation to ensure the noticeable difference in the generated image.
4. Keep the data and the color and other styles of graphic shapes of the image unchanged
5. Ensure that the overall structure and style of the code remain unchanged.
Response Format:
Return the modified complete code surrounded by ```python and ```.

Figure 7: Dataset Construction Prompts.

We mutate plot code by GPT-4o based on the proposed taxonomy of plot error during dataset construction. Prompts for code mutation are listed in Figure 7.

Prompt of visual-code diagnostic verifier

I write a code snippet to reproduce a science chart. I will give you the reference chart and the generated chart by my code. Please help me to modify my code to make the reference chart closer to the original chart. Output difference types and code snippet with TODO comments.
Difference types include:
1. data: number and values of shapes and data points
2. local_style: styles of specific graphic shapes and elements
3. global_style: styles of global elements like Axis, Title, Annotations, Grid, Extra text
TODO comments should be inserted into my code snippet and conform the following format:
1. # TODO: Add extra code in the next line
2. # TODO: Modify the following <number> lines
3. # TODO: Delete the following <number> lines
Return difference types surrounded < and > in one line.
Return the modified complete code surrounded by ```python and ```.
Reference Chart:
{reference image}
Generated Chart:
{generated image}
Code Snippet:
```python
{generated code}
```

Figure 8: Prompt of visual-code diagnostic verifier.

C Test-Time Refinement for Plot-to-Code Generation via Visual-Code Diagnostics

During iterative refinement, we use prompts in Figure 8 to instruct the visual-code diagnostic verifier and Figure 9 for the plot-to-code generator, respectively.

D Experiments

D.1 Hyperparameters

We train the visual-code diagnostic verifier based on Qwen3-VL-2B-Instruct [12]. For the SFT stage, we conduct training on the 6,000 samples for 2 epochs. Our implementation is based on LLaMA-Factory [40] with batch size 64, learning rate 1e-5, cutoff length 16,000. For the reinforcement learning stage, we continually train the SFT model on the other 5,000 samples for 3 epochs. Our training is based on VERL [41] with global batch size 32, ppo_mini_batchsize 16, learning rate 1e-6, rollout_n 8, sampling temperature 1.0, top_p 1.0. For the inference stage, we repeat iterative refinement $N = 5$ times. We set the temperature to 0.7, top_p 0.8 for our verifiers, while default sampling parameters for all plot-to-code generators. All experiments are conducted on 8 Nvidia A6000 GPUs.

D.2 Case Study

Figure 10 illustrates a case of test-time refinement for plot-to-code generation with different visual-code diagnostic verifiers.

Prompt of test-time refinement for plot-to-code generator

You are a helpful assistant who can revise Python matplotlib code to reproduce the reference chart image.
I will provide:
1. A Python code snippet that generates a chart, but the output does not match the reference chart image. The code contains # TODO comments indicating the specific locations that require modification.
2. A list of difference types describing how the generated chart differs from the reference image.
Your task is:
1. Carefully analyze and infer the visual differences between the reference chart image and my code snippet.
2. Identify specific code segments that need to be modified.
3. Return only the code segments that need to be modified, not the complete code.
IMPORTANT: Return the modifications in the following format:
—OLD—
[original code segment 1]
—NEW—
[modified code segment 1]
—OLD—
[original code segment 2]
—NEW—
[modified code segment 2]
You can have multiple OLD/NEW pairs if multiple segments need to be modified.
Code snippet:
```python
{current code}
```
Difference types:
{Plot error types}
Reference image:
{reference image}
Generated image:
{generated image}

Figure 9: Prompts of test-time refinement for plot-to-code generation via visual-code diagnostics.

