From Words to Structured Visuals: A Benchmark and Framework for Text-to-Diagram Generation and Editing

Supplementary Material

A. DiagramGenBenchmark Details

A.1. Dataset Curation

In constructing DiagramGenBenchmark, we employed a rigorous data curation process to ensure quality and diversity across the dataset. This section details our multi-step pipeline, from raw data collection to task-specific dataset creation, as illustrated in Figure 5.

Data Sources We gathered raw Tex and DOT language-based code data from multiple sources, including the VGQA [54], datikz [6], and datikz-v2 [6] datasets on Hugging Face, as well as publicly available repositories on GitHub and Overleaf. These sources yielded a comprehensive collection of over 13,000 raw code samples, originally sourced from various research papers on arXiv.

Data Processing Collected raw code samples were subjected to manual filtering. Out of 13,000 initial samples, we retained only those that could be compiled successfully into images, ensuring code-to-diagram consistency. We executed each code sample using respective Tex and DOT compilers and stored each code-image pair in separate directories. After this filtering, we obtained 6,983 code-image pairs as the final dataset.

Data Annotation To label the data, we defined eight diagram categories: model architecture diagram, flowchart, line chart, directed graph, undirected graph, table, bar chart, and mind map. Automatic labeling was conducted using the GPT-4o [3].

A.2. Dataset Task

Dataset Tasks: We constructed three distinct tasks to enhance the benchmark's applicability:

- *Diagram generation*: This task involves generating code from human instructions. We created human-like instructions by reverse-engineering existing open-source diagram code using the closed-source GPT-4o model. A total of 6,713 training instances and 270 testing instances were generated for this task.
- *Diagram coding*: This task requires generating compliable diagram code from an image. Using the code and image pairs from the diagram generation task, we retained only image-code pairs and created a set of 6,713 training instances and 270 testing instances.
- *Diagram editing*: This task simulates code editing based on revision instructions. We generated modification suggestions using GPT-4o and applied them to the original code with the CodeGeeX-4 [52] model. Only compilable, unique modified codes were selected, resulting in 1,400 training instances and 200 testing instances.

A.3. Dataset Distribution

Figure 6 illustrates the data distribution for different diagram types in the diagram generation / diagram coding task and the diagram editing task.

B. Detailed Prompts for DiagramAgent

This appendix provides a comprehensive view of the prompt designs used in DiagramAgent's core components, including the Plan Agent, Complete Query handling, Diagram-to-Code Agent, Code Agent, and Check Agent. These prompts support different tasks such as drawing, modifying diagrams, and error checking, ensuring that each module contributes to accurate and coherent diagram generation and modification.

C. Comparison with Existing Benchmarks

As shown in Table 8, DiagramGenBenchmark differs from existing benchmarks by providing both natural language and image inputs specifically for structured diagram generation and editing. Unlike traditional code generation benchmarks focused on simple text-based tasks, our benchmark encompasses a diverse range of diagram types and incorporates multi-level evaluation metrics (e.g., Pass@1, CodeBLEU, CLIP-FID, PSNR). Additionally, DiagramGenBenchmark includes three specialized tasks—diagram generation, diagram coding, and diagram editing—designed to evaluate both diagram creation and editing capabilities, making it a comprehensive resource for advancing multimodal diagram generation research.

D. Human Evaluation Details

To evaluate the performance of DiagramAgent, three evaluators with master's degrees independently rated the quality of generated diagrams on a scale from 1 to 5. The evaluation focused on the similarity between the generated diagrams and reference diagrams for two core tasks: the diagram generation and the diagram editing. Each score represents the following specific criteria:

