# Automated Visualization Code Synthesis via Multi-Path Reasoning and Feedback-Driven Optimization

Anonymous ACL submission


## Abstract

Rapid advancements in Large Language Models (LLMs) have accelerated their integration into automated visualization code generation applications. Despite advancements through few-shot prompting and query expansion, existing methods remain limited in handling ambiguous and complex queries, thereby requiring manual intervention. To overcome these limitations, we propose *VisPath: a Multi-Path Reasoning and Feedback-Driven Optimization Framework for Visualization Code Generation*. *VisPath* handles underspecified queries through structured, multi-stage processing. It begins by reformulating the user input via *Chain-of-Thought* (CoT) prompting, which refers to the initial query while generating multiple extended queries in parallel, enabling the LLM to capture diverse interpretations of the user intent. These queries then generate candidate visualization scripts, which are executed to produce diverse images. By assessing the visual quality and correctness of each output, *VisPath* generates targeted feedback that is aggregated to select an optimal final result. Extensive experiments on widely-used benchmarks including *MatPlotBench* and the *Qwen-Agent Code Interpreter Benchmark* show that *VisPath* outperforms state-of-the-art methods, offering a more reliable solution for AI-driven visualization code generation.


## 1 Introduction

Data visualization has long been an essential tool in data analysis and scientific research, enabling users to uncover patterns and relationships in complex datasets (Vondrick et al., 2013; Demiralp et al., 2017; Unwin, 2020; Li et al., 2024a). Traditionally, creating visualizations requires manually writing code using libraries such as Matplotlib, Seaborn, or D3.js (Barrett et al., 2005; Bisong and Bisong, 2019; Zhu, 2013). These approaches demand programming expertise and significant effort

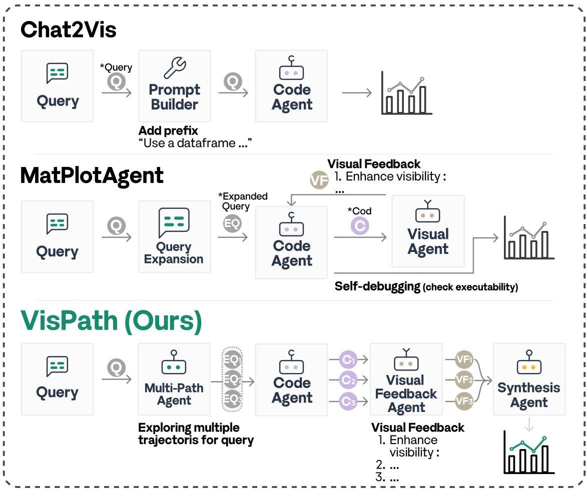



Figure 1: **Overview of different approaches for visualization code generation**. Comparing two baseline methods, namely *Chat2VIS* (Maddigan and Susnjak, 2023) and *MatPlotAgent* (Yang et al., 2024), with our proposed *VisPath* framework.

to craft effective visual representations, which can be a barrier for many users (Bresciani and Eppler, 2015; Saket et al., 2018; Sharif et al., 2024). As datasets continue to grow in size and complexity, researchers have explored ways to automate visualization generation, aiming to make the process more efficient and accessible (Wang et al., 2015; Dibia and Demiralp, 2019; Qian et al., 2021).

In response to this challenge, Large Language Models (LLMs) have emerged as a promising solution for simplifying visualization creation (Wang et al., 2023a; Han et al., 2023; Xie et al., 2024). By translating natural language instructions into executable code, LLM-based systems eliminate the need for extensive programming knowledge, allowing users to generate visualizations more intuitively (Xiao et al., 2023; Ge et al., 2023; Zhang et al., 2024b). Recent visualization methods such as ChartGPT (Tian et al., 2024) and NL4DV (Sah et al., 2024) demonstrate the potential of LLMs to provide interactive, conversational interfaces for visualization. These systems enable users to create complex charts with minimal effort, bridging the gap between technical expertise and effective data exploration (Dibia, 2023; Kim et al., 2024).

More recently, LLM-based visualization frameworks such as Chat2VIS (Maddigan and Susnjak, 2023) and MatPlotAgent (Yang et al., 2024) have been introduced to improve automated visualization code generation. Specifically, Chat2VIS follows a prefix-based approach, guiding LLMs to generate visualization code consistently; and MatPlotAgent expands the query before code generation. However, these methods face several limitations: (1) they generate code in a single-path manner, limiting exploration of alternative solutions and unable to fix-out when caught in misleading bugs; (2) they rely on predefined structures or examples which restrict adaptability to ambiguous or unconventional user queries; (3) they encapsulate limitation in their inability to aggregate and synthesize multi-dimensional feedback. Without a mechanism to retrieve outputs that reflect diverse possibilities, they face difficulties in capturing the intricate details required for visualizations that are both functionally precise and contextually relevant.

To address these limitations, we introduce *VisPath: A Branch Exploration Framework for Visualization Code Synthesis via Multi-Path Reasoning and Feedback-Driven Optimization*, a novel approach that redefines how visualization code is generated, as illustrated in Figure 1. Traditional methods often fall short of delivering the depth and precision users truly need, struggle to capture the intricate details that make a visualization not just functional but meaningful. *VisPath* challenges this limitation by incorporating *Multi-Path Reasoning* and *Feedback-Driven Optimization*, systematically exploring multiple interpretative pathways to construct a more accurate, context-aware, and fully executable visualization.

Rather than simply translating user input into code, *VisPath* ensures that every critical aspect, both explicitly stated and implicitly necessary, is carefully considered, creating a visualization that is not only correct but insightful. At its core, it generates multiple reasoning paths that analyze the user's intent from different perspectives, producing structured blueprints that are then transformed into visualization scripts through Chain-of-Thought (CoT) prompting. These multiple candidates are evaluated using a Vision-Language Model (VLM) to assess accuracy, clarity, and alignment with the intended message. The results are then refined through an Aggregation Module, optimizing the final output for both reliability and impact.

Extensive experiments on benchmark datasets, including *MatPlotBench* (Yang et al., 2024) and *Qwen-Agent Code Interpreter* Benchmark[1], demonstrate that *VisPath* consistently outperforms state-of-the-art methods across all evaluation metrics. In addition, ablation studies confirm that *VisPath*'s performance gains primarily stem from its Multi-Path Reasoning mechanism and feedback-driven optimization. By systematically generating and evaluating multiple reasoning paths and leveraging iterative feedback aggregation, *VisPath* significantly enhances the accuracy, robustness against underspecified queries, and adaptability to diverse user intents. Our investigations demonstrate its ability to capture nuanced user intents, improve execution reliability, and minimize errors, making visualization code generation more accessible and effective for domains such as business intelligence, scientific research, and automated reporting.

## 2 Related Work

Numerous methods have been applied for Text-to-Visualization (Text2Vis) generation, which has significantly evolved over the years, adapting to new paradigms in data visualization and natural language processing (Dibia and Demiralp, 2019; Wu et al., 2022; Chen et al., 2022a,b; Rashid et al., 2022; Zhang et al., 2024a). Early approaches such as Voyager (Wongsuphasawat et al., 2015) and Eviza (Setlur et al., 2016) largely relied on rule-based systems, which mapped textual commands to predefined chart templates or specifications through handcrafted heuristics (de Araújo Lima and Diniz Junqueira Barbosa, 2020). While these methods demonstrated the feasibility of automatically converting text into visualizations (Moritz et al., 2018; Cui et al., 2019), they often required extensive domain knowledge and struggled with more nuanced or ambiguous user requirements (Li et al., 2021; Wang et al., 2023b). Inspired by developments in deep learning, researchers began to incorporate neural networks to handle free-form natural language and broaden the range of supported visualization types (Liu et al., 2021; Luo et al., 2021).