Algorithm 1: Adapted Line-level Levenshtein distance algorithm

Input: Two code strings $code1$, $code2$
Output: $inserted$, $deleted$, $replaced$ pairs of code line numbers in $code1$ and $code2$

$lines1 \leftarrow$ `SplitLines`$(code1)$;
$lines2 \leftarrow$ `SplitLines`$(code2)$;
$n \leftarrow |lines1|,\ m \leftarrow |lines2|$;
Initialize DP table;
$dp \leftarrow$ `Matrix`$(n+1, m+1, 0)$;
for $i \leftarrow 0$ **to** n **do**
    $dp[i][0] \leftarrow i$; `// delete` i `lines`
end
for $j \leftarrow 0$ **to** m **do**
    $dp[0][j] \leftarrow j$; `// insert` j `lines`
end
Fill DP table;
for $i \leftarrow 1$ **to** n **do**
    for $j \leftarrow 1$ **to** m **do**
        if $lines1[i-1] = lines2[j-1]$ **then**
            $dp[i][j] \leftarrow dp[i-1][j-1]$
        else
            $delete_distance \leftarrow dp[i-1][j] + 1$;
            $insert_distance \leftarrow dp[i][j-1] + 1$;
            `// The following condition control for replace_distance is different from the conventional Levenshtein distance algorithm`
            if *common_prefix_length*$(lines[i-1], lines2[j-1]) >= 6$ **then**
                $replace_distance \leftarrow dp[i-1][j-1] + 1$;
            else
                $replace_distance \leftarrow dp[i-1][j-1] + 999$;
            $dp[i][j] \leftarrow \min(delete_distance, insert_distance, replace_distance)$;
    end
end
Backtracking to extract operations pairs;
$i \leftarrow n,\ j \leftarrow m$;
$inserted, deleted, replaced \leftarrow [\,], [\,], [\,]$;
while $i > 0$ **or** $j > 0$ **do**
    if $i > 0$ **and** $j > 0$ **and** $lines1[i-1] = lines2[j-1]$ **then**
        $i \leftarrow i-1,\ j \leftarrow j-1$; `// match`
    else if $j > 0$ **and** $dp[i][j] = dp[i][j-1] + 1$ **then**
        $inserted$.`append`$((i, j))$;
        $j \leftarrow j-1$; `// insert`
    else if $i > 0$ **and** $dp[i][j] = dp[i-1][j] + 1$ **then**
        $deleted$.`append`$((i, j))$;
        $i \leftarrow i-1$; `// delete`
    else if $i > 0$ **and** $j > 0$ **and** $dp[i][j] = dp[i-1][j-1] + 1$ **then**
        $replaced$.`append`$((i, j))$;
        $i \leftarrow i-1,\ j \leftarrow j-1$; `// replace`
    else
        raise `ValueError`("Unexpected Case");
end
return $inserted, deleted, replaced$;