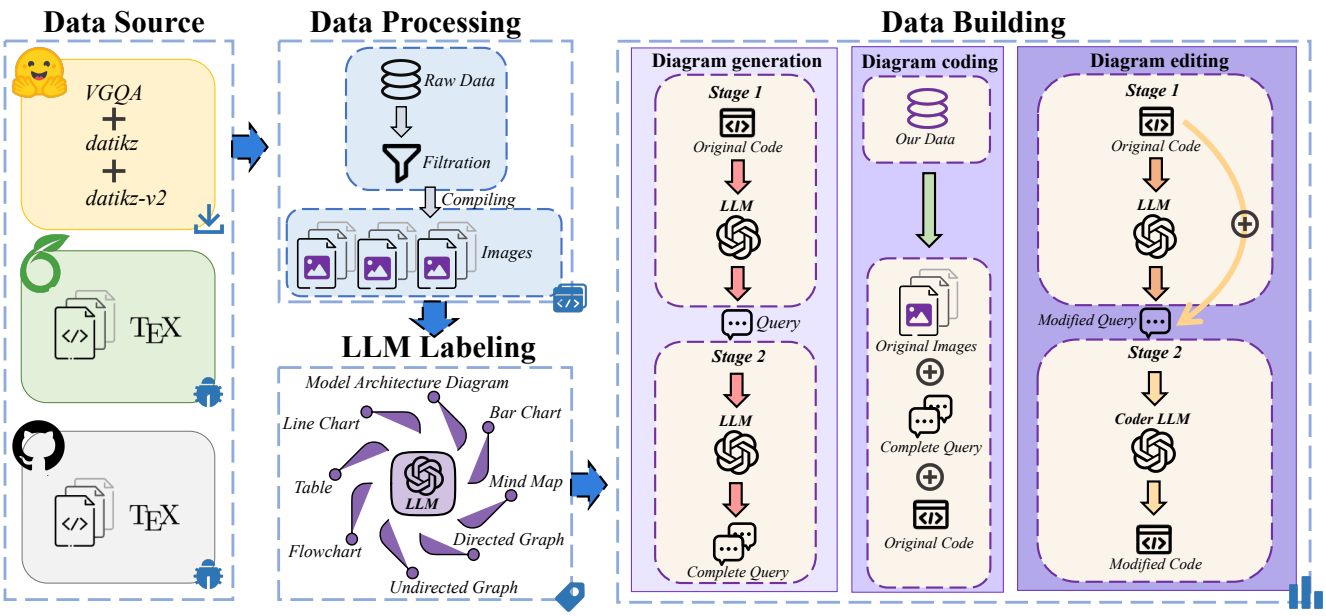

Figure 5. DiagramGenBenchmark Data Curation Process

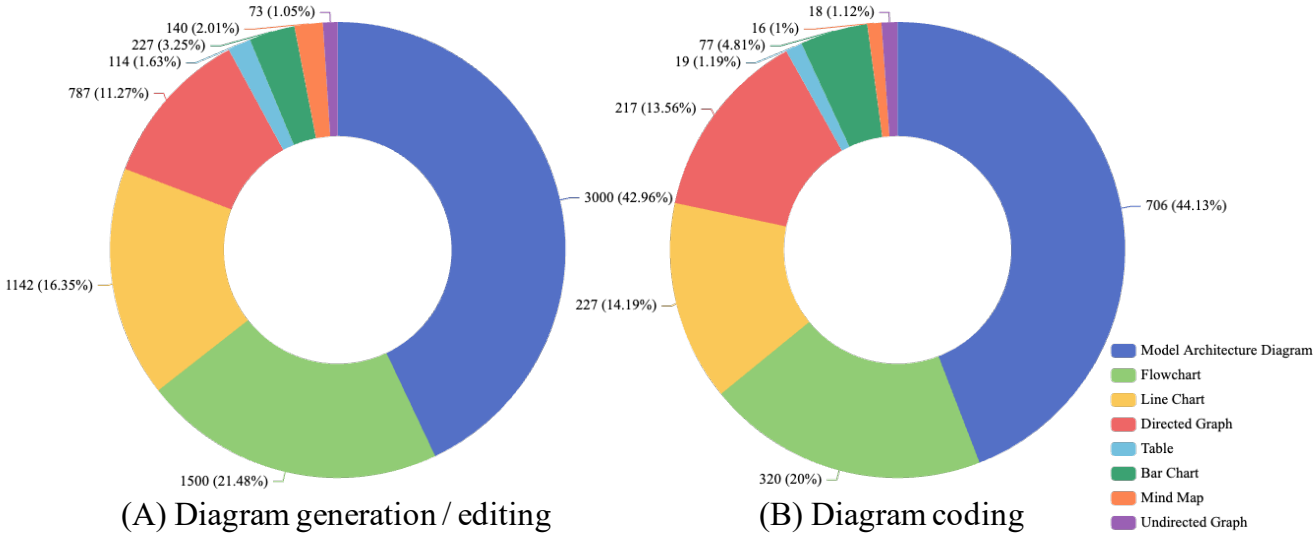

Figure 6. Data Distribution of DiagramGenBenchmark.

- **1**: The generated diagram shows very low similarity to the reference diagram, containing significant logical or structural errors and failing to fulfill the intended instructions.
- **2**: The generated diagram partially reflects the intended design but exhibits several issues, including low accuracy and visual coherence, and falls below the expected quality standards.
- **3**: The generated diagram meets basic requirements for structure and intent, though it has noticeable inconsistencies or errors that reduce its overall quality.
- **4**: The generated diagram is mostly accurate and visually coherent, with only minor, negligible errors; it aligns closely with the intended reference.
- **5**: The generated diagram is completely accurate and highly similar to the reference diagram, with no discernible errors; it fully meets or exceeds expectations for clarity and precision.

Each evaluator independently scored each of the N test items, resulting in three scores per item. To calculate the mean score for each item, we average the three scores from

Table 8. Comparison of DiagramGenBenchmark with Existing Benchmarks. "NL" indicates Natural Language inputs, "I" represents Image inputs, and "Code" denotes code output. Evaluation metrics vary across benchmarks, including pass rate, multi-level metrics, and other task-specific measures.

Benchmark	Source	# of Chart Types	# of Test Instances	Input Format	Output Format	Evaluation Metric
HumanEval [9]	Human Curated	-	164	Code	Code	Pass Rate
MBPP [5]	Human Curated	-	500	NL+Code	Code	Pass Rate
MMCode [20]	Crawl	-	263	NL+Code	Code	Pass Rate
MatPlotBench [46]	Human Curated	13	100	NL	Code	GPT-4V Score
Design2Code [34]	Crawl	-	484	I+NL	Code	Multi-Level
Plot2Code [42]	Human Curated	6	132	I	Code	Pass Rate, Text-Match, etc.
Diagram generation (Ours)	Human Curated	8	270	NL	Code,Diagram	Pass Rate, CodeBLEU, ROUGE-L, etc.
Diagram coding (Ours)	Human Curated	8	270	I	Code	Pass Rate, CodeBLEU, ROUGE-L, etc.
Diagram eidting (Ours)	Human Curated	8	200	NL+Code	Code,Diagram	Pass Rate, CodeBLEU, ROUGE-L, etc.