Building on these machine learning strategies, numerous studies have utilized LLMs to further en-

[1]https://github.com/QwenLM/Qwen-Agent/blob/main/benchmark/code_interpreter/README.md

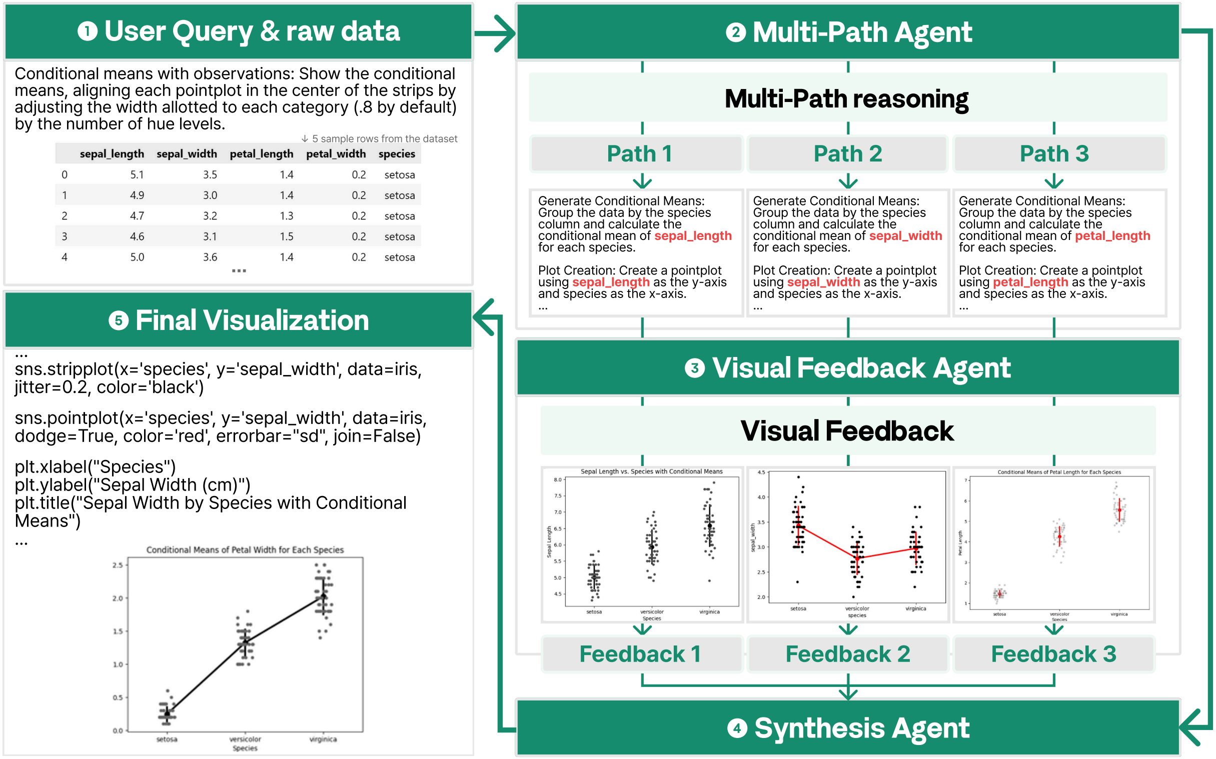



Figure 2: **Overview of the proposed *VisPath* framework for creating robust visualization code generation.** The framework consists of combination of Multi-Path Agent, Visual Feedback Agent, and Synthesis Agent.

hance system flexibility. Recent frameworks such as Chat2VIS (Maddigan and Susnjak, 2023) and Prompt4Vis (Li et al., 2024b) utilize few-shot learning or query expansion to refine user queries, subsequently generating Python visualization scripts through instruction-based prompting. More recent approaches, such as MatPlotAgent (Yang et al., 2024) and PlotGen (Goswami et al., 2025), extend these frameworks by integrating a vision-language feedback model to iteratively optimize the final code based on evaluations of the rendered visualizations. The aforementioned approaches often struggle to effectively capture user intent in complex visualization tasks. By committing to a single reasoning trajectory, they may produce code that is syntactically correct yet semantically misaligned with user expectations, requiring extensive manual adjustments. This challenge is particularly pronounced when user input is ambiguous or underspecified, leading to an iterative cycle of prompt refinement and code modification, consequently limiting the intended efficiency of automation. To address these limitations, we introduce *VisPath*, a novel framework that integrates Multi-Path Reasoning with feedback from VLMs to enhance visualization code generation.

# 3 Methodology

We introduce *VisPath*, a framework for robust visualization code generation that leverages diverse reasoning and visual feedback. *VisPath* is built on three core components: (1) *Multi-Path Query Expansion*, which generates multiple reasoning paths informed by the dataset description; (2) *Code Generation from Expanded Queries*, which synthesizes candidate visualization scripts via Chain-of-Thought (CoT) prompting while grounding them in the actual data context; and (3) *Feedback-Driven Code Optimization*, where a VLM evaluates and refines the outputs to ensure generation robust visualization code. An overview of this process is shown in Figure 2.

## 3.1 Multi-Path Generation

The potential for rigid interpretation is a major limitation in visualization code generation, a single query can have multiple valid representations depending on its dataset structure. *VisPath* mitigates this by generating multiple extended queries within a single interaction. Given a user query $Q$ and a corresponding dataset description $D$, a *Multi-Path Agent* is employed to expand the query into $K$ distinct reasoning pathways:

$$\{R_1, R_2, \ldots, R_K\} = f_{\text{mpa}}(Q, D), \tag{1}$$

where $f_{\text{mpa}}$ denotes the function of the Multi-Path Agent implemented via an LLM. The dataset description $D$ plays a crucial core in shaping these reasoning paths by providing contextual information about variable types, inherent relationships, and the suitability of different chart types for a more grounded interpretation of the query.

Each $R_i$ serves as a detailed logical blueprint outlining one possible approach to fulfill the visualization request. The purpose of generating multiple reasoning paths is not simply to increase their quantity, but to ensure interpretive diversity. *Vispath* mitigates the risk of depending on a single, potentially incorrect assumption by intentionally exploring alternative and context-aware reasoning strategies. This design ensures that our framework effectively considers a broad range of potential interpretations, thereby increasing the quality of reasoning and the likelihood of capturing the true user intent even when queries are ambiguous or underspecified.

### 3.2 Code Generation from Reasoning Paths

Once diverse reasoning paths are established, the subsequent stage involves translating each path into executable Python scripts. For each reasoning path $R_i$ generated in Equation 1, a dedicated Code Generation LLM produces the corresponding visualization code using Chain-of-Thought (CoT) prompting:

$$C_i = f_{\text{code}}(D, R_i), \tag{2}$$

where $f_{\text{code}}$ represents the code generation function. The dataset description $D$ is explicitly provided to ground the generated code in the actual data context, ensuring that variable names, data types, and visualization parameters align correctly with the underlying data attributes. The generated code $C_i$ is then executed to render a candidate visualization:

$$V_i = f_{\text{exec}}(C_i), \quad i = 1, 2, \ldots, K, \tag{3}$$

with $f_{\text{exec}}$ serving as the code execution function. By executing the code directly, we ensure that the visualization accurately reflects the intended operations without reintroducing the dataset description $D$ at this stage, as the code has already been conditioned on $D$ during its generation.

In practice, some generated codes may not be executable. Rather than engaging in an explicit debugging loop, we record the execution status as a binary executability indicator:

$$\epsilon_i = \begin{cases} 1, & \text{if } C_i \text{ is executable} \\ 0, & \text{otherwise.} \end{cases} \tag{4}$$

To route the outputs appropriately, we introduce:

$$Z_i = \begin{cases} \text{plot image}(V_i), & \text{if } \epsilon_i = 1, \\ \text{error message from } C_i, & \text{if } \epsilon_i = 0. \end{cases} \tag{5}$$

The result $Z_i$ (either the rendered visualization or the error message) is then provided, along with $C_i$ and the initial query $Q$, to the feedback model in the subsequent stage.