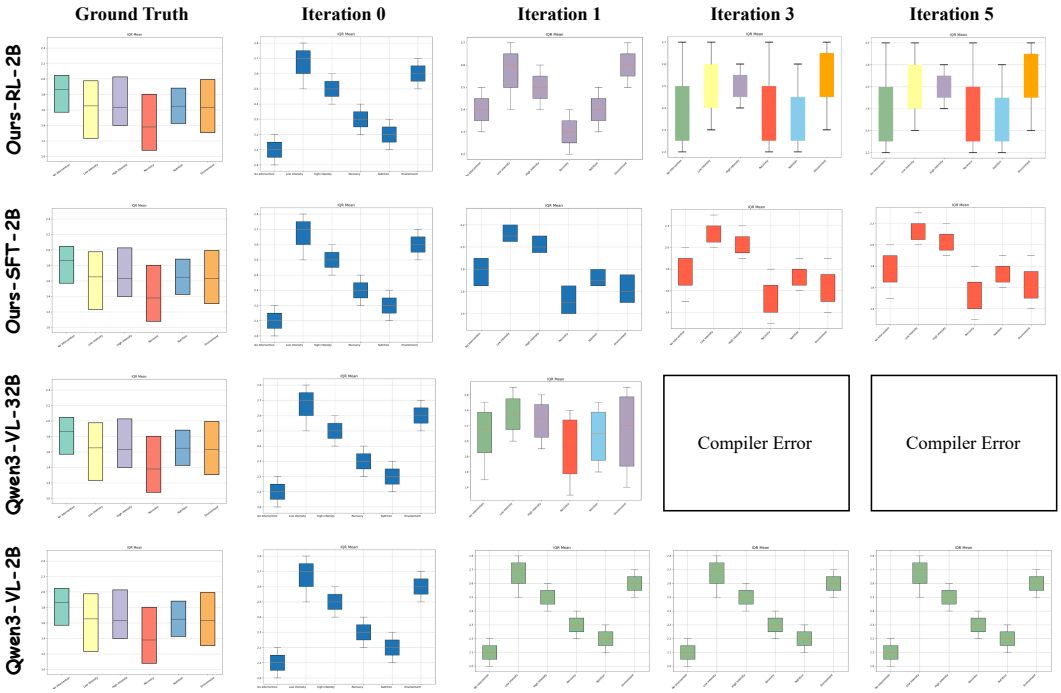


Figure 10: A case of test-time refinement for plot-to-code generation with different visual-code diagnostic verifiers.

NeurIPS Paper Checklist

1. **Claims**

   Question: Do the main claims made in the abstract and introduction accurately reflect the paper's contributions and scope?

   Answer: [Yes]

   Justification: Section 1 and section 0.

   Guidelines:

   - The answer [N/A] means that the abstract and introduction do not include the claims made in the paper.
   - The abstract and/or introduction should clearly state the claims made, including the contributions made in the paper and important assumptions and limitations. A [No] or [N/A] answer to this question will not be perceived well by the reviewers.
   - The claims made should match theoretical and experimental results, and reflect how much the results can be expected to generalize to other settings.
   - It is fine to include aspirational goals as motivation as long as it is clear that these goals are not attained by the paper.

2. **Limitations**

   Question: Does the paper discuss the limitations of the work performed by the authors?

   Answer: [Yes]

   Justification: Section 4.3 Analysis on Specialized Plot-to-Code Generators.

   Guidelines:

   - The answer [N/A] means that the paper has no limitation while the answer [No] means that the paper has limitations, but those are not discussed in the paper.
   - The authors are encouraged to create a separate "Limitations" section in their paper.
   - The paper should point out any strong assumptions and how robust the results are to violations of these assumptions (e.g., independence assumptions, noiseless settings, model well-specification, asymptotic approximations only holding locally). The authors should reflect on how these assumptions might be violated in practice and what the implications would be.
   - The authors should reflect on the scope of the claims made, e.g., if the approach was only tested on a few datasets or with a few runs. In general, empirical results often depend on implicit assumptions, which should be articulated.
   - The authors should reflect on the factors that influence the performance of the approach. For example, a facial recognition algorithm may perform poorly when image resolution is low or images are taken in low lighting. Or a speech-to-text system might not be used reliably to provide closed captions for online lectures because it fails to handle technical jargon.
   - The authors should discuss the computational efficiency of the proposed algorithms and how they scale with dataset size.
   - If applicable, the authors should discuss possible limitations of their approach to address problems of privacy and fairness.
   - While the authors might fear that complete honesty about limitations might be used by reviewers as grounds for rejection, a worse outcome might be that reviewers discover limitations that aren't acknowledged in the paper. The authors should use their best judgment and recognize that individual actions in favor of transparency play an important role in developing norms that preserve the integrity of the community. Reviewers will be specifically instructed to not penalize honesty concerning limitations.

3. **Theory assumptions and proofs**

   Question: For each theoretical result, does the paper provide the full set of assumptions and a complete (and correct) proof?

   Answer: [N/A]

Justification: This paper does not include theoretical results.

Guidelines:

- The answer [N/A] means that the paper does not include theoretical results.
- All the theorems, formulas, and proofs in the paper should be numbered and cross-referenced.
- All assumptions should be clearly stated or referenced in the statement of any theorems.
- The proofs can either appear in the main paper or the supplemental material, but if they appear in the supplemental material, the authors are encouraged to provide a short proof sketch to provide intuition.
- Inversely, any informal proof provided in the core of the paper should be complemented by formal proofs provided in appendix or supplemental material.
- Theorems and Lemmas that the proof relies upon should be properly referenced.

