METAL: A Multi-Agent Framework for Chart Generation with Test-Time Scaling

Bingxuan Li ♠, **Yiwei Wang** ♢♠, **Jiuxiang Gu** ♡, **Kai-Wei Chang** ♠, **Nanyun Peng** ♠
♠ University of California, Los Angeles, ♢ University of California, Merced, ♡ Adobe Research
bingxuan@ucla.edu

https://metal-framework.github.io

Abstract

Chart generation aims to generate code to produce charts satisfying the desired visual properties, e.g., texts, layout, color, and type. It has great potential to empower the automatic professional report generation in financial analysis, research presentation, education, and healthcare. In this work, we build a vision-language model (VLM) based *multi-agent* framework for effective automatic chart generation. Generating high-quality charts requires both strong visual design skills and precise coding capabilities that embed the desired visual properties into code. Such a complex multi-modal reasoning process is difficult for direct prompting of VLMs. To resolve these challenges, we propose **METAL**, a multi-agent framework that decomposes the task of chart generation into the iterative collaboration among specialized agents. **METAL** achieves 5.2% improvement in accuracy over the current best result in the chart generation task. The **METAL** framework exhibits the phenomenon of test-time scaling: its performance increases monotonically as the logarithmic computational budget grows from 512 to 8192 tokens. In addition, we find that separating different modalities during the critique process of **METAL** boosts the self-correction capability of VLMs in the multimodal context.

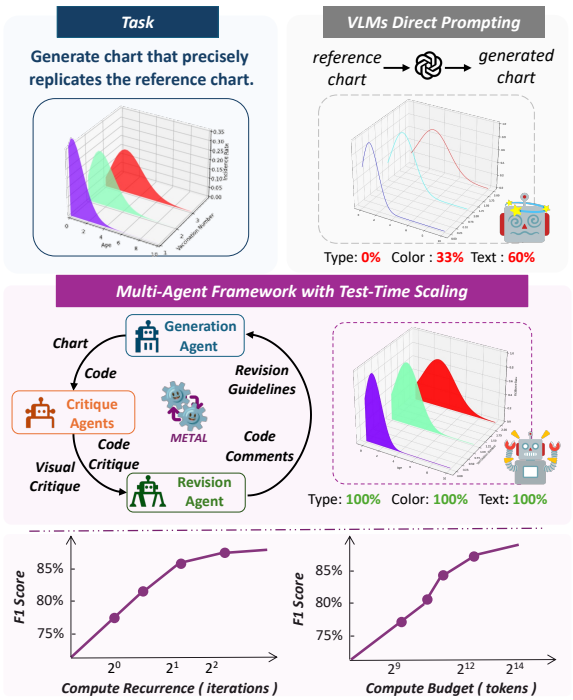

Figure 1: Direct prompting of current VLMs (e.g. GPT-4o) often fails to generate charts that accurately replicate reference charts, resulting in errors in structure, color, and text alignment. Our proposed approach, **METAL**, tackle this challenge with iterative refinement through generation, critique, and revision. Our experiments show that increasing the logarithm of test-time compute recurrence and token usage leads to improved accuracy.

1 Introduction

Data visualization through charts is an important part of the communication and research life cycle. Well-designed visualizations help distill complex data into digestible insights, allowing researchers, analysts, and stakeholders to identify relationships that might remain hidden in raw data (Qin et al., 2020; Xu et al., 2023; Yang et al., 2024).

Recent advancements in vision language models (VLMs), such as GPT-4V (OpenAI, 2023) and LLaVA (Li et al., 2024), have expanded the capabilities of language models in tackling complex multi-modal problem-solving tasks. These breakthroughs have sparked growing interests in designing *intelligent AI assistants* to help humans with limited coding expertise create compelling charts, leading to the emergence of a complex multi-modal generation task with crucial practical value - Chart to code generation task (Wu et al., 2024a; Han et al., 2023; Shi et al., 2024).

The chart to code generation task focuses on automatically generating visualization code based on visual references. This task embodies a **highly challenging visually-grounded code generation problem** that demands robust visual understand-

ing and advanced reasoning. The model must interpret complex visual elements—such as layouts, color schemes, and data relationships—and translate them into syntactically correct, semantically meaningful code. Successfully addressing this challenge not only improves chart replication but also paves the way for advancing the general capabilities of VLMs in multimodal learning and program synthesis.

In this paper, we present **METAL**, a multi-agent framework designed for chart generation. Our framework decomposes this complex multimodal reasoning task into four specialized roles, each handled by a specialized agent: (1) *Generation Agent*: Responsible for the initial translation of chart images into the corresponding code. (2) *Visual Critique Agent*: Analyzes and identifies visual differences between the reference chart and the generated output. (3) *Code Critique Agent*: Reviews the generated code and suggests improvements to better match the reference chart. (4) *Revision Agent*: Implements code modifications based on the combined feedback from both critique agents. During inference, these agents collaborate iteratively, critiquing and refining the code until the rendered chart achieves the desired accuracy.

As illustrated in Figure 1, current state-of-the-art VLMs, such as GPT-4o, often fail to accurately interpret and reproduce the intricate visual elements and relationships embedded in reference charts. Existing solutions, such as Best-of-N and Hint-enhanced (Wang et al., 2024), have not effectively improved upon direct prompting of VLMs. The core challenge in leveraging VLMs for chart generation lies in effectively integrating visual comprehension with code synthesis. This complex task exceeds the capabilities of the single model or single agent. In contrast to existing methods, our approach delivers more concrete and targeted feedback, and iteratively refines outputs through the multi-agentic framework, leading to enhanced chart generation performance.

Experiment results show that our framework improves chart generation accuracy by over 11.33%, demonstrating its potential to significantly enhance VLMs' ability to integrate visual understanding with code synthesis.

Furthermore, we have two key findings through in-depth analysis: (1) **Test-time scaling in the METAL framework**: We found that there is a near-linear relationship between the performance and the logarithm of the computational budget in experiments. Specifically, the performance of **METAL** increases monotonically as the logarithm of the computational budget grows from 512 to 8192 tokens. (2) **Modality-tailored critiques enhance self-correction**: We observe that explicitly separating different modalities during the critique process—such as visual evaluation and code analysis—substantially enhances the multimodal self-correction capabilities of VLMs. Ablation study shows that **METAL** with the separate-critique design achieves a 5.16% improvement over the single-critique baseline.

In summary, we present **METAL**, a VLMs-based multi-agent framework, which achieves significant improvements over the current best methods for chart generation, and our insights into test-time scaling and multi-modal critique offer a promising pathway for enhanced visually-grounded code generation with VLMs.

2 Related Works

We discuss three lines of related work: chart-to-code generation, multi-agent framework, and test-time scaling research.

2.1 Chart Generation with VLMs

Chart generation, or chart-to-code generation, is an emerging task aimed at automatically translating visual representations of charts into corresponding visualization code (Shi et al., 2024; Wu et al., 2024a). This task is inherently challenging as it requires both visual understanding and precise code synthesis, often demanding complex reasoning over visual elements.