### 3.3 Feedback-Driven Code Optimization

While most code generation frameworks primarily focus on producing syntactically correct scripts, our framework extends beyond this by incorporating an additional mechanism. As final stage, *VisPath* synthesizes the most robust and accurate visualization code by leveraging both the executability information and structured visual feedback. A VLM is employed to analyze each candidate by evaluating the initial query $Q$, the generated code $C_i$, and the routed output $Z_i$. This evaluation is formalized as:

$$F_i = f_{\text{feedback}}(Q, C_i, Z_i), \tag{6}$$

where $F_i$ provides structured feedback on key aspects such as chart layout, the alignment between the intended request and the rendered visualization (or error context), and visual readability (including potential improvements). To capture the complete quality signal from each candidate, we pair the feedback with its corresponding generated code:

$$S_i = (C_i, F_i), \quad i = 1, 2, \ldots, K. \tag{7}$$

Leveraging the collective code-feedback pairs along with the initial query $Q$ and the dataset description $D$, an *Integration Module* synthesizes the final, refined visualization code:

$$C^* = f_{\text{integrate}}\left(Q, D, \{S_i\}_{i=1}^{K}\right), \tag{8}$$

where $C^*$ represents the optimized visualization code and $f_{\text{integrate}}$ denotes the function that aggregates the strengths of each candidate code alongside its corresponding feedback. This formulation ensures that the final code is not only constructed based on the insights extracted from the candidate outputs but is also meticulously aligned with the original user query and the provided dataset description. An algorithm for *VisPath* is detailed in Algorithm 1 as shown below:

**Algorithm 1** *Algorithm for VisPath*

**Require:** User query $Q$, dataset $D$, number of reasoning paths $K$, Multi-Path Agent $f_{\text{mpa}}$, Code Generation LLM $f_{\text{code}}$, Code Execution Function $f_{\text{exec}}$, Feedback Model $f_{\text{feedback}}$, Integration Module $f_{\text{integrate}}$

*// Step 1: Multi-Path Query Expansion*

1: $\{R_1, R_2, \ldots, R_K\} \leftarrow f_{\text{mpa}}(Q, D)$

*// Step 2: Code Generation from Reasoning Paths*

2: **for** $i = 1$ to $K$ **do**

3: $\quad C_i \leftarrow f_{\text{code}}(D, R_i)$

4: $\quad V_i \leftarrow f_{\text{exec}}(C_i)$

5: $\quad \epsilon_i \leftarrow \begin{cases} 1, & \text{if } C_i \text{ executes successfully} \\ 0, & \text{otherwise} \end{cases}$

6: $\quad Z_i \leftarrow \begin{cases} \text{plot image}(V_i), & \text{if } \epsilon_i = 1 \\ \text{error message from } C_i, & \text{if } \epsilon_i = 0 \end{cases}$

7: **end for**

*// Step 3: Feedback-Driven Code Optimization*

8: **for** $i = 1$ to $K$ **do**

9: $\quad F_i \leftarrow f_{\text{feedback}}(Q, C_i, Z_i)$

10: $\quad S_i \leftarrow (C_i, F_i)$

11: **end for**

12: $C^* \leftarrow f_{\text{integrate}}(Q, D, \{S_1, S_2, \ldots, S_K\})$

13: **return** $C^*$

# 4 Experiments

## 4.1 Setup

In this section, we detail our experimental configuration, including (1) experimental datasets, (2) model specifications, and (3) baseline methods for evaluating the performance of the proposed *VisPath*.

### 4.1.1 Experimental Datasets

We evaluate our approach on two Text-to-Visualization benchmarks: *MatPlotBench* (Yang et al., 2024) and *Qwen-Agent Code Interpreter Benchmark*. Specifically, *MatPlotBench* comprises 100 items with ground truth images; we focus on its simple instruction subset for nuanced queries. In contrast, the *Qwen-Agent Code Interpreter Benchmark* includes 295 records: 163 related to visualization, and evaluates Python code interpreters on tasks such as mathematical problem solving, data visualization, and file handling based on Code Executability and Code Correctness.

### 4.1.2 Models Used

**Large Language Models (LLMs):** For the code inference stage, we experiment with *GPT-4o mini* (Achiam et al., 2023) and *Gemini 2.0 Flash* (Team et al., 2024) to generate candidate visualization code from the reasoning paths. Both models are configured with a temperature of 0.2 to ensure precise and focused outputs, in line with previous work (Yang et al., 2024). To evaluate the generated code quality and guide the subsequent optimization process, we utilize *GPT-4o* (Achiam et al., 2023) and *Gemini 2.0 Flash* (Team et al., 2024) as our visualization feedback model, which provides high-quality reference assessments.

**Vision-Language Models (VLMs):** In order to assess the visual quality and correctness of the rendered plots, we incorporate vision evaluation models into our framework. Specifically, *GPT-4o* (Achiam et al., 2023) is employed for detailed plot evaluation in all evaluation tasks. This setup ensures the thorough evaluation of both the syntactic correctness of the code and the aesthetic quality of the resulting visualizations.

### 4.1.3 Evaluation Metrics

In our experiments, we utilized evaluation metrics introduced by previous work to ensure consistency and comparability. *MatPlotBench* (Bisong and Bisong, 2019) assesses graph generation models using two key metrics: Plot Score, which measures similarity to the Ground Truth (0–100), and Executable Score, which represents the percentage of error-free code executions. *Qwen-Agent Code Interpreter benchmark*[2] evaluates visualization models based on *Visualization-Hard* and *Visualization-Easy*, measuring how well generated images align with queries of different difficulty levels. Compared to *MatPlotBench*, *Qwen-Agent Code Interpreter benchmark* assesses image alignment via a code correctness metric. Previous studies showed that GPT-based VLM evaluations align well with human assessments (Yang et al., 2024), hence VLM was used for evaluation.

### 4.1.4 Baseline Methods

We compare *VisPath* against competitive baselines: (1) *Zero-Shot* directly generates visualization code without intermediate reasoning, (2) *CoT Prompting* uses Chain-of-Thought (CoT) prompting to articulate its reasoning, while (3) *Chat2VIS* (Maddigan and Susnjak, 2023) employs guiding prefixes to mitigate ambiguity, and (4) *MatPlotAgent* (Yang et al., 2024) first expands the query and then refines the code via a self-debugging loop with feedback. Moreover, our proposed framework *VisPath* generates three reasoning paths with corresponding visual feedback to refine the final output.[3] For a

[2] https://github.com/QwenLM/Qwen-Agent/blob/main/benchmark/code_interpreter/README.md

[3] Prompts are detailed in Appendix A.

| Model | Methods | MatPlotBench | | Qwen-Agent Code Interpreter benchmark | | |
|---|---|---|---|---|---|---|
| | | Plot Score | Executable Rate (%) | Visualization-Hard | Visualization-Easy | Avg. |
| **GPT-4o mini** | Zero-Shot | 62.38 | 53 | 59.68 | 45.50 | 52.59 |
| | CoT Prompting (Wei et al., 2022) | 61.95 | 50 | 57.50 | 40.00 | 48.75 |
| | Chat2VIS (Maddigan and Susnjak, 2023) | 56.98 | 53 | 59.36 | 36.50 | 47.93 |
| | MatPlotAgent (Yang et al., 2024) | 63.90 | 58 | 67.50 | 53.25 | 60.38 |
| | VisPath† (Ours) | **66.12** | **60** | **70.68** | **57.23** | **63.96** |
| **Gemini 2.0 Flash** | Zero-Shot | 55.00 | 54 | 68.97 | 52.18 | 60.58 |
| | CoT Prompting (Wei et al., 2022) | 53.56 | 61 | 40.00 | **63.89** | 51.95 |
| | Chat2VIS (Maddigan and Susnjak, 2023) | 54.89 | 55 | 59.36 | 56.50 | 57.93 |
| | MatPlotAgent (Yang et al., 2024) | 56.31 | 58 | 77.62 | 51.50 | 64.56 |
| | VisPath† (Ours) | **59.37** | **63** | **80.79** | 57.17 | **68.98** |

Table 1: **Performance comparison of various methods across different benchmarks.** Zero-Shot refers to directly generating code. CoT Prompting utilizes Chain of Thought Prompting. Visualization-Hard and Visualization-Easy refer to the Accuracy of Code Execution Results on different subsets of the *Qwen-Agent Code Interpreter benchmark*. **Bold text** indicates the best performance, underlined text indicates the second-best performance. † denotes our proposed method.

fair comparison aligned with our experimental setting, *MatPlotAgent* is limited to three iterations, and uses critique-based debugging loop as well.