Plan Agent

< System >

As a task distributor, analyze user instructions and expand them as needed. For "Drawing" pass the expanded instructions to the code agent. For "Modifying" directly forward the instructions to the diagram-to-code agent without expansion.

< User >

As a task distributor, your role is to analyze user instructions and expand them before passing the expanded instructions to the code agent. Users can choose between "Drawing" or "Modifying":

1. **Drawing**:
   - Carefully analyze the user's request and perform any necessary expansions to ensure all details are covered.
   - Pass the expanded instructions to the code agent.
2. **Modifying**:
   - Directly forward the user's instructions to the diagram-to-code agent without any expansion.

Make sure to account for all possible details and variations in the expansion of drawing instructions to generate accurate outputs.

Figure 7. Prompt for Plan Agent: Analyzing and expanding user instructions based on the type of task (Generation or Editing) to ensure clarity before passing to the code generation stage.

the evaluators:

$$\text{Mean Score for Item } i = \frac{S_1^{(i)} + S_2^{(i)} + S_3^{(i)}}{3} \tag{10}$$

where $S_1^{(i)}$, $S_2^{(i)}$, and $S_3^{(i)}$ represent the scores given by the three evaluators for item i.

After obtaining the mean score for each item, we calculate the overall average score for the model across all N test items. This final score is computed as:

$$\text{Final Score for Model} = \frac{\sum_{i=1}^{N} \text{Mean Score for Item } i}{N} \tag{11}$$

This final score provides a standardized measure of the model's ability to generate diagrams that closely resemble the reference diagrams. It reflects the model's performance across each task.

Expand Complete Queries

< System >

You are an expert in expanding the original question based on the code results. Your task is to extend the original question. Before the extension, you are not aware of the code results. Assume the code is the final desired outcome, but the original question does not sufficiently describe your needs. You need to elaborate on the original question, but the extended content cannot include code snippets. Please make the extension as detailed as possible.

< User >

Please refer to the output format of the example below and extend the question based on the provided TikZ or DOT code and the original question. Do not include any code content. The extended question should relate to the code and maintain the same tone as the original question. The output format is as follows:
1. Do not explain the code.
2. The output should not contain any code.
3. The output should be concise and in a human-like tone.
4. Please ensure that the extended output is consistent with the code results.

Example:
Original question:
How can I use LaTeX and TikZ to draw a neural network architecture diagram that includes an input layer, multiple hidden layers, and an output layer? I would like each layer to have a different number of neurons and to label each layer with its name.
Code:
```
\documentclass[tikz]{standalone}
\usepackage{neuralnetwork}
. . . . . .
\hiddenlayer[count=5, bias=false]
\linklayers
\outputlayer[count=8, title=Output Layer, text=\xout]
\linklayers

\end{neuralnetwork}
\end{document}
```
Output:
How can I use LaTeX and TikZ to draw a neural network architecture diagram that includes an input layer, multiple hidden layers, and an output layer? I want each layer to have a different number of neurons and to be labeled appropriately. . . . .

The original question is as follows:
{ Query }
The code is as follows:
{ Code }

Figure 8. Prompt for Expanding Drawing Query to Generate Complete Query.

E. Diagram Coding Complete Results

This appendix provides a complete analysis of the experimental results for the diagram coding task with DiagramAgent, expanding on the brief introduction provided in Section Sec. 6.2 of the main text. Due to space constraints, only key findings were highlighted in the main text, while this appendix presents a more in-depth evaluation of DiagramAgent's performance relative to other models across various metrics, as shown in Tables 9 and 10.

Table 9. Main results for diagram coding task (Diagram-to-Code Agent). The best result in each metric is bolded.

Model	Size	Pass@1↑	ROUGE-L↑	codeBLEU↑	Edit Dist.↓	chrF↑	RUBY↑
Yi-VL [47]	34B	2.22	20.01	70.57	95.43	11.68	12.53
Qwen2-VL [39]	8B	28.89	31.74	80.04	88.13	28.39	21.21
Internlm-xcomposer2.5 [50]	7B	3.33	28.47	77.35	92.35	18.74	17.97
Llama-3.2-Vision [11]	11B	27.78	21.94	75.37	92.92	16.37	13.95
Phi-3.5-vision [2]	4B	24.07	27.53	76.56	90.01	20.86	17.96
Llava-v1.6 [19]	34B	8.89	26.68	76.53	93.46	21.00	16.30
Cogvlm2-llama3 [16]	19B	3.70	14.42	70.72	97.07	8.27	8.91
Deepseek-vl [24]	7B	50.74	25.18	76.48	88.82	18.35	16.13
GPT-4o [3]	-	64.07	39.95	81.78	86.68	34.40	26.18
GLM-4-plus [13]	-	51.48	35.92	80.16	86.12	29.10	24.60
Gemini-1.5-pro [31]	-	17.78	38.66	80.75	88.05	30.00	25.62
DiagramAgent	7B	**68.89**	**48.99**	**84.64**	**72.74**	**46.98**	**37.46**

Table 10. Ablation study for diagram coding task (Diagram-to-Code Agent). Each result shows the performance of DiagramAgent under various component configurations, with the decrease in each metric from the full model indicated in parentheses.