Recent advances in Vision-Language Models (VLMs) have expanded the capabilities of language models in tackling complex multimodal problem-solving tasks, such as visually-grounded code generation. Leading proprietary models, such as GPT-4V (OpenAI, 2023), Gemini (Google, 2023), and Claude-3 (Anthropic, 2024), have demonstrated impressive capabilities in understanding complex visual patterns. The open-source community has contributed models like LLaVA (Xu et al., 2024; Li et al., 2024), Qwen-VL (Bai et al., 2023), and DeepSeek-VL (Lu et al., 2024), which provide researchers with greater flexibility for specific applications like chart generation.

Despite these advancements, current VLMs often struggle with accurately interpreting chart structures and faithfully reproducing visualization code.

2.2 Multi-Agents Framework

Many researchers have suggested a paradigm shift from single monolithic models to compound systems comprising multiple specialized components (Zaharia et al., 2024; Du and Kaelbling, 2024). One prominent example is the multi-agent framework.

LLMs-driven multi-agent framework has been widely explored in various domains, including narrative generation (Huot et al., 2024), financial trading (Xiao et al., 2024), and cooperative problem-solving (Du et al., 2023).

Our work investigates the application of multi-agent framework to the visually-grounded code generation task.

2.3 Test-Time Scaling

Inference strategies have been a long-studied topic in the field of language processing. Traditional approaches include greedy decoding (Teller, 2000), beam search (Graves, 2012), and Best-of-N.

Recent research has explored test-time scaling law for language model inference. For example, Wu et al. (2024b) empirically demonstrated that optimizing test-time compute allocation can significantly enhance problem-solving performance, while Zhang et al. (2024) and Snell et al. (2024) highlighted that dynamic adjustments in sample allocation can maximize efficiency under compute constraints. Although these studies collectively underscore the promise of test-time scaling for enhancing reasoning performance of LLMs, its existence in other contexts, such as different model types and application to cross-modal generation, remains under-explored.

3 Method

In this section, we introduce our method for generating precise chart representations from a given reference chart. Section 3.1 formally defines the task, Section 3.2 outlines the components of our proposed approach **METAL**, and Section 3.3 presents the inference process of **METAL**.

3.1 Task Definition

Given a reference chart image x_{ref} and a chart generation model, the objective is to learn the mapping

$$f : x_{\text{ref}} \to y,$$

where y is a programmatic specification (e.g., Python code). When executed, y should render a chart $O(y)$ that faithfully replicates the reference x_{ref}.

3.2 METAL

As illustrated in Figure 2, **METAL** is structured with four specialized agents (G, C, V, R) and a multi-criteria verifier. All components collaborate together to iteratively refine the final output, making it more accurately replicates the reference chart. The framework is composed as follows:

Generation Agent (G) This agent is tasked to generate an initial program from the reference:

$$y_0 = G(x_{\text{ref}}), \quad G : \mathcal{X} \to \mathcal{Y}.$$

This serves as the basis for further refinement.

Visual Critique Agent (V) This agent is tasked to assess the rendered chart $O(y_t)$ against x_{ref} to detect visual discrepancies:

$$v_t = V(O(y_t), x_{\text{ref}}), \quad V : \mathcal{O} \times \mathcal{X} \to \mathcal{V}.$$

Here, $\mathcal{O}$ represents the space of visual outputs, and $\mathcal{V}$ denotes the space of visual feedback metrics.

Code Critique Agent (C) This agent is tasked to review the generated code and provide structured critique to improve the generated code:

$$c_t = C(y_t), \quad C : \mathcal{Y} \to \mathcal{C}.$$

$\mathcal{C}$ represents the set of code critique messages ensuring correctness and efficiency.

Revision Agent (R) This agent integrates feedback from both critique agents to update the generated code:

$$y_{t+1} = R(y_t, v_t, c_t), \quad R : \mathcal{Y} \times \mathcal{V} \times \mathcal{C} \to \mathcal{Y}.$$

Multi-Criteria Verifier We design a heuristic-based verification function to evaluate the chart quality. Let m_j be the verification metrics for $j = 1, 2, 3$, and let θ^t be dynamic thresholds. Then,

$$Q_t(O(y_t), x_{\text{ref}}) = \begin{cases} 1, & \bigwedge_{j=1}^{3} m_j(O(y_t), x_{\text{ref}}) \geq \theta^t \\ 0, & \text{otherwise.} \end{cases}$$

More details on the implementation of the verifier, such as each verification metric, are introduced in Appendix A.2.

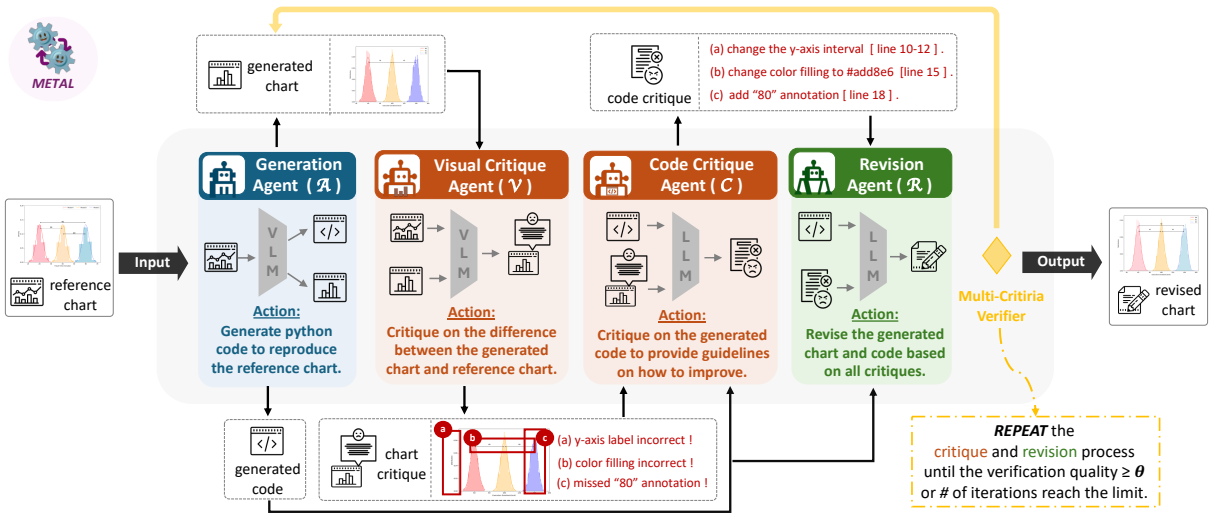

Figure 2: Overview of METAL: A multi-agents system that consists of four specialized agents working in an iterative pipeline: (1) Generation Agent creates initial Python code to reproduce the reference chart, (2) Visual Critique Agent identifies visual discrepancies between the generated and reference charts, (3) Code Critique Agent analyzes the code and provides specific improvement guidelines, and (4) Revision Agent modifies the code based on the critiques. The process iterates until either reaching the accuracy target or maximum attempts limit.