### 4.2 Experimental Analysis

*VisPath* is evaluated against four baselines: Zero Shot prompting, CoT prompting, *Chat2VIS*, and *MatPlotAgent*. The evaluation is conducted on *MatPlotBench* and the *Qwen-Agent Code Interpreter benchmark* using GPT-4o mini and Gemini 2.0 Flash, as shown in Table 1.

Zero-Shot prompting generates visualization code directly from natural language queries without intermediate reasoning. While computationally efficient, it often struggles to handle ambiguity or under-specification, resulting in misaligned or incomplete outputs. On *MatPlotBench (GPT-4o mini)*, it achieves a Plot Score of 62.38 and an Executable Rate of 53%. CoT prompting further introduces a single reasoning step to expose intermediate decisions and improve interpretability. However, on *MatPlotBench*, it slightly underperforms Zero-Shot in both Plot Score and Executable Rate, indicating reliance on a fixed reasoning path may reduce adaptability to diverse input structures.

*Chat2VIS* extends CoT prompting by adopting prefix templates to improve coherence and reduce ambiguity in user instructions. While this approach is effective for well-structured or common query formats, its dependence on fixed templates limits adaptability when processing loosely specified or novel queries. Such limitation is evident in its performance on *MatPlotBench*, where it achieves a Plot Score of 56.98 and an Executable Rate of 53%. Furthermore, *MatPlotAgent* incorporates query expansion and iterative self-debugging mechanisms to enhance robustness. While effective at correcting execution-level errors, its revisions are confined to localized adjustments and do not address higher-order semantic ambiguities.

In contrast, our proposed framework *VisPath* is specifically designed to overcome these limitations observed in prior methods by dynamically generating multiple reasoning paths and refining them through structured visual feedback. In particular, template-based approaches such as *Chat2VIS* offer limited adaptability due to their reliance on predefined input formats, while methods such as *MatPlotAgent* focus on localized corrections without addressing broader semantic ambiguity. Unlike prior methods, *VisPath* generates diverse interpretations of user intent and evaluates them holistically using structured vision-language feedback. This enables more flexible handling of under-specified or ambiguous inputs, resulting in semantically aligned and executable visualizations.

Evaluated across multiple benchmark settings, *VisPath* notably outperforms baselines, achieving up to 9.14 point gains in Plot Score and a 10% point increase in Executable Rate. These improvements well demonstrate *VisPath*'s robustness in exploring diverse reasoning paths and iteratively refine outputs through structured visual feedback, effectively reducing semantic ambiguity and improving execution reliability.

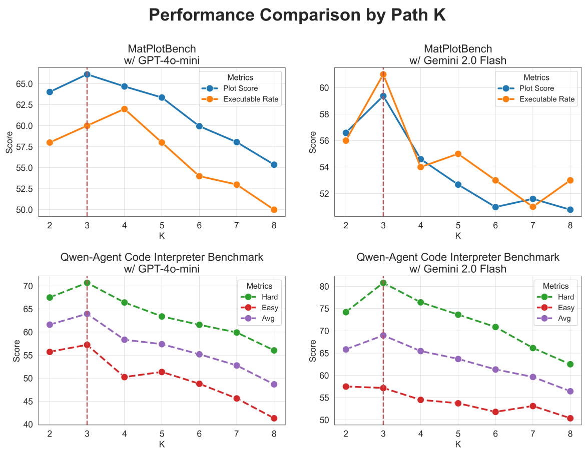



Figure 3: **Effect of varying the number of reasoning paths $K$ on performance across datasets and models.** Metrics include Plot Score, Executable Rate. The results show that $K = 3$ achieves the best overall balance, with larger $K$ values reducing performance.

### 4.3 Ablation Study

To further examine the robustness and design choices of *VisPath*, we conduct a series of ablation experiments in this section. Specifically, we analyze the following three aspects: (i) varying the number of generated reasoning paths, (ii) the effect of removing visual feedback during integration, and (iii) the contribution of visual feedback beyond binary executability.

#### 4.3.1 Varying the Number of Reasoning Paths

To investigate the contribution of reasoning path diversity, we conducted ablation experiments by varying the number of generated reasoning paths $K$. In particular, we extended the range of $K$ from 2 to 8 to examine the effect of increased path multiplicity on the overall performance of *VisPath*, as shown in Figure 3.

We observe a consistent pattern across all model and dataset combinations: performance improves as $K$ increases from 2 to 3, confirming that limited diversity ($K = 2$) often fails to capture nuanced interpretations of user queries. For example, with GPT-4o mini on *MatPlotBench*, the Plot Score improves from 64.02 to 66.12 (+2.10), and the Executable Rate improves from 58% to 60% (+2 points). On Gemini 2.0 Flash, the Plot Score increases from 56.59 to 59.37 (+2.78), and the Executable Rate from 56% to 61% (+5 points).

While $K = 4$ achieves the highest executable rate on *MatPlotBench* with GPT-4o mini (62%), we further extend our analysis up to $K = 8$ to comprehensively assess the impact of reasoning path diversity. However, beyond $K = 4$, we observe diminishing returns and even performance degradation, which is likely due to noisy or redundant reasoning paths. While added diversity initially aids interpretation, excessive expansion burdens the integration process and reduces overall efficiency.

Among all configurations tested up to $K = 8$, $K = 3$ emerges as the most balanced choice, offering substantial performance gains in both the Executable Rate and the Plot Score while avoiding the inefficiencies observed at higher values of $K$. Hence, we adopt $K = 3$ as the default configuration throughout our experiments.

#### 4.3.2 Robustness with a Simple Integration

We evaluate an alternative integration strategy that simplifies the aggregation of multiple reasoning paths to further validate the robustness of *VisPath*.

| Model | Feedback | MatPlotBench | | Qwen-Agent |
|---|---|---|---|---|
| | | Plot Score | Executable Rate (%) | Avg. |
| GPT-4o mini | (w/o) feedback | 63.76 | 56 | 58.00 |
| | **(w) feedback** | **66.12** | **60** | **63.96** |
| Gemini 2.0 Flash | (w/o) feedback | 55.28 | 57 | 64.03 |
| | **(w) feedback** | **59.37** | **63** | **68.98** |

Table 2: **Performance comparison of *VisPath* with and without visual feedback**. The MatPlotBench scores (Plot Score and Executable Rate) and the average score from the Qwen-Agent Code Interpreter benchmark are shown for two LLMs.

Instead of refining each candidate visualization with feedback from vision-language feedback, this approach aggregates three candidate codes: each derived from a distinct reasoning path, without intermediate corrections. This setup reduces computational overhead and execution time while preserving the benefits of interpretive diversity.