4. **Experimental result reproducibility**

   Question: Does the paper fully disclose all the information needed to reproduce the main experimental results of the paper to the extent that it affects the main claims and/or conclusions of the paper (regardless of whether the code and data are provided or not)?

   Answer: [Yes]

   Justification: Section 4.

   Guidelines:

   - The answer [N/A] means that the paper does not include experiments.
   - If the paper includes experiments, a [No] answer to this question will not be perceived well by the reviewers: Making the paper reproducible is important, regardless of whether the code and data are provided or not.
   - If the contribution is a dataset and/or model, the authors should describe the steps taken to make their results reproducible or verifiable.
   - Depending on the contribution, reproducibility can be accomplished in various ways. For example, if the contribution is a novel architecture, describing the architecture fully might suffice, or if the contribution is a specific model and empirical evaluation, it may be necessary to either make it possible for others to replicate the model with the same dataset, or provide access to the model. In general. releasing code and data is often one good way to accomplish this, but reproducibility can also be provided via detailed instructions for how to replicate the results, access to a hosted model (e.g., in the case of a large language model), releasing of a model checkpoint, or other means that are appropriate to the research performed.
   - While NeurIPS does not require releasing code, the conference does require all submissions to provide some reasonable avenue for reproducibility, which may depend on the nature of the contribution. For example
     (a) If the contribution is primarily a new algorithm, the paper should make it clear how to reproduce that algorithm.
     (b) If the contribution is primarily a new model architecture, the paper should describe the architecture clearly and fully.
     (c) If the contribution is a new model (e.g., a large language model), then there should either be a way to access this model for reproducing the results or a way to reproduce the model (e.g., with an open-source dataset or instructions for how to construct the dataset).
     (d) We recognize that reproducibility may be tricky in some cases, in which case authors are welcome to describe the particular way they provide for reproducibility. In the case of closed-source models, it may be that access to the model is limited in some way (e.g., to registered users), but it should be possible for other researchers to have some path to reproducing or verifying the results.

5. **Open access to data and code**

   Question: Does the paper provide open access to the data and code, with sufficient instructions to faithfully reproduce the main experimental results, as described in supplemental material?

Answer: [Yes]

Justification: Section 1.

Guidelines:

- The answer [N/A] means that paper does not include experiments requiring code.
- Please see the NeurIPS code and data submission guidelines (`https://neurips.cc/public/guides/CodeSubmissionPolicy`) for more details.
- While we encourage the release of code and data, we understand that this might not be possible, so [No] is an acceptable answer. Papers cannot be rejected simply for not including code, unless this is central to the contribution (e.g., for a new open-source benchmark).
- The instructions should contain the exact command and environment needed to run to reproduce the results. See the NeurIPS code and data submission guidelines (`https://neurips.cc/public/guides/CodeSubmissionPolicy`) for more details.
- The authors should provide instructions on data access and preparation, including how to access the raw data, preprocessed data, intermediate data, and generated data, etc.
- The authors should provide scripts to reproduce all experimental results for the new proposed method and baselines. If only a subset of experiments are reproducible, they should state which ones are omitted from the script and why.
- At submission time, to preserve anonymity, the authors should release anonymized versions (if applicable).
- Providing as much information as possible in supplemental material (appended to the paper) is recommended, but including URLs to data and code is permitted.

6. **Experimental setting/details**

   Question: Does the paper specify all the training and test details (e.g., data splits, hyperparameters, how they were chosen, type of optimizer) necessary to understand the results?

   Answer: [Yes]

   Justification: Section 1.

   Guidelines:

   - The answer [N/A] means that the paper does not include experiments.
   - The experimental setting should be presented in the core of the paper to a level of detail that is necessary to appreciate the results and make sense of them.
   - The full details can be provided either with the code, in appendix, or as supplemental material.

7. **Experiment statistical significance**

   Question: Does the paper report error bars suitably and correctly defined or other appropriate information about the statistical significance of the experiments?

   Answer: [Yes]

   Justification: Section 4.