3.3 Inference Procedure

During inference, METAL iteratively refines the generated code until the rendered chart meets a predefined quality threshold. The refinement process is as follows:

$$y_0 = G(x_{\text{ref}}), \tag{1}$$
$$v_t = V(O(y_t), x_{\text{ref}}), \tag{2}$$
$$c_t = C(y_t), \tag{3}$$
$$y_{t+1} = R(y_t, v_t, c_t). \tag{4}$$

The iterations terminate when

$$Q_t(O(y_t), x_{\text{ref}}) < \epsilon,$$

where $\epsilon > 0$ is the predefined threshold for the discrepancy between the generated chart and the reference chart.

4 Experiments

In this section, we systematically evaluate METAL. Section 4.1 details the experimental setup, Section 4.2 presents the results, and Section 4.3 provides an ablation study to further elucidate the model's performance.

4.1 Experiment Setup

Dataset We select the ChartMIMIC dataset to evaluate METAL. It is a benchmark that includes 1,000 human-curated (figure, instruction, code) triplets, which represent the authentic chart use cases found in scientific papers across various domains. These charts span 18 regular types and 4 advanced types, diversifying into 191 subcategories (Shi et al., 2024) .

Automatic Evaluation Metric Following the approach in (Shi et al., 2024), we assess four key low-level chart elements: text, layout, type, and color. During code execution, relevant information for each element is logged for evaluation. We then compare each element in the generated chart to its counterpart in the reference chart and calculate the F1 score. Note that the evaluation metric used here *differs* from the multi-criteria verification metric described in the section 3.

Base Model We assess the effectiveness of METAL on both open-source and closed-source vision-language models. Specifically, we evaluate GPT-4o (Hurst et al., 2024) and LLaMA 3.2-11B (AI@Meta, 2024). The details of model size and computation budget are introduced in Appendix B.

Implementation Details The implementation details of each component of METAL and the baselines are described in Appendix A. We include the prompt templates for each VLM-driven agent and the baselines in Appendix C.

Baselines We compare METAL against three baseline methods, detailed as follows:

Base Model	Method	Average F1 Score				
		Text	*Type*	*Color*	*Layout*	*Average*
LLaMA 3.2-11B	Direct Prompting	36.70%	37.07%	33.46%	54.56%	40.45%
	Hint-Enhanced Prompting	38.82%	38.47%	36.82%	51.22%	41.33%
	Best-of-N (n = 5)	40.28%	36.60%	38.43%	57.22%	43.13%
	METAL (n = 5)	**46.69 %**↑	**54.42%**↑	**47.32 %**↑	**68.10%**↑	**51.78 %**↑
GPT-4o	Direct Prompting	74.83%	81.24%	74.24%	94.76%	81.26%
	Hint-Enhanced Prompting	77.02%	80.84%	72.75%	93.89%	81.12%
	Best-of-N (n = 5)	75.47%	82.16%	75.30%	96.37%	81.70%
	METAL (n = 5)	**86.31%** ↑	**84.17%** ↑	**79.86**↑	**95.50%**↑	**86.46%**↑

Table 1: Performance comparison of **METAL** and baseline methods across various base models using four evaluation metrics: Text, Type, Color, and Layout. The best performance for each metric on each base model is highlighted in **bold**. Our approach consistently outperforms the baselines, achieving the highest average F1 scores across both models, with significant improvements observed in all evaluation categories.

1. *Direct Prompting*: This baseline generates charts directly from the input prompt without any modifications or explicit guidance. It relies solely on the model's inherent ability.
2. *Hint-Enhanced Prompting*: In this approach, the input prompt is augmented with additional hints or structured guidance to help the model better understand the desired chart components (Wang et al., 2024). Specifically, we augment the generation with a self-generated short description of a chart that provides context for elements such as layout, text, type, and color.
3. *Best-of-N*: This baseline generates multiple candidate charts in parallel, and the one that best meets a predefined verification metric is selected. We compare against Best-of-N by matching the number of iterations used in our approach.

4.2 Experiment Results

The primary research question of the experiment was to assess whether our proposed method could improve the performance of the base model in the chart generation task. Table 1 presents the experiment result.

For the LLaMA base model, the results indicate that the performance of baseline methods varied moderately. Direct Prompting and Hint-Enhanced Prompting achieved average F1 scores of 40.45% and 41.33%, respectively, while Best-of-N reached 43.13%. In contrast, **METAL** yielded an average F1 score of 51.78% with improvements observed in each metric. The average F1 score improves by 11.33% over Direct Prompting with 5 test-time compute recurrences, which is a significant improvement.

Similarly, for the GPT base model, the baseline methods demonstrated high performance with average F1 scores ranging from 81.12% to 81.70%. However, **METAL** outperformed these methods by a considerable margin, achieving an average F1 score of 86.46%. Specifically, our method improved 5.2% in average over Direct Prompting.

These results clearly demonstrate that our approach consistently improves performance across both base models and all evaluation metrics. In particular, the significant gains observed in the Text and Layout metric, along with the overall increase in average F1 scores, indicate that our model effectively captures both the structural attributes and finer details of visual data. This enhancement not only boosts the performance of both open-source and closed-source vision-language models but also maintains a high level of consistency across all metrics, underscoring the robustness and generalizability of our method.

4.3 Ablation Study

To further analyze the impact of different components, we performed an ablation study by selectively removing key elements of **METAL** to assess the influence of each component on the overall performance.

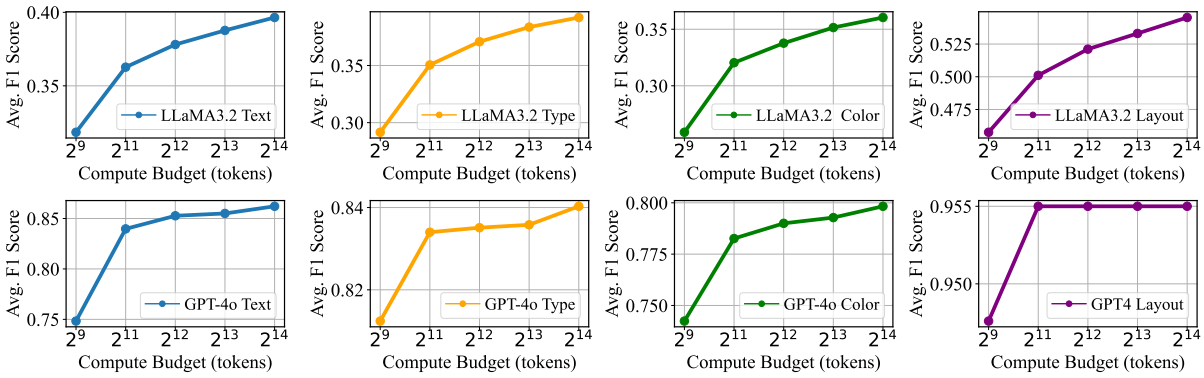

Figure 3: The performance of METAL demonstrates an near-linear relationship with the log of compute budget.