Diagram Coding	Size	Pass@1↑	ROUGE-L↑	codeBLEU↑	Edit Dist.↓	chrF↑	RUBY↑
DiagramAgent	7B	**68.89**	**48.99**	**84.64**	72.74	**46.98**	**37.46**
– w/o GPT-4o	7B	62.59 (-6.30)	48.71 (-0.28)	84.57 (-0.07)	**72.69** (-0.05)	46.49 (-0.49)	37.22 (-0.24)
– w/o Compiler	7B	53.33 (-15.56)	48.46 (-0.53)	84.52 (-0.12)	73.51 (+0.77)	46.91 (-0.07)	36.66 (-0.80)
– w/o GPT-4o & Compiler	7B	52.59 (-16.30)	47.81 (-1.18)	84.21 (-0.43)	74.56 (+1.82)	45.28 (-1.70)	35.81 (-1.65)

Diagram to Code Agent

< User >

Please output the code directly according to the following instructions.

(<image> Please convert this image to LaTeX code./<image> Please convert this image to DOT code.)

< a. GPT-4o Feedback >

You are a code design expert. I will provide LaTeX or DOT code that may contain errors, along with an image, a problem description, and code modification suggestions.

Your task is:
- Correct the code based on the image, the provided problem, and the modification suggestions.
- Output the corrected code directly, without any explanation or commentary.

Question: { Question }
Code modification suggestions: { GPT-4o Feedback }
May contain incorrect LaTeX or DOT code: { Model Response }
Answer:

< b. Compiler Feedback >

You are a code design expert. I will provide LaTeX or DOT code that may contain errors, along with an image, a problem description, and error messages.

Your task is:
- Correct the code based on the image, the provided problem, and the error messages.
- Output the corrected code directly, without any explanation or commentary.

Question: { Question }
Error messages: { Error Content }
May contain incorrect LaTeX or DOT code: { Compiler Feedback }

Figure 9. Prompt for Diagram-to-Code Agent: Handling direct conversion of diagrams to LaTeX or DOT code based on user-provided images and instructions.

Check Agent

< User >

You are an expert in code validation. Below is LaTeX or DOT code, an image, and a question.

Your task is:
- Check for Errors: Identify any syntax errors, discrepancies, or mismatches between the code and the image.
- Provide Suggestions: If errors exist, explain the issues and suggest specific changes without directly providing the modified code.
- Validation: If no errors are found, respond with 'The code is correct' instead of just 'True'.
- Ensure your feedback is clear, thorough, and actionable, but do not include the corrected code itself.

Code: { Model Response }
Answer:

Figure 10. Check Agent Prompt for Diagram Coding: Verifying code accuracy for Diagram-to-Code Agent outputs, ensuring logical consistency and correctness.

Main Results Table 9 summarizes the main results for the diagram coding task, where DiagramAgent consistently outperforms other models across all key evaluation metrics, demonstrating its robustness and precision in translating diagrams into code. For Pass@1, DiagramAgent achieves 68.89, which is significantly higher than the scores of open-source models, such as Qwen2-VL (28.89) and Llama-3.2-Vision (27.78), as well as the scores of closed-source models such as GPT-4o (64.07) and GLM-4-plus (51.48). This high Pass@1 indicates that DiagramAgent performs well in generating correct code on the first attempt, reducing the need for iterative refinements and thus enhancing efficiency in the coding process.

The ROUGE-L score of 48.99 for DiagramAgent highlights its strength in preserving the structural and sequen-

Code Agent

< *System* >

You are an expert in TiKZ and DOT.

< *User* >

Please output the complete code.

{ Complete Query }

< a. GPT-4o Feedback >

You are a code design expert. I will provide LaTeX or DOT code that may contain errors, along with a problem description, and code modification suggestions.

Your task is:
- Correct the code based the provided problem, and the modification suggestions.
- Output the corrected code directly, without any explanation or commentary.

Question:
{ Question }
Code modification suggestions:
{ Suggestion }
May contain incorrect LaTeX or DOT code:
{ Code }

Answer:

< b. Compiler Feedback >

The code is as follows:
{ Code }
Error :
{ Error Content }

Figure 11. Code Agent Prompt for Diagram Generation: Generating code based on detailed instructions provided by the user.

Code Agent

< *System* >

You are an expert at modifying code.

< *User* >

Your task is to return the fully modified code according to the user's instructions.

{ Question }

The code is as follows:

{ Code }

Output the code directly.

< a. GPT-4o Feedback >

You are a code design expert. I will provide LaTeX or DOT code that may contain errors, along with a problem description, and code modification suggestions.

Your task is:
- Correct the code based the provided problem, and the modification suggestions.
- Output the corrected code directly, without any explanation or commentary.