   Guidelines:

   - The answer [N/A] means that the paper does not include experiments.
   - The authors should answer [Yes] if the results are accompanied by error bars, confidence intervals, or statistical significance tests, at least for the experiments that support the main claims of the paper.
   - The factors of variability that the error bars are capturing should be clearly stated (for example, train/test split, initialization, random drawing of some parameter, or overall run with given experimental conditions).
   - The method for calculating the error bars should be explained (closed form formula, call to a library function, bootstrap, etc.)
   - The assumptions made should be given (e.g., Normally distributed errors).
   - It should be clear whether the error bar is the standard deviation or the standard error of the mean.

- It is OK to report 1-sigma error bars, but one should state it. The authors should preferably report a 2-sigma error bar than state that they have a 96% CI, if the hypothesis of Normality of errors is not verified.
- For asymmetric distributions, the authors should be careful not to show in tables or figures symmetric error bars that would yield results that are out of range (e.g., negative error rates).
- If error bars are reported in tables or plots, the authors should explain in the text how they were calculated and reference the corresponding figures or tables in the text.

8. **Experiments compute resources**

   Question: For each experiment, does the paper provide sufficient information on the computer resources (type of compute workers, memory, time of execution) needed to reproduce the experiments?

   Answer: [Yes]

   Justification: Appendix hyperparameters subsection.

   Guidelines:

   - The answer [N/A] means that the paper does not include experiments.
   - The paper should indicate the type of compute workers CPU or GPU, internal cluster, or cloud provider, including relevant memory and storage.
   - The paper should provide the amount of compute required for each of the individual experimental runs as well as estimate the total compute.
   - The paper should disclose whether the full research project required more compute than the experiments reported in the paper (e.g., preliminary or failed experiments that didn't make it into the paper).

9. **Code of ethics**

   Question: Does the research conducted in the paper conform, in every respect, with the NeurIPS Code of Ethics `https://neurips.cc/public/EthicsGuidelines`?

   Answer: [Yes]

   Justification: Section 1.

   Guidelines:

   - The answer [N/A] means that the authors have not reviewed the NeurIPS Code of Ethics.
   - If the authors answer [No], they should explain the special circumstances that require a deviation from the Code of Ethics.
   - The authors should make sure to preserve anonymity (e.g., if there is a special consideration due to laws or regulations in their jurisdiction).

10. **Broader impacts**

    Question: Does the paper discuss both potential positive societal impacts and negative societal impacts of the work performed?

    Answer: [Yes]

    Justification: Section 4.

    Guidelines:

    - The answer [N/A] means that there is no societal impact of the work performed.
    - If the authors answer [N/A] or [No], they should explain why their work has no societal impact or why the paper does not address societal impact.
    - Examples of negative societal impacts include potential malicious or unintended uses (e.g., disinformation, generating fake profiles, surveillance), fairness considerations (e.g., deployment of technologies that could make decisions that unfairly impact specific groups), privacy considerations, and security considerations.

- The conference expects that many papers will be foundational research and not tied to particular applications, let alone deployments. However, if there is a direct path to any negative applications, the authors should point it out. For example, it is legitimate to point out that an improvement in the quality of generative models could be used to generate Deepfakes for disinformation. On the other hand, it is not needed to point out that a generic algorithm for optimizing neural networks could enable people to train models that generate Deepfakes faster.
- The authors should consider possible harms that could arise when the technology is being used as intended and functioning correctly, harms that could arise when the technology is being used as intended but gives incorrect results, and harms following from (intentional or unintentional) misuse of the technology.
- If there are negative societal impacts, the authors could also discuss possible mitigation strategies (e.g., gated release of models, providing defenses in addition to attacks, mechanisms for monitoring misuse, mechanisms to monitor how a system learns from feedback over time, improving the efficiency and accessibility of ML).

11. **Safeguards**

    Question: Does the paper describe safeguards that have been put in place for responsible release of data or models that have a high risk for misuse (e.g., pre-trained language models, image generators, or scraped datasets)?

    Answer: [Yes]

    Justification: Section 4.