As shown in Table 2, even under this simplified configuration, *VisPath* outperforms all baseline methods, confirming that Multi-Path Reasoning alone offers a strong foundation for visualization code generation. While full feedback-driven optimization leads to additional performance improvements, this result highlights that the primary strength of *VisPath* lies in its capacity to explore and leverage diverse reasoning trajectories. The framework remains effective and adaptable, even with minimal refinements, further validating importance of its core design.

| LLM | Variant | Plot Score | Executable Rate (%) |
|---|---|---|---|
| GPT-4o mini | (w) feedback | **66.12** | **60** |
| | (w) binary feedback | 64.82 | 58 |
| Gemini 2.0 Flash | (w) feedback | **59.37** | **63** |
| | (w) binary feedback | 57.68 | 59 |

Table 3: **Ablation results isolating the impact of rendered visual feedback.** Comparing the full *VisPath* model using structured plot-based feedback with a binary-only feedback variant. Visual feedback provides consistent gains in both Plot Score and Executable Rate across LLMs.

### 4.3.3 Distinct Contribution of Visual Feedback

To assess the role of visual feedback in improving code quality, we compare two variants of our framework. The first, *VisPath (w/ feedback)*, uses a VLM to evaluate both rendered plots and error messages. The second, *VisPath Execute (w/ binary feedback)*, simplifies evaluation by relying solely on the binary success or failure of code execution.

Incorporating rendered visual feedback improves the Executable Rate by $2\% - 4\%$ and consistently boosts the Plot Score across both LLMs, as shown in Table 3. On GPT-4o mini, Plot Score increases from 64.82 to 66.12 (+1.30) and Executable Rate from 58% to 60% (+2 points). On Gemini 2.0 Flash, Plot Score rises from 57.68 to 59.37 (+1.69), and Executable Rate from 59% to 63% (+4 points). Despite the numerical gains are modest, the results demonstrate the unique value of structured visual evaluation. Visual feedback enables more refined and user-aligned outputs by capturing subtle rendering issues that may not affect executability, demonstrating its importance in the final synthesis stage.

## 5 Discussion

Our proposed *VisPath* framework substantially advances visualization code generation by addressing the core weaknesses of existing methods: limited interpretive flexibility and insufficient refinement. By employing Multi-Path Reasoning, *VisPath* explores diverse interpretations of user intent, which leads to more accurate visualizations, especially for ambiguous queries. Experimental results confirm its superiority: *VisPath* outperforms all baselines on both *MatPlotBench* and the *Qwen-Agent Code Interpreter benchmark*, with up to 9.14% improvement in Plot Score and 10% in Executable Rate. Ablation studies further validate *VisPath*'s design. First, increasing the number of reasoning paths enhances both visual quality and code executability. Second, even without visual feedback, Multi-Path Reasoning alone proves highly effective. Third, using structured plot-based feedback, rather than binary execution signals, significantly improves output alignment with user intent, confirming the value of our feedback-driven optimization loop.

The figures provided in Appendix clearly illustrate *VisPath*'s robustness.[4] In Case 1, it correctly centers conditional means using adjusted strip widths, unlike other methods that misalign pointplots or ignore key parameters. In Case 2, *VisPath* generates a proper polar bar chart with radial coordinates and legible labels, while others either fail to render or produce standard rectangular plots, with text often overlapping or unreadable. Case 3 highlights *VisPath*'s ability to handle compositional visualization requests: it correctly interprets the instruction to "visualize in 3 different ways" by generating a multi-subplot layout that includes line, scatter, and bar plots. Other approaches, by contrast, either collapsed all time series into a single chaotic plot or failed to differentiate the visualization modalities at all. These examples demonstrate how *VisPath*'s Multi-Path Reasoning and visual feedback lead to more precise and semantically aligned visualizations than current alternatives.

## 6 Conclusion

In this work, we present *VisPath*, a framework that leverages Multi-Path Reasoning and feedback-driven optimization to enhance automated visualization code generation. Unlike prior methods, our approach seamlessly combines Multi-Path Reasoning with feedback-driven optimization. Accurately capturing diverse user intents and refining generated code, *VisPath* achieves notable improvements in both execution success and visual quality on challenging benchmarks such as *MatPlotBench* and the *Qwen-Agent Code Interpreter Benchmark*. By prioritizing adaptability, *VisPath* is uniquely positioned to handle ambiguous user queries through a combination of diverse reasoning paths and visual feedback integration. Future work could explore *VisPath*'s adaptability in more dynamic, real-world scenarios, further broadening its scope and practical utility in complex data analysis contexts.

[4] The detailed cases are provided in Appendix B.

## 7 Limitations

Despite its effectiveness, the current framework relies on a limited feedback mechanism focused on query-code and query-plot alignment. While informative, these signals may overlook finer-grained elements essential to interpretability. Thus, future work could improve feedback depth by assessing individual plot components, such as readability and visual coherence, enabling more precise and refined visualization code generation. Moreover, while achieving strong performance, *VisPath* requires several rounds of agent interaction, including multi-path reasoning, execution, and feedback integration, which may introduce inefficiencies in certain use cases. Future work could explore ways to selectively identify the most promising reasoning paths early in the process, reducing redundant computation while preserving the benefits of diverse interpretation.

## References


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

Paul Barrett, John Hunter, J Todd Miller, J-C Hsu, and Perry Greenfield. 2005. matplotlib–a portable python plotting package. In *Astronomical data analysis software and systems XIV*, volume 347, page 91.

Ekaba Bisong and Ekaba Bisong. 2019. Matplotlib and seaborn. *Building machine learning and deep learning models on google cloud platform: A comprehensive guide for beginners*, pages 151–165.

Sabrina Bresciani and Martin J Eppler. 2015. The pitfalls of visual representations: A review and classification of common errors made while designing and interpreting visualizations. *Sage Open*, 5(4):2158244015611451.

Qiaochu Chen, Shankara Pailoor, Celeste Barnaby, Abby Criswell, Chenglong Wang, Greg Durrett, and Işil Dillig. 2022a. Type-directed synthesis of visualizations from natural language queries. *Proceedings of the ACM on Programming Languages*, 6(OOPSLA2):532–559.

Yiru Chen, Ryan Li, Austin Mac, Tianbao Xie, Tao Yu, and Eugene Wu. 2022b. Nl2interface: Interactive visualization interface generation from natural language queries. *arXiv preprint arXiv:2209.08834*.

Weiwei Cui, Xiaoyu Zhang, Yun Wang, He Huang, Bei Chen, Lei Fang, Haidong Zhang, Jian-Guan Lou, and Dongmei Zhang. 2019. Text-to-viz: Automatic generation of infographics from proportion-related natural language statements. *IEEE transactions on visualization and computer graphics*, 26(1):906–916.

Raul de Araújo Lima and Simone Diniz Junqueira Barbosa. 2020. Vismaker: a question-oriented visualization recommender system for data exploration. *arXiv e-prints*, pages arXiv–2002.

Çağatay Demiralp, Peter J Haas, Srinivasan Parthasarathy, and Tejaswini Pedapati. 2017. Foresight: Recommending visual insights. *arXiv preprint arXiv:1707.03877*.

Victor Dibia. 2023. Lida: A tool for automatic generation of grammar-agnostic visualizations and infographics using large language models. *arXiv preprint arXiv:2303.02927*.

Victor Dibia and Çağatay Demiralp. 2019. Data2vis: Automatic generation of data visualizations using sequence-to-sequence recurrent neural networks. *IEEE computer graphics and applications*, 39(5):33–46.

Yan Ge, Victor Junqiu Wei, Yuanfeng Song, Jason Chen Zhang, and Raymond Chi-Wing Wong. 2023. Automatic data visualization generation from chinese natural language questions. *arXiv preprint arXiv:2309.07650*.

Kanika Goswami, Puneet Mathur, Ryan Rossi, and Franck Dernoncourt. 2025. PlotGen: Multi-Agent LLM-based Scientific Data Visualization via Multimodal Feedback. *Preprint*, arXiv:2502.00988.

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*.

Jaeyoung Kim, Sihyeon Lee, Hyeon Jeon, Keon-Joo Lee, Hee-Joon Bae, Bohyoung Kim, and Jinwook Seo. 2024. Phenoflow: A human-llm driven visual analytics system for exploring large and complex stroke datasets. *IEEE Transactions on Visualization and Computer Graphics*.