Variations Setup We evaluate the following variants of METAL to assess the contribution of each agent:

- METAL_V: Uses only the visual critique agent, omitting the code critique component.
- METAL_C: Uses only the code critique agent, omitting the visual critique component.
- METAL_S: Combines the visual and code critique agents into a single, unified critique agent.

We implement each variation with GPT-4o as the base model. The prompt templates of each variation are attached to Appendix C.

Results As shown in Table 2, the full METAL achieves the highest average F1 score, outperforming all ablated variants. Specifically, when only the visual critique agent is used (METAL_V), the system obtains an average F1 score of 84.12%, while the variant using only the code critique agent (METAL_C) achieves 82.89%. The unified agent variant (METAL_S) yields the lowest average performance at 81.22%.

These results indicate that both the visual and code critique agents play crucial roles in enhancing the model's performance. The degradation observed in the ablated variants highlights that removing either component, or merging them into a single unit, compromises the system's ability to effectively capture and refine the critical attributes of the visual data.

5 Analysis and Discussion

We analyze the experimental results in this section. We highlight two interesting findings: Test-time scaling in Multi-Agent system (Section 5.1), and modality-tailored critiques enhance the self-correction ability (Section 5.2). Additionally, we discuss the advantage of METAL (Section 5.3), and the benefit of agentic design (Section 5.4)

Method	Average F1 Scores (%)				
	Text	*Type*	*Color*	*Layout*	*Average*
METAL_V	83.43	82.57	77.57	93.69	84.12
METAL_C	82.35	80.90	76.69	91.93	82.89
METAL_S	80.26	78.88	74.50	89.82	81.22
METAL	**86.31**	**84.17**	**79.86**	**95.50**	**86.38**

Table 2: Ablation study on different variants of METAL across four evaluation metrics. The best performance for each metric is highlighted in **bold**.

5.1 Test-Time Scaling

We investigate the relationship between the test-time computational budget and model performance. As illustrated in Figure 3, our analysis reveals an interesting trend: increasing the logarithm of the computational budget leads to continuous performance improvements. This near-linear relationship indicates the test-time scaling phenomenon, demonstrating that allowing more iterations during inference could potentially enhance performance.

One potential reason for this phenomenon is the strong self-improvement capability of METAL. Our framework is designed so that specialized agents iteratively collaborate, allowing each agent to refine its output based on feedback from others. With each iteration, errors are corrected and insights from different modalities are integrated, leading to incremental performance gains. This continual refinement process leverages the strengths of individual agents, resulting in the self-improvement

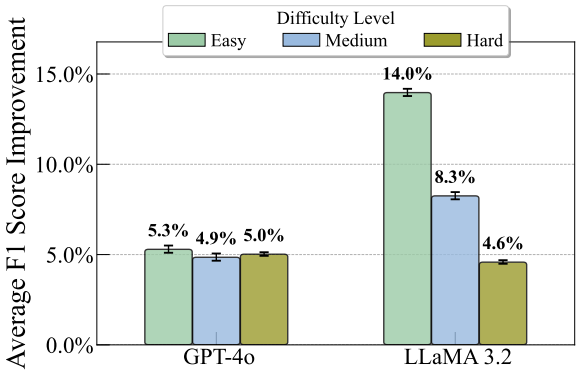

Figure 4: Performance gain after 5 compute recurrences of **METAL** over different difficulty.

capability that drives the observed performance enhancements as computational resources increase.

Due to limited resources, we have not extended the experiment range further. However, the observed scaling implies that the framework can benefit from more iterations of collaborative self-improvement. We leave a more comprehensive exploration of this potential to the future work.

5.2 Modality-Tailored Critiques

From the ablation study result shown in Table 2, we observed that separating visual and code critiques enhances the model's self-correction capabilities. In contrast, **METAL**$_S$ struggles to effectively self-improve in the chart-to-code generation task.

We identify two potential reasons for this observation. First, combining both visual and code inputs results in an extended context that can overwhelm the model, leading to information loss. This dilution makes it difficult to capture key details from each modality, resulting in less accurate critiques and a reduction in overall self-correction effectiveness. Second, the self-critique process for chart generation involves distinct requirements: visual data demands spatial understanding, color analysis, and fine detail recognition, while code data requires strict adherence to syntax and logical consistency. A unified critique approach is ill-suited to address these differing needs. Without modality-specific feedback, the model struggles to detect and correct errors unique to each data type.

These findings suggest that self-correction in the multimodal context can be enhanced by leveraging tailored critique strategies for each modality.

5.3 Why METAL

We believe **METAL** provides three advantages. First, by assigning specialized tasks to individual agents, the system effectively reduces error propagation. During inference, each agent evaluates whether to take action based on the available information and insights from other agents. This process enables each agent to serve as a safeguard, detecting and correcting mistakes before they escalate.

Second, the modular design of **METAL** enables easy modification and adaptation. For instance, one can integrate different base models tailored for specific tasks—such as employing a critique-trained model for critique agents and a generation-trained model for generation agents—to maximize overall performance.

Third, **METAL** is robust with the strong base model. Figure 4 compares the performance of **METAL** to that of Direct Prompting over five iterations across varying chart difficulty levels. **METAL** with the GPT-4O base model achieved consistent improvements regardless of difficulty. When using LLAMA 3.2-11B as the base model, the performance gains tend to diminish with increasing reference chart complexity, but the improvements remain substantial. This drop might be due to the limited critique capabilities of the LLAMA 3.2-11B base model. Nonetheless, the flexibility of **METAL** to replace the base model for different agents allows us to tailor the system optimally—using, for example, a critique-optimized model for critique agents and a generation-focused model for generation agents—to maximize overall performance.

5.4 Multi-Agent System vs. Modular System

We further investigate the impact of agentic behavior of **METAL** on final performance. We think self-decision-making and code execution abilities are key features that distinguish the multi-agent system from a modular system. We implement a self-revision modular system without these two key abilities, and conduct an additional ablation study on a subset of 50 data points to examine the impact of these agentic behaviors on final performance.

The results show that, compared to **METAL**, there is a 4.51% reduction in average performance gain over direct prompting. The absence of decision-making and code execution abilities in the modular system hinders its capacity to refine generated charts effectively. Specifically, the inability to execute code for chart rendering significantly diminishes the quality of the critique, and the ab-