    Guidelines:

    - The answer [N/A] means that the paper poses no such risks.
    - Released models that have a high risk for misuse or dual-use should be released with necessary safeguards to allow for controlled use of the model, for example by requiring that users adhere to usage guidelines or restrictions to access the model or implementing safety filters.
    - Datasets that have been scraped from the Internet could pose safety risks. The authors should describe how they avoided releasing unsafe images.
    - We recognize that providing effective safeguards is challenging, and many papers do not require this, but we encourage authors to take this into account and make a best faith effort.

12. **Licenses for existing assets**

    Question: Are the creators or original owners of assets (e.g., code, data, models), used in the paper, properly credited and are the license and terms of use explicitly mentioned and properly respected?

    Answer: [Yes]

    Justification: Section 4.

    Guidelines:

    - The answer [N/A] means that the paper does not use existing assets.
    - The authors should cite the original paper that produced the code package or dataset.
    - The authors should state which version of the asset is used and, if possible, include a URL.
    - The name of the license (e.g., CC-BY 4.0) should be included for each asset.
    - For scraped data from a particular source (e.g., website), the copyright and terms of service of that source should be provided.
    - If assets are released, the license, copyright information, and terms of use in the package should be provided. For popular datasets, `paperswithcode.com/datasets` has curated licenses for some datasets. Their licensing guide can help determine the license of a dataset.
    - For existing datasets that are re-packaged, both the original license and the license of the derived asset (if it has changed) should be provided.

- If this information is not available online, the authors are encouraged to reach out to the asset's creators.

13. **New assets**

    Question: Are new assets introduced in the paper well documented and is the documentation provided alongside the assets?

    Answer: [Yes]

    Justification: Section 1.

    Guidelines:

    - The answer [N/A] means that the paper does not release new assets.
    - Researchers should communicate the details of the dataset/code/model as part of their submissions via structured templates. This includes details about training, license, limitations, etc.
    - The paper should discuss whether and how consent was obtained from people whose asset is used.
    - At submission time, remember to anonymize your assets (if applicable). You can either create an anonymized URL or include an anonymized zip file.

14. **Crowdsourcing and research with human subjects**

    Question: For crowdsourcing experiments and research with human subjects, does the paper include the full text of instructions given to participants and screenshots, if applicable, as well as details about compensation (if any)?

    Answer: [N/A]

    Justification: The paper does not involve crowdsourcing.

    Guidelines:

    - The answer [N/A] means that the paper does not involve crowdsourcing nor research with human subjects.
    - Including this information in the supplemental material is fine, but if the main contribution of the paper involves human subjects, then as much detail as possible should be included in the main paper.
    - According to the NeurIPS Code of Ethics, workers involved in data collection, curation, or other labor should be paid at least the minimum wage in the country of the data collector.

15. **Institutional review board (IRB) approvals or equivalent for research with human subjects**

    Question: Does the paper describe potential risks incurred by study participants, whether such risks were disclosed to the subjects, and whether Institutional Review Board (IRB) approvals (or an equivalent approval/review based on the requirements of your country or institution) were obtained?

    Answer: [N/A]

    Justification: This paper does not involve crowdsourcing nor research with human subjects.

    Guidelines:

    - The answer [N/A] means that the paper does not involve crowdsourcing nor research with human subjects.
    - Depending on the country in which research is conducted, IRB approval (or equivalent) may be required for any human subjects research. If you obtained IRB approval, you should clearly state this in the paper.
    - We recognize that the procedures for this may vary significantly between institutions and locations, and we expect authors to adhere to the NeurIPS Code of Ethics and the guidelines for their institution.
    - For initial submissions, do not include any information that would break anonymity (if applicable), such as the institution conducting the review.

16. **Declaration of LLM usage**

Question: Does the paper describe the usage of LLMs if it is an important, original, or non-standard component of the core methods in this research? Note that if the LLM is used only for writing, editing, or formatting purposes and does *not* impact the core methodology, scientific rigor, or originality of the research, declaration is not required.

Answer: [Yes]

Justification: This paper uses gpt-4o in data construction and evaluation.

Guidelines:

- The answer [N/A] means that the core method development in this research does not involve LLMs as any important, original, or non-standard components.
- Please refer to our LLM policy in the NeurIPS handbook for what should or should not be described.