Guozheng Li, Xinyu Wang, Gerile Aodeng, Shunyuan Zheng, Yu Zhang, Chuangxin Ou, Song Wang, and Chi Harold Liu. 2024a. Visualization generation with large language models: An evaluation. *arXiv preprint arXiv:2401.11255*.

Haotian Li, Yong Wang, Songheng Zhang, Yangqiu Song, and Huamin Qu. 2021. Kg4vis: A knowledge graph-based approach for visualization recommendation. *IEEE Transactions on Visualization and Computer Graphics*, 28(1):195–205.

Shuaimin Li, Xuanang Chen, Yuanfeng Song, Yunze Song, and Chen Zhang. 2024b. Prompt4Vis: Prompting Large Language Models with Example Mining and Schema Filtering for Tabular Data Visualization. *Preprint*, arXiv:2402.07909.

Can Liu, Yun Han, Ruike Jiang, and Xiaoru Yuan. 2021. Advisor: Automatic visualization answer for natural-language question on tabular data. In *2021 IEEE 14th Pacific Visualization Symposium (PacificVis)*, pages 11–20. IEEE.

Yuyu Luo, Nan Tang, Guoliang Li, Jiawei Tang, Chengliang Chai, and Xuedi Qin. 2021. Natural language to visualization by neural machine translation. *IEEE Transactions on Visualization and Computer Graphics*, 28(1):217–226.

Paula Maddigan and Teo Susnjak. 2023. Chat2vis: Fine-tuning data visualisations using multilingual natural language text and pre-trained large language models. *arXiv preprint arXiv:2303.14292*.

Dominik Moritz, Chenglong Wang, Greg L Nelson, Halden Lin, Adam M Smith, Bill Howe, and Jeffrey Heer. 2018. Formalizing visualization design knowledge as constraints: Actionable and extensible models in draco. *IEEE transactions on visualization and computer graphics*, 25(1):438–448.

Xin Qian, Ryan A Rossi, Fan Du, Sungchul Kim, Eunyee Koh, Sana Malik, Tak Yeon Lee, and Joel Chan. 2021. Learning to recommend visualizations from data. In *Proceedings of the 27th ACM SIGKDD conference on knowledge discovery & data mining*, pages 1359–1369.

Md Mahinur Rashid, Hasin Kawsar Jahan, Annysha Huzzat, Riyasaat Ahmed Rahul, Tamim Bin Zakir, Farhana Meem, Md Saddam Hossain Mukta, and Swakkhar Shatabda. 2022. Text2chart: A multi-staged chart generator from natural language text. In *Pacific-Asia Conference on Knowledge Discovery and Data Mining*, pages 3–16. Springer.

Subham Sah, Rishab Mitra, Arpit Narechania, Alex Endert, John Stasko, and Wenwen Dou. 2024. Generating analytic specifications for data visualization from natural language queries using large language models. *arXiv preprint arXiv:2408.13391*.

Bahador Saket, Alex Endert, and Çağatay Demiralp. 2018. Task-based effectiveness of basic visualizations. *IEEE transactions on visualization and computer graphics*, 25(7):2505–2512.

Vidya Setlur, Sarah E Battersby, Melanie Tory, Rich Gossweiler, and Angel X Chang. 2016. Eviza: A natural language interface for visual analysis. In *Proceedings of the 29th annual symposium on user interface software and technology*, pages 365–377.

Ather Sharif, Joo Gyeong Kim, Jessie Zijia Xu, and Jacob O Wobbrock. 2024. Understanding and reducing the challenges faced by creators of accessible online data visualizations. In *Proceedings of the 26th International ACM SIGACCESS Conference on Computers and Accessibility*, pages 1–20.

Gemini Team, Petko Georgiev, Ving Ian Lei, Ryan Burnell, Libin Bai, Anmol Gulati, Garrett Tanzer, Damien Vincent, Zhufeng Pan, Shibo Wang, et al. 2024. Gemini 1.5: Unlocking multimodal understanding across millions of tokens of context. *arXiv preprint arXiv:2403.05530*.

Yuan Tian, Weiwei Cui, Dazhen Deng, Xinjing Yi, Yurun Yang, Haidong Zhang, and Yingcai Wu. 2024. Chartgpt: Leveraging llms to generate charts from abstract natural language. *IEEE Transactions on Visualization and Computer Graphics*.

Antony Unwin. 2020. Why is data visualization important? what is important in data visualization? *Harvard Data Science Review*, 2(1):1.

Carl Vondrick, Aditya Khosla, Tomasz Malisiewicz, and Antonio Torralba. 2013. Hoggles: Visualizing object detection features. In *Proceedings of the IEEE International Conference on Computer Vision*, pages 1–8.

Chenglong Wang, John Thompson, and Bongshin Lee. 2023a. Data formulator: Ai-powered concept-driven visualization authoring. *IEEE Transactions on Visualization and Computer Graphics*.

Lei Wang, Songheng Zhang, Yun Wang, Ee-Peng Lim, and Yong Wang. 2023b. Llm4vis: Explainable visualization recommendation using chatgpt. *arXiv preprint arXiv:2310.07652*.

Lidong Wang, Guanghui Wang, and Cheryl Ann Alexander. 2015. Big data and visualization: methods, challenges and technology progress. *Digital Technologies*, 1(1):33–38.

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

Kanit Wongsuphasawat, Dominik Moritz, Anushka Anand, Jock Mackinlay, Bill Howe, and Jeffrey Heer. 2015. Voyager: Exploratory analysis via faceted browsing of visualization recommendations. *IEEE transactions on visualization and computer graphics*, 22(1):649–658.

Chenfei Wu, Jian Liang, Lei Ji, Fan Yang, Yuejian Fang, Daxin Jiang, and Nan Duan. 2022. Nüwa: Visual synthesis pre-training for neural visual world creation. In *European conference on computer vision*, pages 720–736. Springer.

Shishi Xiao, Suizi Huang, Yue Lin, Yilin Ye, and Wei Zeng. 2023. Let the chart spark: Embedding semantic context into chart with text-to-image generative model. *IEEE Transactions on Visualization and Computer Graphics*.

Yupeng Xie, Yuyu Luo, Guoliang Li, and Nan Tang. 2024. Haichart: Human and ai paired visualization system. *arXiv preprint arXiv:2406.11033*.

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*.

Songheng Zhang, Lei Wang, Toby Jia-Jun Li, Qiaomu Shen, Yixin Cao, and Yong Wang. 2024a. Chartifytext: Automated chart generation from data-involved texts via llm. *arXiv preprint arXiv:2410.14331*.

Zhehao Zhang, Weicheng Ma, and Soroush Vosoughi. 2024b. Is gpt-4v (ision) all you need for automating academic data visualization? exploring vision-language models' capability in reproducing academic charts. In *Findings of the Association for Computational Linguistics: EMNLP 2024*, pages 8271–8288.

Nick Qi Zhu. 2013. *Data visualization with D3. js cookbook*. Packt Publishing Ltd.