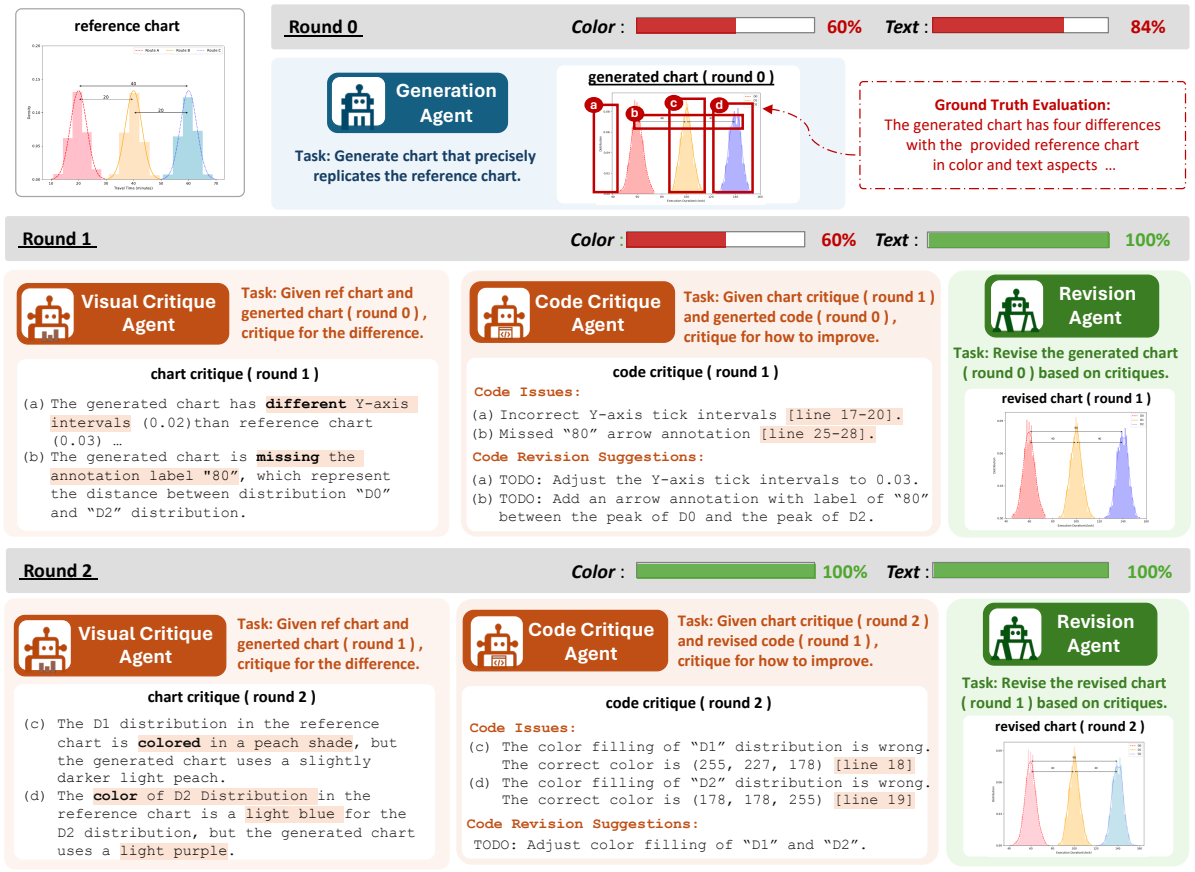

Figure 5: Case study of METAL's progressive refinement from initial generation to perfect accuracy. Starting from Round 0's initial generation (60% color accuracy, 84% text accuracy), the system iteratively improves the output. In Round 1, the system identifies and corrects Y-axis scale issues and missing annotations, achieving 100% text accuracy. Round 2 refines the color representations of distributions, achieving perfect accuracy across all metrics.

sence of self-decision-making ability potentially leads to error propagation that further negatively impacts the self-correction process.

This comparison underscores the critical role of the agentic approach.

6 Case Study

We perform a case study to better understand METAL. Figure 5 illustrates an example.

In Round 1, two specialized critique agents analyze the generated chart. The visual critique agent detects inconsistencies in axis scaling and missing annotations, while the code critique agent identifies the corresponding code-level issues (e.g. incorrect tick intervals and absent annotations). Based on these critiques, the revision agent modifies the chart by adjusting the Y-axis scale and adding the missing annotation. These corrections result in a significant improvement, reaching perfect text accuracy, though color accuracy remains unchanged.

In Round 2, the critique agents further refine the chart. The visual critique agent highlights inaccuracies in the color assignments of distributions, noting that the generated chart does not precisely match the reference chart's colors. The code critique agent pinpoints the exact color discrepancies in the code and provides specific RGB values for correction. The revision agent incorporates these insights, adjusting the color specifications in the code. This final revision achieves perfect alignment with the reference chart, with 100% accuracy across all evaluation metrics.

This case study demonstrates the effectiveness of METAL's multi-agent collaborative refinement process. By decomposing the task into distinct stages, METAL can iteratively enhance the generated output. The separation of visual and code critiques ensures that both perceptual and implementation-level issues are systematically identified and addressed.

7 Conclusion

In conclusion, we introduce METAL, a novel multi-agent framework that significantly enhances

VLMs' performance in the chart generation task. We also reveal two interesting insights from the experiment: the test-time scaling phenomenon in the multi-agent context, and enhanced self-correction with modality-tailored critiques.

Limitation

Our work is not without limitations. First, our **METAL** is based on VLMs, which require extensive prompt engineering. Although we selected the best-performing prompts available, it is possible that even more effective prompts could further enhance our results. Second, automatic evaluations have inherent imperfections and may not capture all details in the chart perfectly. We adopted the evaluation metric from previous work to ensure fairness. Third, **METAL** has higher costs than direct prompting. Future work could explore how to optimize these costs.

References

AI@Meta. 2024. Llama 3 model card.

Anthropic. 2024. Introducing the next generation of claude.

Jinze Bai, Shuai Bai, Shusheng Yang, Shijie Wang, Sinan Tan, Peng Wang, Junyang Lin, Chang Zhou, and Jingren Zhou. 2023. Qwen-vl: A frontier large vision-language model with versatile abilities. *arXiv preprint arXiv:2308.12966*.

Yilun Du and Leslie Kaelbling. 2024. Compositional generative modeling: A single model is not all you need. *arXiv preprint arXiv:2402.01103*.

Yilun Du, Shuang Li, Antonio Torralba, Joshua B Tenenbaum, and Igor Mordatch. 2023. Improving factuality and reasoning in language models through multiagent debate. *arXiv preprint arXiv:2305.14325*.

Google. 2023. Gemini: A family of highly capable multimodal models. *ArXiv*, abs/2312.11805.

Alex Graves. 2012. Sequence transduction with recurrent neural networks. *arXiv preprint arXiv:1211.3711*.

Yucheng Han, Chi Zhang, Xin Chen, Xu Yang, Zhibin Wang, Gang Yu, Bin Fu, and Hanwang Zhang. 2023. Chartllama: A multimodal llm for chart understanding and generation. *arXiv preprint arXiv:2311.16483*.

Fantine Huot, Reinald Kim Amplayo, Jennimaria Palomaki, Alice Shoshana Jakobovits, Elizabeth Clark, and Mirella Lapata. 2024. Agents' room: Narrative generation through multi-step collaboration. *arXiv preprint arXiv:2410.02603*.

Aaron Hurst, Adam Lerer, Adam P Goucher, Adam Perelman, Aditya Ramesh, Aidan Clark, AJ Ostrow, Akila Welihinda, Alan Hayes, Alec Radford, et al. 2024. Gpt-4o system card. *arXiv preprint arXiv:2410.21276*.