Question:
{ Question }
Code modification suggestions:
{ Suggestion }
May contain incorrect LaTeX or DOT code:
{ Code }

Answer:

< b. Compiler Feedback >

The code is as follows:
{ Code }
Error :
{ Error Content }

Figure 12. Code Agent Prompt for Diagram Editing: Handling feedback-based adjustments to existing code by applying suggestions from either GPT-4o or compiler feedback.

Check Agent

< *System* >

You are an expert in TiKZ and DOT.

< *User* >

You are an expert in code validation. Below is LaTeX or DOT code and a question.

Your task is:
- Check for Errors: Identify any syntax errors, discrepancies, or mismatches between the code and the question.
- Provide Suggestions: If errors exist, explain the issues and suggest specific changes without directly providing the modified code.
- Validation: If no errors are found, respond with 'The code is correct' instead of just 'True'.
- Ensure your feedback is clear, thorough, and actionable, but do not include the corrected code itself.

Question:
{ Question }
Code:
{ Code }

Figure 13. Check Agent: Using GPT-4o to perform error detection and provide correction suggestions within Code Agent outputs.

tial accuracy of generated code, outperforming models such as GLM-4-plus (35.92) and GPT-4o (39.95). ROUGE-L is crucial in tasks that require alignment with the target's structural elements, and DiagramAgent's high score reflects its ability to capture complex syntactic patterns essential for correct code generation from diagrams. Additionally, DiagramAgent attains a codeBLEU score of 84.64, the highest among all models, which captures both syntactic and semantic alignment with the target code. Compared to other models, such as Qwen2-VL (80.04) and Gemini-1.5-pro (80.75), DiagramAgent demonstrates a marked improvement in generating both structurally and functionally accurate code, underscoring its capability to handle the nuanced requirements of diagram coding tasks. In terms of Edit Distance, DiagramAgent achieves a score of 72.74, which is the lowest among all models, indicating that fewer modifications are needed to align generated code with the target. Lower Edit Distance highlights DiagramAgent's initial output accuracy, minimizing post-generation adjustments and improving overall workflow efficiency. This metric contrasts with higher Edit Distances observed in other models, such as Cogvlm2-llama3 (97.07) and Yi-VL (95.43), which indicate a greater need for corrections. Additionally, DiagramAgent's chrF score of 46.98, higher than all compared models, reflects the fine-grained character-level alignment achieved by DiagramAgent, essential for high-quality code generation. Finally, DiagramAgent's RUBY score of 37.46 is the highest across models, underscoring the model's robustness and its capacity to generate code that aligns well with human coding standards, surpassing both close-source models, such as GPT-4o (26.18) and open-source options like Qwen2-VL (21.21). These comprehensive results col-

lectively affirm DiagramAgent's superior performance in the diagram coding task across all evaluation metrics.

Ablation Study Table 10 presents the ablation study results, which evaluate the contributions of the GPT-4o verification and compiler debugging modules to DiagramAgent's overall performance. When the GPT-4o verification module is removed, DiagramAgent's Pass@1 score drops by 6.30, while ROUGE-L decreases slightly by 0.28. These reductions suggest that the verification module enhances first-attempt accuracy, as indicated by the drop in Pass@1, and plays a role in preserving the structural fidelity of generated code, reflected in the ROUGE-L score. The minor reduction in codeBLEU by 0.07 further points to the verification component's contribution to maintaining both syntactic and semantic consistency, while the minimal changes in Edit Distance (+0.05) and chrF (-0.49) indicate that this module has a moderate impact on fine-grained accuracy and error reduction. Removing the compiler debugging module has a more substantial effect, leading to a 15.56 decrease in Pass@1 and a 0.53 drop in ROUGE-L. These changes reflect the compiler's crucial role in achieving syntactic accuracy, as it likely identifies and corrects structural inconsistencies early in the generation process. The Edit Distance increase of +0.77 suggests that the absence of debugging requires more extensive post-generation edits to achieve target alignment. The drops in chrF (-0.07) and RUBY (-0.80) underscore the compiler's importance in ensuring character-level precision and maintaining the overall quality of the generated code. When both the GPT-4o verification and compiler debugging components are removed, DiagramAgent's performance declines across all metrics, with a Pass@1 decrease of 16.30 and an Edit Distance increase of +1.82. The additional decreases in ROUGE-L (-1.18), codeBLEU (-0.43), chrF (-1.70), and RUBY (-1.65) further underscore the importance of these components for achieving DiagramAgent's high standards in code generation. Without these modules, the model's outputs show a higher propensity for errors, requiring greater manual adjustment and impacting both structural and semantic accuracy. This configuration demonstrates the complementary roles of verification and debugging in maintaining DiagramAgent's precision and robustness in diagram coding tasks.

F. Error Analysis

Our analysis of errors covers three primary tasks: diagram generation, diagram editing, and diagram coding, with common error types illustrated in Figures 14, 15, and 16. Identifying and addressing these errors is crucial for enhancing the overall accuracy of generated diagrams, and it represents an essential focus for future work.