# A Appendix A. Prompts Used

**Prompt for *Multi-Path Reasoning***

[**System Prompt**] According to the user query, expand and solidify the query into detailed instruction on how to write python code to fulfill the user query's requirements. Import the appropriate libraries. Pinpoint the correct library functions to call and set each parameter in every function call accordingly.
[**User Prompt**] Think step by step. Generate three distinct extended queries based on the given query. Ensure that you first analyze the given data description and create queries that align with the data. If no data description is provided, follow the initial query as is. You must follow the Python list format for the output. Do not modify the detailed instructions from the original user query. initial query: ori_query
Data description: data_description
Output format: [extended_path_1, extended_path_2, extended_path_3]

**Prompt for *Code Generation***

[**System Prompt**] You are an expert in data visualization code generation. Think step by step and write the generated code in the format “‘python...“‘, where "..." represents the generated code. The code must end with ‘plt.show()‘.
[**User Prompt**] Think step by step. Based on the user's query and the provided data description, generate Python code using ‘matplotlib.pyplot‘ and ‘seaborn‘ to create the requested plot. Ensure that the code is formatted as “‘...“‘, where "..." represents the generated code.
User query: {query}
Data description: {data_description}

### Prompt for *Visual Feedback*

[**System Prompt**] Given a code, a user query, and an image of the current plot, please determine whether the plot accurately follows the user query. Provide detailed instructions on how to enhance the plot using Python code.
[**User Prompt**] Carefully analyze the provided Python code, the user query, and the plot image (if available) to assess whether the generated plot meets the user query requirements. If the plot image is missing, check the error message that occurred in the code. Compare the plot with the user query, highlight discrepancies, and provide clear, actionable instructions to modify the code. Additionally, suggest improvements for better visualization, focusing on clarity, readability, and alignment with the user's objectives.
Code: {code}
User query: {ori_query}

### Prompt for *Synthesis*

[**System Prompt**] You are an expert on data visualization code judgement and aggregation.
[**User Prompt**] Think step by step. Given the provided user query, data description, multiple data visualization codes generated for the query, and feedback for each code's generated image. Your task is to:
1. Carefully review the user query and the data description.
2. Examine each version of the data visualization code along with the feedback provided for each version.
3. Synthesize the feedbacks for each code, user query insights, data description to create a final version of the code.
4. Your goal is to produce a final version of code that more effectively fulfills the user query by integrating the best elements from all versions and applying necessary corrections.
User Query: {ori_query}
Data Description: {data_description}
Code for aggregation with corresponding feedback: {code_for_aggregation}

### Prompt for Evaluation: *MatplotBench*

You are an excellent judge at evaluating visualization plots between a model generated plot and the ground truth. You will be giving scores on how well it matches the ground truth plot.
The generated plot will be given to you as the first figure. If the first figure is blank, that means the code failed to generate a figure. Another plot will be given to you as the second figure, which is the desired outcome of the user query, meaning it is the ground truth for you to reference. Please compare the two figures head to head and rate them. Suppose the second figure has a score of 100, rate the first figure on a scale from 0 to 100. Scoring should be carried out in the following aspect:
1. Plot correctness:
Compare closely between the generated plot and the ground truth, the more resemblance the generated plot has compared to the ground truth, the higher the score. The score should be proportionate to the resemblance between the two plots. In some rare occurrence, see if the data points are generated randomly according to the query, if so, the generated plot may not perfectly match the ground truth, but it is correct nonetheless.
Only rate the first figure, the second figure is only for reference.
If the first figure is blank, that means the code failed to generate a figure.
Give a score of 0 on the Plot correctness.
After scoring from the above aspect, please give a final score. The final score is preceded by the [FINAL SCORE] token.
For example [FINAL SCORE]: 40.

### Prompt for Evaluation: *Qwen-Agent Code Interpreter benchmark*

Please judge whether the image is consistent with the [Question] below, if it is consistent then reply "right", if not then reply "wrong".
Question: {query}

# B Appendix B. Case Study

## Case 1

### User Query

Conditional means with observations: **Show the conditional means, aligning each pointplot in the center of the strips by adjusting the width allotted to each category** (.8 by default) by the number of hue levels.

### Raw Data (First 5 Rows)

| | sepal_length | sepal_width | petal_length | petal_width | species |
|---|---|---|---|---|---|
| 0 | 5.1 | 3.5 | 1.4 | 0.2 | setosa |
| 1 | 4.9 | 3.0 | 1.4 | 0.2 | setosa |
| 2 | 4.7 | 3.2 | 1.3 | 0.2 | setosa |
| 3 | 4.6 | 3.1 | 1.5 | 0.2 | setosa |
| 4 | 5.0 | 3.6 | 1.4 | 0.2 | setosa |

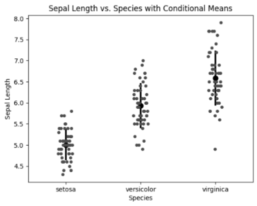
### Multi-Path Reasoning

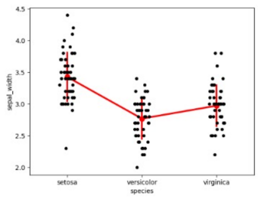
Generate Conditional Means: Group the data by the species column and calculate the conditional mean of sepal_length for each species. Plot Creation**: Create a pointplot using sepal_length as the y-axis and species as the x-axis. Align the pointplot in the center of the strips by adjusting the width allotted to each category**
...

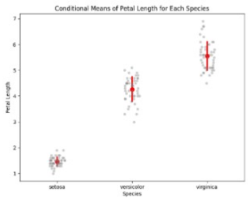
Generate Conditional Means: Group the data by the species column and calculate the conditional mean of sepal_width for each species. Plot Creation**: Create a pointplot using sepal_width as the y-axis and species as the x-axis. Align the pointplot in the center of the strips by adjusting the width allotted to each category** (default width of .8 adjusted by the number of hue levels).
...

Generate Conditional Means: Group the data by the species column and calculate the conditional mean of petal_length for each species.
Plot Creation**: Create a pointplot using petal_length as the y-axis and species as the x-axis.**
Align the pointplot in the center of the strips by adjusting the width allotted to each category
...

### The Middle Generated Figures




### Visual Feedback

...
Centering Point Plots: The primary goal is to center the point plots within each group of strips. The current code uses dodge=True in sns.pointplot, **which is intended for use with a hue variable to separate points within each category. Since there's no hue here, dodge=True isn't helping.**
...

...
The current plot displays both strip plots and point plots, which is good. The point plots are overlaid on the strip plots, **but they are not precisely centered within each category's strip plot distribution.** The user specifically mentions adjusting the width allotted to each category.
...

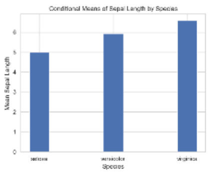
...
Adjust pointplot position: To align the pointplot markers with the center of the strip plot observations, use the dodge parameter in sns.pointplot. **The user query specifies that the width allotted to each category should be adjusted by the number of hue levels, but since there's no hue specified,**
...

### Final Code

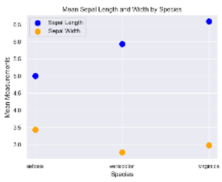
```
...
sns.stripplot(x='species', y='sepal_width', data=iris, jitter=0.2, color='black')
sns.pointplot(x='species', y='sepal_width', data=iris, dodge=True, color='red', errorbar="sd", join=False)
plt.xlabel("Species")
plt.ylabel("Sepal Width (cm)")
plt.title("Sepal Width by Species with Conditional Means")
...
```

### Final Figure

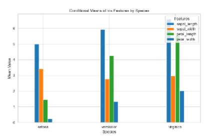
**Zero Shot**



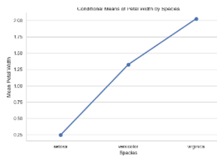
**CoT**



**Chat2VIS**



**MatPlotAgent**



**VisPath**

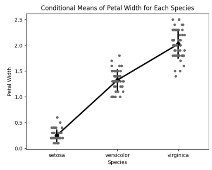



Figure 4: **Overview of the end-to-end process of visualizing conditional means alignment.**

## Case 2

### User Query

**Create a bar chart on a polar axis**

### Raw Data (First 5 Rows)

| | theta | radii | width |
|---|---|---|---|
| 0 | 0.000000 | 7.003673 | 0.511723 |
| 1 | 0.314159 | 7.427508 | 0.311652 |
| 2 | 0.628319 | 7.092800 | 0.426659 |
| 3 | 0.942478 | 5.667455 | 0.628240 |
| 4 | 1.256637 | 9.777853 | 0.566700 |

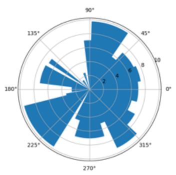
### Multi-Path Reasoning

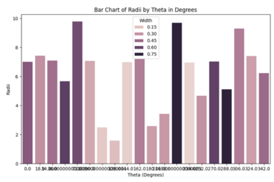
**To create a bar chart on a polar axis using Matplotlib,** we will follow these steps: 1. Import the necessary libraries: pandas for data manipulation and matplotlib for plotting. 2. Load the data from 'data.csv' using pandas. **3. Extract the 'theta', 'radii', and 'width' columns from the DataFrame. 4. Use the 'bar' function from Matplotlib to create a bar chart on a polar axis.**
...