Bo Li, Kaichen Zhang, Hao Zhang, Dong Guo, Renrui Zhang, Feng Li, Yuanhan Zhang, Ziwei Liu, and Chunyuan Li. 2024. Llava-next: Stronger llms supercharge multimodal capabilities in the wild.

Haoyu Lu, Wen Liu, Bo Zhang, Bingxuan Wang, Kai Dong, Bo Liu, Jingxiang Sun, Tongzheng Ren, Zhuoshu Li, Yaofeng Sun, Chengqi Deng, Hanwei Xu, Zhenda Xie, and Chong Ruan. 2024. Deepseek-vl: Towards real-world vision-language understanding.

OpenAI. 2023. Gpt-4v(ision) system card.

Xuedi Qin, Yuyu Luo, Nan Tang, and Guoliang Li. 2020. Making data visualization more efficient and effective: a survey. *The VLDB Journal*, 29(1):93–117.

Chufan Shi, Cheng Yang, 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. 2024. Chartmimic: Evaluating lmm's cross-modal reasoning capability via chart-to-code generation. *arXiv preprint arXiv:2406.09961*.

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

Virginia Teller. 2000. Speech and language processing: an introduction to natural language processing, computational linguistics, and speech recognition.

Yifan Wang, Qingyan Guo, Xinzhe Ni, Chufan Shi, Lemao Liu, Haiyun Jiang, and Yujiu Yang. 2024. Hint-enhanced in-context learning wakes large language models up for knowledge-intensive tasks. In *ICASSP 2024-2024 IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP)*, pages 10276–10280. IEEE.

Chengyue Wu, Yixiao Ge, Qiushan Guo, Jiahao Wang, Zhixuan Liang, Zeyu Lu, Ying Shan, and Ping Luo. 2024a. Plot2code: A comprehensive benchmark for evaluating multi-modal large language models in code generation from scientific plots. *arXiv preprint arXiv:2405.07990*.

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

Yijia Xiao, Edward Sun, Di Luo, and Wei Wang. 2024. Tradingagents: Multi-agents llm financial trading framework. *arXiv preprint arXiv:2412.20138*.

Ruyi Xu, Yuan Yao, Zonghao Guo, Junbo Cui, Zanlin Ni, Chunjiang Ge, Tat-Seng Chua, Zhiyuan Liu, and Gao Huang. 2024. LLaVA-UHD: an lmm perceiving any aspect ratio and high-resolution images. *arXiv preprint arXiv:2403.11703*.

Zhengzhuo Xu, Sinan Du, Yiyan Qi, Chengjin Xu, Chun Yuan, and Jian Guo. 2023. Chartbench: A benchmark for complex visual reasoning in charts. *arXiv preprint arXiv:2312.15915*.

Zhiyu Yang, Zihan Zhou, Shuo Wang, Xin Cong, Xu Han, Yukun Yan, Zhenghao Liu, Zhixing Tan, Pengyuan Liu, Dong Yu, et al. 2024. Matplotagent: Method and evaluation for llm-based agentic scientific data visualization. *arXiv preprint arXiv:2402.11453*.

Matei Zaharia, Omar Khattab, Lingjiao Chen, Jared Quincy Davis, Heather Miller, Chris Potts, James Zou, Michael Carbin, Jonathan Frankle, Naveen Rao, and Ali Ghodsi. 2024. The shift from models to compound ai systems. https://bair.berkeley.edu/blog/2024/02/18/compound-ai-systems/.

Kexun Zhang, Shang Zhou, Danqing Wang, William Yang Wang, and Lei Li. 2024. Scaling llm inference with optimized sample compute allocation. *arXiv preprint arXiv:2410.22480*.

Appendix

A Implementation

In this section, we present the detailed implementation of our approach.

A.1 VLMs driven agent Agents

In our implementation, we leverage VLMS to drive agents. The agents are designed to process and generate multimodal information as follows:

Generation Agent and Visual Critique Agent: Both the Generation Agent and the Visual Critique Agent are designed to handle multimodal inputs. Specifically, they take as input a combination of visual data (e.g., the reference chart image or rendered chart) and textual descriptions. These agents are implemented using VLM architectures that can effectively integrate and reason over both image and text modalities. Their outputs are generated in the form of text, which provides either the initial code (in the case of the Generation Agent) or detailed visual discrepancy feedback (in the case of the Visual Critique Agent).

Code Critique Agent and Revision Agent: In contrast, the Code Critique Agent and the Revision Agent are fully text-based. They accept textual inputs—either the generated code or the code accompanied by critique feedback—and produce textual outputs. Both agents are configured to generate responses up to approximately 600 tokens.

Integration of Agents: The agents interact in an iterative pipeline, where the Generation Agent first produces an initial code snippet. The Visual Critique Agent then examines the rendered output for any discrepancies relative to the reference chart, while the Code Critique Agent inspects the code for logical or syntactic issues. Finally, the Revision Agent integrates the feedback from both critique agents to modify the code. We have a Multi-Criteria Verifier (described in Appendix A.2 to verify the output of each iteration.

A.2 Multi-Criteria Verifier

We design three heuristic-based criteria—color, text, and overall—to assess the similarity between two images. The process begins by using EasyOCR to extract text from both the golden and generated images, and then computing a text similarity score based on the Jaccard index of the extracted text sets. In parallel, a verification from the color aspect is performed by converting the images into the HSV color space and applying predefined color ranges to count the pixels corresponding to specific colors; the resulting color histograms are compared using cosine similarity. Finally, an overall similarity measure is obtained by resizing the grayscale versions of the images and calculating the Structural Similarity Index (SSIM). The final verification result is a combination of these three metrics, providing a comprehensive assessment of image equivalence.