F.1. Diagram Generation Errors

For the diagram generation, three prominent error types emerge:

- **Diagram Shape Understanding Error**: This error occurs when the model misinterprets the shape requirements specified in the user query. For instance, the model may generate incorrect or overly simplified shapes, as seen in Figure 14 (left). This typically arises from ambiguity in the instruction or ineffective shape recognition capabilities.
- **Diagram Structure Understanding Error**: The model occasionally fails to capture the hierarchical or relational structure of diagrams, leading to incorrect element placements or connections, as shown in Figure 14 (center). This error is often due to limited structural parsing of complex diagram elements.
- **Diagram Content Understanding Error**: Errors in content understanding arise when the model misinterprets content-specific details, such as labels or numeric data, leading to inaccuracies in representation, as illustrated in Figure 14 (right). This may result from insufficient parsing of content semantics in instructions.

F.2. Diagram Coding Errors

For the diagram coding, errors typically involve spatial and layout issues:

- **Position Understanding Error**: Misinterpretations of element positions result in misplaced components, as shown in Figure 15 (left). This error often stems from insufficient spatial understanding of diagram elements.
- **Layout Understanding Error**: Incorrect layout arrangements, where elements are poorly aligned or spaced, disrupt diagram coherence, as illustrated in Figure 15 (center). This error arises from limitations in layout parsing and spatial arrangement.
- **Color Understanding Error**: Similar to the diagram editing, color errors also appear in diagram coding, leading to misinterpreted color schemes that impact semantic meaning, as seen in Figure 15 (right). This is due to inadequate interpretation of color attributes in diagrams.

F.3. Diagram Editing Errors

In the diagram editing, the following errors are most prevalent:

- **Color Understanding Error**: Color misinterpretation often leads to incorrect assignments, reducing visual clarity, as shown in Figure 16 (left). This issue typically arises from inadequate parsing of color-related instructions.
- **Line Type Understanding Error**: Misunderstandings in line type specifications, such as solid vs. dashed lines, lead to errors in edge representation, as seen in Figure

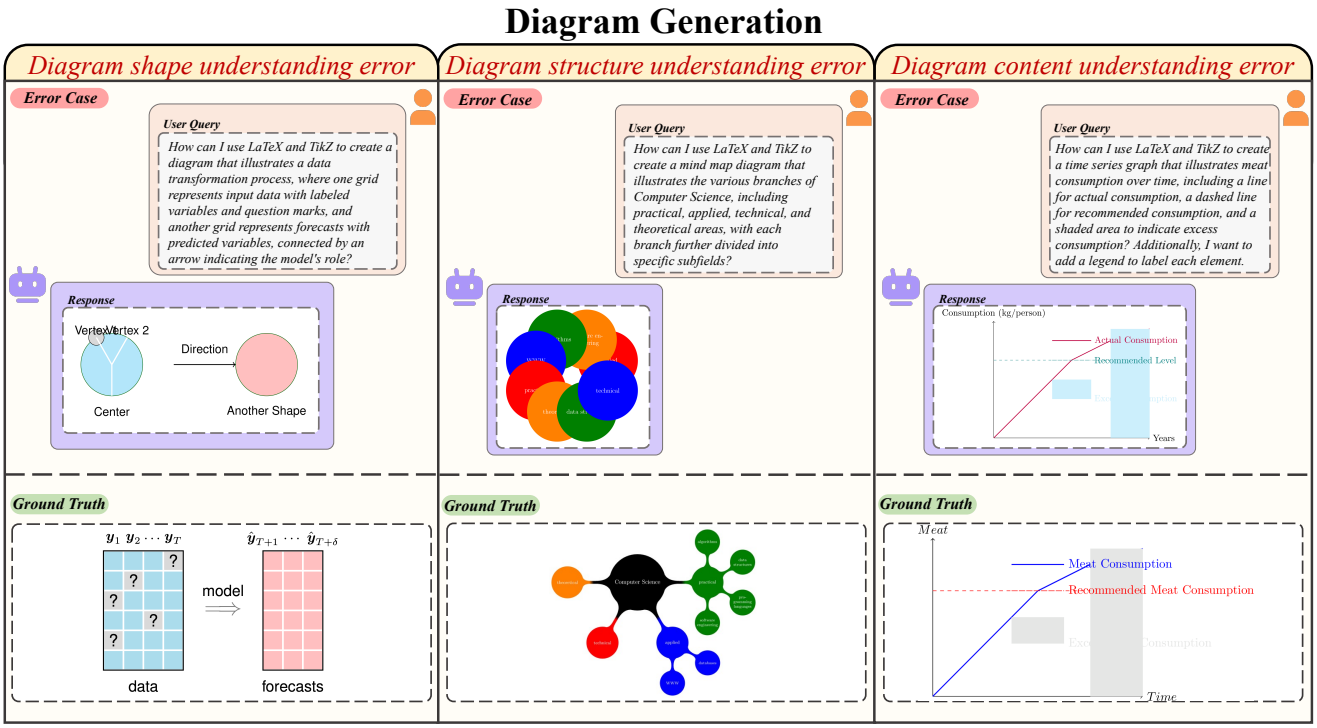

Figure 14. Common Error Types in the Diagram Generation: (Left) Shape Understanding Error, (Center) Structure Understanding Error, (Right) Content Understanding Error