**In this methodology, we will utilize Seaborn for enhanced visualization aesthetics.** Steps: 1. Import pandas for data handling and seaborn for plotting. 2. Load the data from 'data.csv' into a DataFrame. 3. Create a new column in the DataFrame that converts 'theta' to degrees for better readability. **4. Use Seaborn's 'barplot' function to create a bar chart, specifying the polar** coordinates manually
...

For this approach, **we will use Plotly for interactive visualization.** Steps: 1. Import pandas for data manipulation and plotly.express for plotting. 2. Load the data from 'data.csv' into a DataFrame. 3. Prepare the data by ensuring 'theta' is in radians and 'radii' and 'width' are correctly formatted. **4. Use Plotly's 'px.bar_polar' function to create an interactive polar bar chart**
...

### The Middle Generated Figures






Not Executable

### Visual Feedback

7. **Enhancements for Clarity**:
   - Add a title to the plot using `ax.set_title('Your Title Here')` to provide context.
   **- Adjust the limits of the radial axis if necessary using `ax.set_ylim(min_value, max_value)` to ensure all data points are visible.**
...

2. **Convert Data for Polar Plot**:
   - Since you are creating a polar bar plot, you need to convert the `theta_degrees` to radians because the polar plot in Matplotlib uses radians.
3. **Create a Polar Plot**:
   **- Instead of using `sns.barplot`, you will need to create a polar plot using `plt.subplot(projection='polar')`.**
...

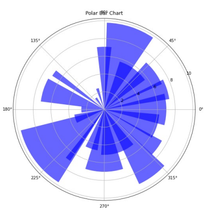
...
4. **Fix Rendering Error by Adjusting Plot Parameters**:
- Remove the width='width' argument from px.bar_polar if the user did not request it. **This prevents potential rendering errors and simplifies the chart for basic use cases.**
...

### Final Code

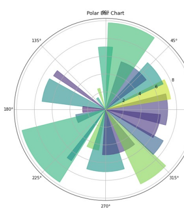
```
...
# Create a polar plot
fig, ax = plt.subplots(subplot_kw={'projection': 'polar'})
bars = ax.bar(theta, radii, width=width, bottom=0.0)
# Customize the plot
ax.set_title('Polar Bar Chart of Radii by Theta', va='bottom')
...
```

### Final Figure

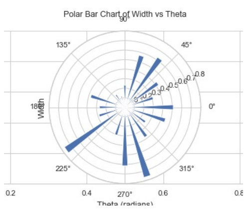
**Zero Shot**



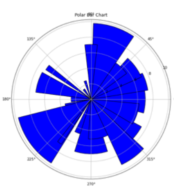
**CoT**



**Chat2VIS**




**MatPlotAgent**



**VisPath**

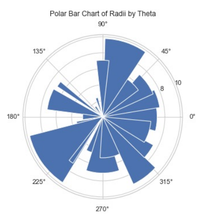



Figure 5: **Overview of the end-to-end process of generating a polar bar chart.**

## Case 3

### User Query

**Visualize a large number of time series in 3 different ways**

### Raw Data (First 5 Rows)

| | 0 | 1 | 2 | 3 | 4 |
|---|---|---|---|---|---|
| 0 | 0.122476 | -0.594418 | -1.499821 | -1.844712 | 0.479895 |
| 1 | 0.089344 | 1.409404 | 0.469888 | -1.492742 | -3.272114 |
| 2 | 0.277713 | 0.607031 | 0.295746 | 1.783222 | 2.551155 |
| 3 | 1.222288 | 1.631578 | 2.498500 | 3.093966 | 2.751248 |
| 4 | 0.905164 | 3.029735 | 3.327207 | 3.499925 | 3.822442 |

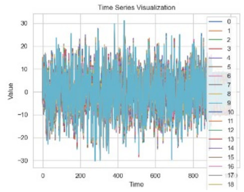
### Multi-Path Reasoning

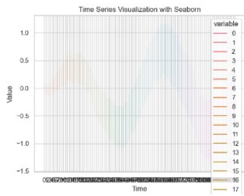
1. **Methodology 1: **Using Matplotlib for Line Plots****
...
- **Step 3:** **Iterate through each column of the DataFrame and plot the time series using `plt.plot(df[column])`.** This will create a line plot for each time series.
  - **Step 4:** Use `plt.title('Time Series Visualization')` to set a title for the plot and `plt.xlabel('Time')` and `plt.ylabel('Value')` to label the axes. This provides context to the viewer.
...

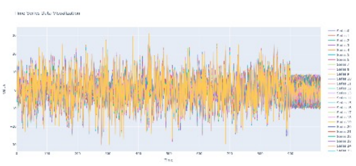
2. **Methodology 2: **Using Seaborn for Enhanced Aesthetics****
...
  - **Step 3:** Reshape the DataFrame using `pd.melt(df)` to convert it into a long format suitable for Seaborn. This allows us to plot multiple time series easily.
  - **Step 4:** Use **`sns.lineplot(data=melted_df, x='variable', y='value', hue='variable')`** to create a line plot with different colors for each time series.
...

3. **Methodology 3: **Using Plotly for Interactive Visualizations****
...
- **Step 3:** Reshape the DataFrame using `pd.melt(df)` to convert it into a long format suitable for Plotly.
  - **Step 4:** **Create an interactive line plot using `fig = px.line(melted_df, x='variable', y='value', color='variable', title='Interactive Time Series Visualization')`.** This enables users to hover over points for more information.
...

### The Middle Generated Figures







### Visual Feedback

...
**Create Multiple Subplots**: **Instead of plotting all time series on a single plot, create a grid of subplots to visualize the data in three different ways.** You can use `plt.subplots()` to create a grid layout.
...

...
**Create Multiple Plots**: Since the user requested three different visualizations, **you need to create three separate plots.** …
**Modify the Code to Create Subplots**: Use `plt.subplots()` to create a figure with multiple subplots.
...

...
2. **Modify the Code to Include Multiple Plots**:
- **After creating the line plot, add code to create a scatter plot using `px.scatter()`.**
- Then, create a bar plot using `px.bar()`.
...

### Final Code

```
...
# Create subplots
fig, axs = plt.subplots(1, 3, figsize=(15, 5))
 …
# Scatter Plot
axs[1].scatter(df.index, df.iloc[:, 1], label='Time Series 2', color='orange')
axs[1].set_title('Scatter Plot of Time Series 2')
...
```

### Final Figure

**Zero Shot** **CoT** **Chat2VIS** **MatPlotAgent** **VisPath**

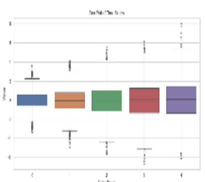
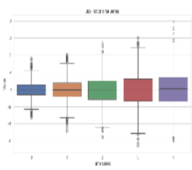
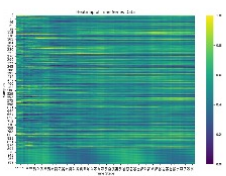
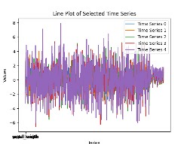
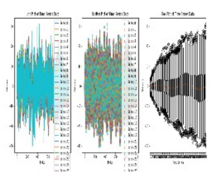






Figure 6: **Overview of the end-to-end process of visualizing time series data in three different ways.**