During the inference, the iteration will stop if the average of verification results exceeds the predefined threshold. The complete implementation code is attached as follows.

```
from collections import Counter
from sklearn.metrics.pairwise import
    cosine_similarity
from skimage.metrics import
    structural_similarity as ssim
import numpy as np
import cv2
import easyocr

class Verifier():
  def __init__(self, model, **kwargs):
    ocr_model_path = os.environ.get("
    EASYOCR_MODEL_PATH", "./
    easyocr_model")
    self.reader = easyocr.Reader(['en'],
     model_storage_directory=
    ocr_model_path)

  def extract_text(self, image_path):
        results = self.reader.readtext(
    image_path)
        return [text[1] for text in
    results]

  def text_similarity(self, golden_img,
    generated_img):
      text1 = self.extract_text(
    golden_img)
      text2 = self.extract_text(
    generated_img)

      intersection = len(set(text1).
    intersection(set(text2)))
      union = len(set(text1).union(set(
    text2)))
      return intersection / union if
    union > 0 else 0

  def extract_colors(self, image_path,
    top_n=20):
      color_ranges = {
        "red1": [(0, 100, 100), (10,
    255, 255)],  # First red hue range
        "red2": [(170, 100, 100), (180,
    255, 255)],  # Second red hue range
        "green": [(35, 50, 50), (85,
    255, 255)],  # Green hue range
        "blue": [(100, 50, 50), (140,
    255, 255)],  # Blue hue range
        "orange": [(10, 100, 100), (25,
    255, 255)],  # Orange hue range
```

```
        "yellow": [(25, 100, 100), (35,
    255, 255)],  # Yellow hue range
        "cyan": [(85, 100, 100), (100,
    255, 255)],  # Cyan hue range
        "magenta": [(140, 100, 100),
    (170, 255, 255)],  # Magenta hue
    range
        "purple": [(125, 100, 50), (150,
     255, 255)],  # Purple hue range
        "brown": [(10, 50, 20), (30,
    255, 200)],  # Merged Brown & Beige
    range
        "pink": [(150, 100, 100), (170,
    255, 255)],  # Pink hue range
        "light_blue": [(90, 50, 100),
    (110, 255, 255)],  # Light blue
    range
        "dark_blue": [(100, 150, 50),
    (130, 255, 150)],  # Dark blue range
        "dark_green": [(35, 100, 20),
    (85, 255, 120)],  # Dark green hue
    range
        "lime": [(40, 100, 100), (70,
    255, 255)],  # Lime hue range
        "teal": [(80, 100, 100), (100,
    255, 255)],  # Teal hue range
        "olive": [(30, 50, 50), (40,
    255, 150)],  # Olive hue range
        "black_gray": [(0, 0, 0), (180,
    50, 200)],  # Merged Black & Gray
    range
        "white": [(0, 0, 200), (180, 50,
     255)]  # White range
     }

      image = cv2.imread(image_path, cv2
    .IMREAD_UNCHANGED)

      if image.shape[2] == 3:
          image = cv2.cvtColor(image,
    cv2.COLOR_BGR2BGRA)
      hsv_image = cv2.cvtColor(image[:,
    :, :3], cv2.COLOR_BGR2HSV)

      color_counter = Counter()
      for color_name, (lower, upper) in
    color_ranges.items():
          lower = np.array(lower, dtype=
    "uint8")
          upper = np.array(upper, dtype=
    "uint8")
          mask = cv2.inRange(hsv_image,
    lower, upper)
          color_counter[color_name] =
    cv2.countNonZero(mask)

      return color_counter


  def color_similarity(self, golden_img,
     generated_img):
      colors1 = self.extract_colors(
    golden_img)
      colors2 = self.extract_colors(
    generated_img)

      all_colors = set(colors1.keys()).
    union(set(colors2.keys()))
      vec1 = np.array([colors1.get(c, 0)
     for c in all_colors]).reshape(1,
    -1)
      vec2 = np.array([colors2.get(c, 0)
     for c in all_colors]).reshape(1,
    -1)

      return cosine_similarity(vec1,
    vec2)[0, 0]

  def overall_similarity(self,
    golden_img, generated_img):
      img1 = cv2.imread(golden_img, cv2.
    IMREAD_GRAYSCALE)
      img2 = cv2.imread(generated_img,
    cv2.IMREAD_GRAYSCALE)

      img1 = cv2.resize(img1, (300, 300)
    )
      img2 = cv2.resize(img2, (300, 300)
    )

      return ssim(img1, img2)

  def verify(self, golden_img,
    generated_img):

      text_similarity = self.
    text_similarity(golden_img,
    generated_img)
      color_similarity = self.
    color_similarity(golden_img,
    generated_img)
      overall_similarity = self.
    overall_similarity(golden_img,
    generated_img)

      results = {
          "text": text_similarity,
          "color": color_similarity,
          "overall": overall_similarity
      }

      return results
```

B Model Size and Computational Requirement

We have developed two versions of **METAL**, each built upon a different foundational model to cater to varying operational needs.

For the version using the GPT-4o base model, we integrate the model via the OPENAI API. In this setup, each of the four agents makes one API call per action. One single iteration—where each agent acts once—results in 4 API calls in total. In our main experiments, we perform up to 5 iterations per trial.

Alternatively, the LLAMA 3.2-11B-based version of **METAL** is hosted locally on two NVIDIA A100 Tensor Core GPUs, each with 40 GB of GPU memory. Each of the four agents runs its own instance of the LLAMA 3.2-11B model, leading to an overall GPU memory requirement of approximately 70 GB.

C Prompt Templates

C.1 METAL

This section lists all prompt templates used in **METAL**.

```
## Generation Agent ##
# Note: The generation prompt template is adapted from ChartMIMIC.
generation_prompt_template = "You are an expert Python developer who specializes in writing matplotlib code based on a given picture. I found a very nice picture in a STEM paper, but there is no corresponding source code available. I need your help to generate the Python code that can reproduce the picture based on the picture I provide. \nNote that it is necessary to use figsize ={figsize} to set the image size to match the original size. \nNow, please give me the matplotlib code that reproduces the picture below."

## Visual Critique Agent ##
visual_critique_prompt_template = "You are a professional data scientist tasked with evaluating a generated chart against a reference chart.

Objective:
Please identify whether there is (are) issue(s) in the generated chart that diverge from the reference chart concerning the {lowest_metric}, and give concrete feedback.
Compare the original chart (left) and the generated chart (right) in the provided image.
Provide a detailed critique of how the generated chart diverges from the reference chart concerning the {lowest_metric}.
Avoid commenting on unrelated metrics or general stylistic choices unless they directly affect the {lowest_metric}.
Only critique on elements that in reference chart but not in generated chart.
For example, if the reference chart doesn't have a title, don't critique the title in the generated chart.

Instructions:
{instructions}

Response Format:
1. Observation (Reference Chart): Identify the chart elements in the reference chart.
2. Observation (Generated Chart): Identify the chart elements in the generated chart.
3. Critique: Issues in the generated chart that diverge from the reference chart concerning the {lowest_metric}. Be specific and detailed. If there is numeric value or text, please provide the exact value or text.

Note: If you believe there is no issue, please respond with SKIP.
"

## Code Critique Agent ##
code_critique_prompt_template = "You are a professional data scientist tasked with analyzing input code and adding targeted TODO comments based on the provided critique.

Instructions:
Please identify whether there is(are) issue(s) in the input code that need to be addressed based on the visual critique, and give concrete feedback.
Identify the specific issues raised in the critique.
Offer clear and actionable suggestions to address the identified issues.
Insert TODO comments directly into the input code to indicate necessary changes.
Place the TODO comment above the line of code that requires modification.
Be specific and practical in the TODO comments. Avoid generic suggestions or additions unrelated to the critique.
Do not make changes beyond adding TODO comments to the code, such as change existing code or add new lines of code.
Do not modify the code or add comments about code style, unrelated improvements, or hypothetical enhancements outside the critique's scope.
Do not mention reference charts in the TODO comments, making the comment self-contained.

Response Format:
1. Issues: Summarize the issues identified in the critique.
2. Suggestions: Provide specific suggestions for addressing the issues.
3. Full Code with added TODO Comments: Present the input code with the TODO comments added above the relevant lines. Please ONLY add TODO comments to the input code, do not modify the code in any other way.

Critique:
{critique}

Code to Comment On:
```python
{code}
```

Note: If you believe there is no issue, please respond with SKIP.
```

```
"

## Revision Agent ##
code_revision_prompt_template = "You are
    a professional data scientist
    tasked with revising the input code.

Objective:
The input code contains TODO comments
    that need to be addressed.
Please carefully review the code and
    make the necessary revisions to
    address the TODO comments.
Each comment might need more than one
    line of code to address.
Match the other lines of code style and
    structure in the input code.
Ensure that the revised code is correct
    and functional.
Return the FULL revised code to ensure
    the code is ready for the next stage
    of development.

Response Format:
Full Code of the Revised Version:
    Present the revised code with the
    changes made to address the TODO
    comments.

Code to Revise:
```python
{code}
```
"
```