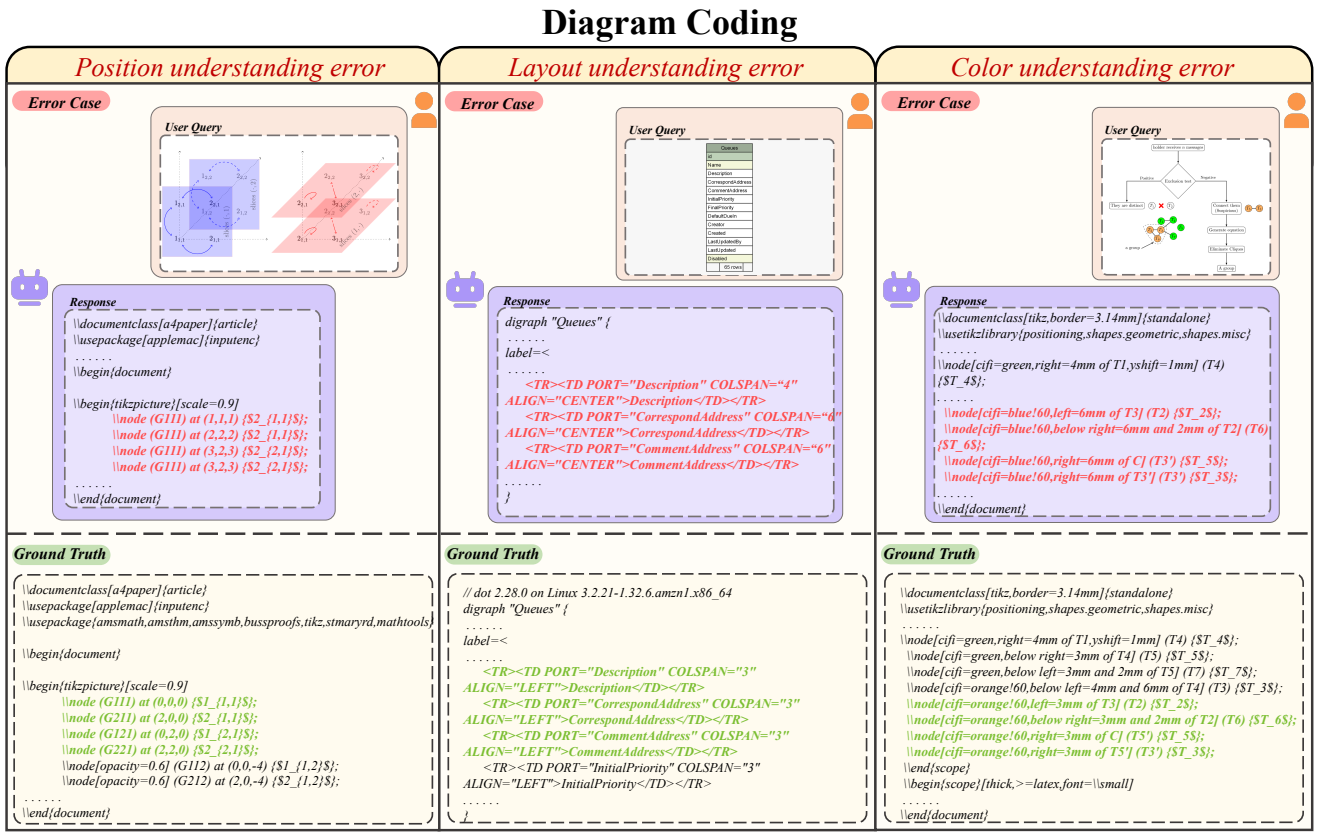

Figure 15. Common Error Types in the Diagram Coding: (Left) Position Understanding Error, (Center) Layout Understanding Error, (Right) Color Understanding Error

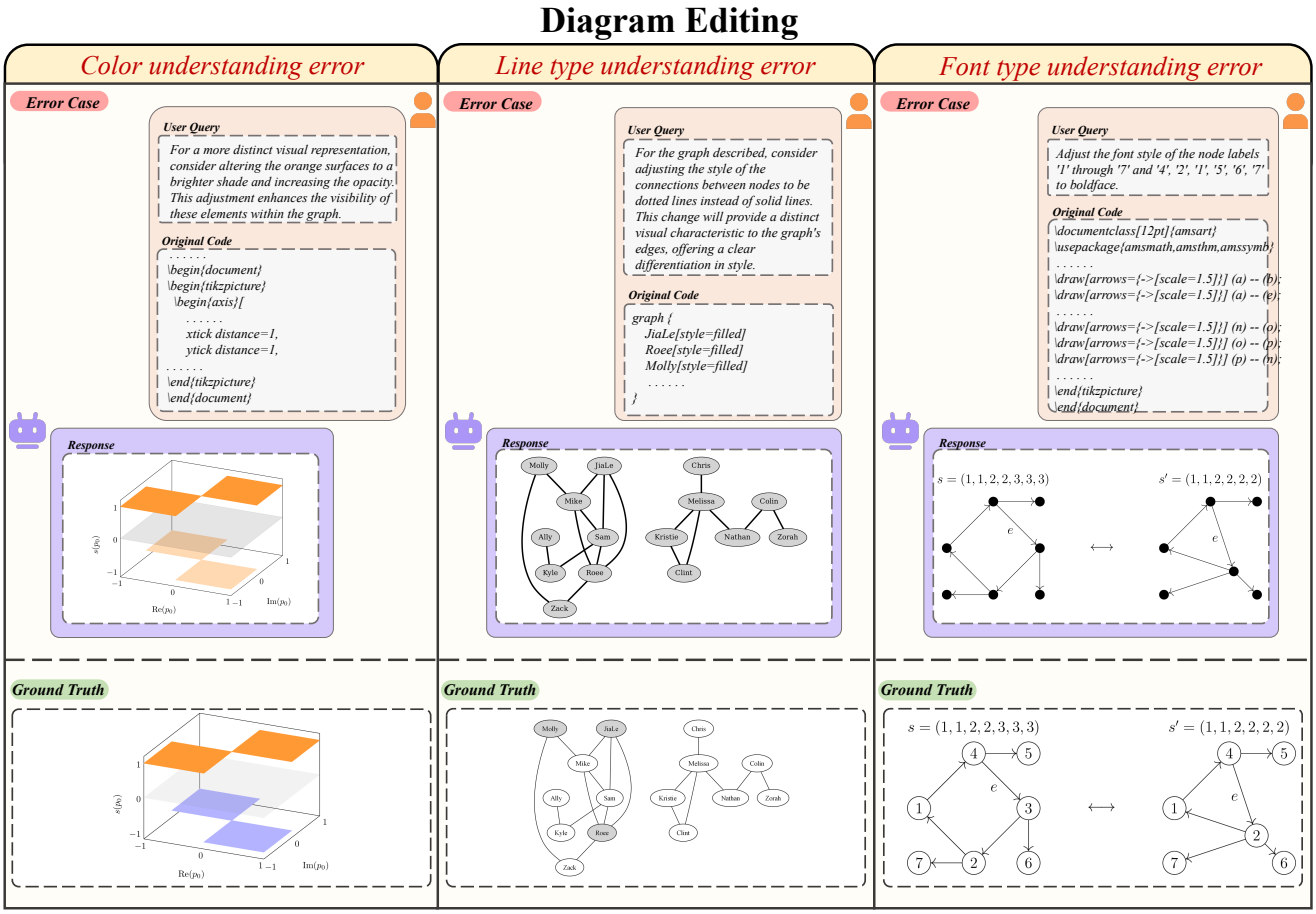

Figure 16. Common Error Types in the Diagram Editing: (Left) Color Understanding Error, (Center) Line Type Understanding Error, (Right) Font Type Understanding Error

16 (center). This stems from insufficient differentiation between line types in code interpretation.

- **Font Type Understanding Error**: Incorrect font usage in diagrams, especially for labels, affects readability and visual coherence, as illustrated in Figure 16 (right). This issue is often due to limitations in font parsing and application.

Addressing these errors is essential for enhancing the precision and fidelity of the generated diagrams. Future work should focus on refining the model's capabilities in shape, structure, content, color, line type, font, position, and layout understanding to better align generated diagrams with user expectations.

G. Qualitative Analysis

We provide qualitative examples to illustrate the performance of DIAGRAMAGENT on both *Diagram Generation* and *Diagram Editing* tasks.

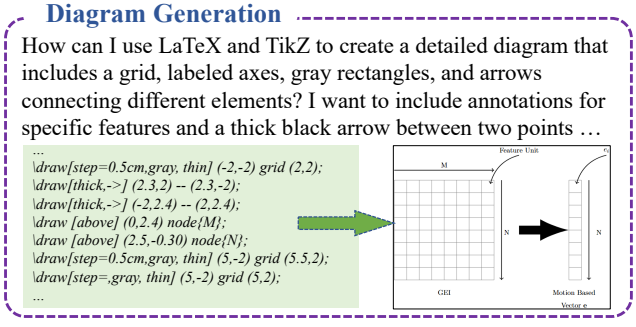

Figure 17. Example of diagram generation.

Diagram Generation Given a textual description, DIAGRAMAGENT generates a corresponding diagram. Figure 17 presents an example where the system correctly constructs structural elements, including grids, arrows, and labeled components, while maintaining spatial alignment.

Diagram Editing Beyond generation, DIAGRAMAGENT allows interactive modifications based on user-specified edits. Figure 18 illustrates an example where the system suc-

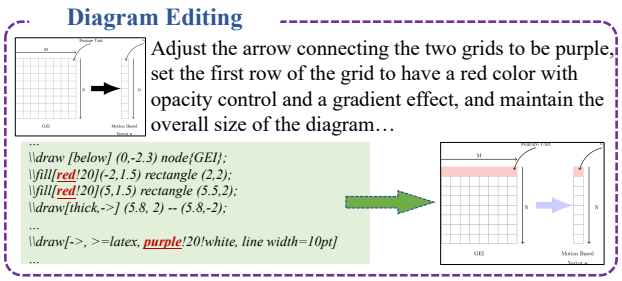

Figure 18. Example of diagram editing.

cessfully updates diagram attributes, including modifying arrow colors, adjusting grid configurations, and refining annotations.