C.2 Variations

This section lists all prompt templates used in variations from the ablation study.

```
# Variations
# 1. Metal-s: GenerationAgent,
    SingleCritiqueAgent, RevisionAgent,
    VerificationAgent
# 2. Metal-v: GenerationAgent,
    VisualCritiqueAgent,
    VisualRevisionAgent,
    VerificationAgent
# 3. Metal-c: GenerationAgent,
    TextCritiqueVisualAgent,
    RevisionAgent, VerificationAgent


## SingleCritiqueAgent (Metal-s) ##
SingleCritiqueAgent_prompts_template = "
    You are a professional data
    scientist tasked to critique the
    generated chart against a reference
    chart to improve the code for
    generating the chart.

Objective:
There is (are) issue(s) in the generated
    chart that diverge from the
    reference chart regarding the {
    lowest_metric}. Compare the original
    chart (left) and the generated
    chart (right) in the provided image.
Observe the differences between the
    reference chart and the generated
    chart, and provide a detailed
    critique of the generated chart.
Insert TODO comments directly into the
    input code to indicate necessary
    changes. Place the TODO comment
    above the line of code that requires
    modification. Be specific and
    practical in the TODO comments.
    Avoid generic suggestions or
    additions unrelated to the critique.

Instructions:
- Step1-2:
Avoid commenting on unrelated metrics or
    general stylistic choices unless
    they directly affect the {
    lowest_metric}.
Only critique on elements that in
    reference chart but not in generated
    chart.
For example, if the reference chart
    doesn't have a title, don't critique
    the title in the generated chart.
Here are insturction for the chart
    critique:
{chart_instructions}
- Step3-4:
If the chart critique specifies
    particular color values, include a
    TODO comment in the code to remove
    the opacity setting (e.g. alpha).
    This ensures the color is accurately
    replaced with the specified values
    from the critique
If the chart critique identifies
    incorrect or missing text (of label,
    annotation, etc.), please include
    the correct text in the TODO comment
    for easy reference.
If the chart critique identifies the
    mismatched type of chart, please add
    TODO comments above the type-
    related functions calls to correct
    the chart type.

Response Format:
1. Chart Critique: Issues in the
    generated chart that diverge from
    the reference chart. Be specific and
    detailed.
2. Code Critique: Provide a critique of
    the code that generated the chart.
    Identify the issues and suggest
    improvements by adding TODO comments
    .
3. Full Code with added TODO Comments:
    Present the input code with the TODO
    comments added above the relevant
    lines. Do not modify the code
    directly.

Code to Comment On:
```python
{code}
```
"

VisualRevisionAgent (Metal-v)
VisualRevisionAgent_prompt_tempate ="
```

```
You are a professional data scientist
    tasked with revising the input code
    based on the visual critique
    provided.

Objective:
The input code contains a Python script
    that generates a chart.
The chart is intended to replicate the
    reference image, but there could be
    discrepancies between the two.
The chart that generated by input code
    has been reviewed by a data
    visualization expert who provided a
    visual critique.
Your task is to revise the code to
    address the issues identified in the
     critique and improve the chart
    accordingly.
Identify the discrepancies and issues in
     the generated chart based on the
    critique.
Make necessary modifications to the
    input code to resolve the identified
     issues and improve the chart.
Match the existing style and structure
    of the input code.
Ensure correctness and functionality.
Return the FULL revised code to ensure
    it is ready for the next stage of
    development.

Inputs:
- Visual Critique:
{visual_critique}

- Code to Revise:
```python
{code}
```
"""

TextCritiqueVisualAgent (Metal-c)
TextCritiqueVisualAgent_prompt_template
    = "You are a professional data
    scientist tasked with analyzing
    input code and adding targeted TODO
    comments based on the reference
    image.

Instructions:
The input code contain python script to
    generate a chart.
The chart generated by the input code
    should replicate the reference image
     provided.

There is(are) issue(s) in the input code
     that need to be addressed to match
    the reference image.
Identify the specific issues raised in
    the input code that prevent it from
    replicating the reference image.
Offer clear and actionable suggestions
    to address the identified issues.
Insert TODO comments directly into the
    input code to indicate necessary
    changes.
Place the TODO comment above the line of
     code that requires modification.
Be specific and practical in the TODO
    comments. Avoid generic suggestions
    or additions unrelated to the
    critique.
Do not make changes beyond adding TODO
    comments to the code, such as change
     existing code or add new lines of
    code.
Do not modify the code or add comments
    about code style, unrelated
    improvements, or hypothetical
    enhancements outside the critique's
    scope.
Do not mention reference charts in the
    TODO comments, making the comment
    self-contained.

Response Format:
1. Issues: Summarize the issues
    identified.
2. Suggestions: Provide specific
    suggestions for addressing the
    issues.
3. Full Code with added TODO Comments:
    Present the input code with the TODO
     comments added above the relevant
    lines. Please ONLY add TODO comments
     to the input code, do not modify
    the code in any other way.

Code to Comment On:
```python
{code}
```
"
```

C.3 Baselines

This section lists all prompt templates used in baselines.

```
## HintEnhanced Baseline ##
# Note: This prompt template is adapted
    from ChartMIMIC.

hint_enhanced_prompt_template = "You are
     an expert Python developer who
    specializes in writing matplotlib
    code based on a given picture. I
    found a very nice picture in a STEM
    paper, but there is no corresponding
     source code available. I need your
    help to generate the Python code
    that can reproduce the picture based
     on the picture I provide.\n\nTo
    ensure accuracy and detail in your
    recreation, begin with a
    comprehensive analysis of the figure
     to develop an elaborate caption.\
    nThis caption should cover, but not
    be limited to, the following aspects
    :\n1. Layout Analysis: e.g.,
    identify the picture's composition,
    noting the presence and arrangement
    of any subplots.\n2. Chart Type
    Identification: e.g., determine how
    many charts within a subplot. Are
    they independent, or do they share a
     common axis?\n3. Data Analysis: e.g
```

```
., summarize the data trend or
pattern.\n4. Additional Features: e.
g.,identify any supplementary
elements such as legends, colormaps,
 tick labels, or text annotations
that contribute to the figure's
clarity or aesthetic appeal.\n\nNow,
 given the picture below, please
first output your comprehensive
caption and then use the caption to
assist yourself to generate
matplotlib code that reproduces the
picture.\nNote that it is necessary
to use figsize={figsize} to set
the image size to match the original
 size."
```