NVAGENT: Automated Data Visualization from Natural Language via Collaborative Agent Workflow

Anonymous ACL submission

Abstract

Natural Language to Visualization (NL2VIS) seeks to convert natural-language descriptions into visual representations of given tables, empowering users to derive insights from large-scale data. Recent advancements in *Large Language Models* (LLMs) show promise in automating code generation to transform tabular data into accessible visualizations. However, they often struggle with complex queries that require reasoning across multiple tables. To address this limitation, we propose a collaborative agent workflow, termed NVAGENT, for NL2VIS. Specifically, NVAGENT comprises three agents: *processor* for database processing and context filtering, *composer* for planning visualization generation, and *validator* for code translation and output verification. Comprehensive evaluations on the VisEval benchmark demonstrate that NVAGENT consistently surpasses state-of-the-art baselines, achieving 7.88% and 9.23% improvements in *single-* and *multi-table* scenarios. Qualitative analyses further highlight that NVAGENT maintains nearly a 20% performance margin over previous methods, underscoring its capacity to produce high-quality visual representations from complex, heterogeneous data sources.[*]

1 Introduction

"Turning data into insight", has long been a key goal in our increasingly data-rich, information-driven society (Fiorina). To achieve this, *Natural Language to Visualization* (NL2VIS) plays a crucial role in transforming natural-language descriptions into visual representations (*e.g.*, charts, plots, and histograms) grounded on tabular data (Sah et al., 2024). This approach enables users to interact with data intuitively, facilitating the extraction of patterns and insights from large and complex datasets (Yin et al., 2024; Vartak et al., 2017).

[*]All datasets and source code are available at: `https://anonymous.4open.science/r/nvAgent-60A1`. A demo video is also provided. We strongly recommend giving a try to visualize multi-table data using chat-style NL instructions with NVAGENT.

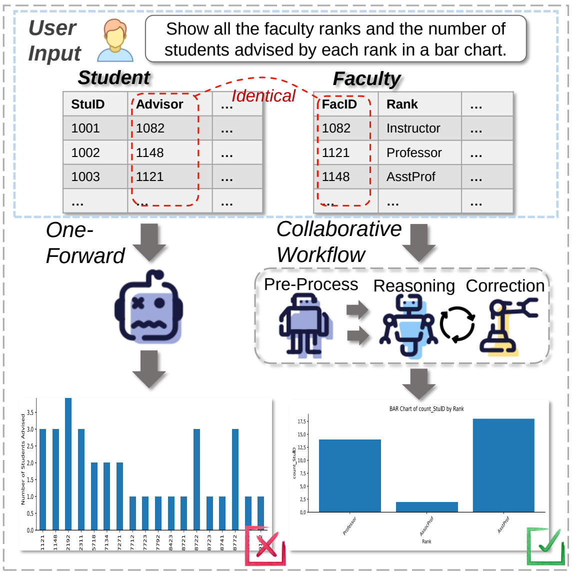


Figure 1: An example to illustrate the NL2VIS task. Formerly *"One Forward"* workflow struggled with multi-table queries due to its complex and heterogeneous structure, which could easily cause an error. NVAGENT uses a collaborative agent-based workflow for iterative interaction with data and validation to ensure accurate and valid visualization.

Recently, *Large Language Models* (LLMs) have demonstrated promising performance in NL2VIS tasks, excelling in various stages such as preprocessing (Li et al., 2024b) and code generation for visualization (Maddigan and Susnjak, 2023). These models effectively generate readable visualizations for individual datasets or databases (Li et al., 2024a). However, existing approaches encounter challenges when processing queries involving multiple tables due to incorrect joins or mis-filtering conditions, leading to visualization errors (Maddigan and Susnjak, 2023; Dibia, 2023; Chen et al., 2024c). These limitations severely restrict their applicability in real-world scenarios where data is typically distributed across multiple related tables (Khan, 2024; Lu et al., 2024).

Figure 1 shows an example to illustrate the motivation of our study. Given a natural-language (NL) query such as *"Show all the faculty ranks and the number of students advised by each rank in a bar chart"*, the system must understand that *"faculty"* information corresponds to the column *"Advisor"* and *"FacID"* across two tables. These complex cross-table visualization highlights the challenge between NL queries and databases, requiring a framework that can preprocess metadata, think *"step-by-step"* with plans, and iterative validation to ensure correctness.

These observations inspire our NVAGENT, a collaborative agent workflow for NL2VIS. NVAGENT follows the *"divide-and-conquer"* paradigm, consisting of three specialized LLM agents: a *processor* agent for database processing and context filtering, a *composer* agent for planning visualization generation, and a *validator* agent for code translation and output verification. This collaborative workflow provides a more systematic approach that can effectively handle multi-table scenarios while maintaining visualization accuracy and quality.

To validate the effectiveness of NVAGENT, we conducted extensive experiments on the VisEval benchmark (Chen et al., 2024c), which includes two scenarios: the *single-table* scenario, involving generating visualizations from individual tables, and the *multi-table* scenario, which entails integrating information from multiple tables. The results demonstrate that NVAGENT outperforms all baseline methods, achieving a 7.88% higher pass rate in *single-* and 9.23% in *multi-table* scenarios compared to the state-of-the-art method. Our ablation study that breakdown every module within NVAGENT provide solid evidence of our framework design. Qualitative analyses further highlight that NVAGENT maintains 3.64% and 18.15% margin in *single-* and *multi-table* over previous frameworks, underscoring its efficacy in producing high-quality visual representations from complex, heterogeneous data sources.

In summary, this paper makes the following key contributions: **(1)** We propose **NVAGENT**, a collaborative agent-based workflow for complex NL2VIS tasks, which decomposes the visualization generation process into manageable subtasks. **(2)** Extensive experiments and analysis are performed to validate the effectiveness *divide-and-conquer* strategy of NVAGENT for NL2VIS.

2 Problem Formulation

A typical workflow of NL2VIS tasks involves assembling queries along with tabular data as input, and automatically generating code based on established visualization libraries (*e.g.*, Matplotlib (Barrett et al., 2005), Seaborn (Waskom, 2021)) to be executed in a sandboxed environment to obtain the final chart image. However, directly generating visualization code often leads to errors due to the complexity of visualization requirements and the semantic gap between natural language and programming constructs.

Following previous works (Luo et al., 2021b; Wu et al., 2024b), we introduce *Visualization Query Language* (VQL) as an intermediate representation that bridges natural language queries and visualization code. As exemplified below, VQL combines SQL-like syntax for data operations with visualization-specific constructs (*i.e.*, VisType and Binning), making the generation process more controllable and reliable while maintaining simplicity in structure.

```
VisType: VISUALIZE BAR
Data: SELECT Date_Stored, COUNT(Document_ID)
FROM ALL_Documents GROUP BY Date_Stored
Binning: BIN Date_Stored BY WEEKDAY
```

Formally, given a natural language query q about a database schema S comprising multiple tables T and columns C, the objective of NL2VIS is to generate a visualization query v as an intermediate step, which is then translated into a visualization V that accurately represents the data in S to answer the user's query.

3 NVAGENT: Our Approach

3.1 An Overview

Figure 2 shows an overview of NVAGENT, which is composed of three specialized agents: *processor*, *composer*, and *validator*, working collaboratively to transform natural language queries into accurate visualizations. Starting with a user query q and schema S, our approach first leverages the *processor* to filter schema S' and generate additional context including augmented explanation and query complexity classification. The *composer* then generates a VQL query as an intermediate representation through reasoning step by step. Finally, the *validator* ensures correctness via iterative validation and refinement until a valid visualization is produced.

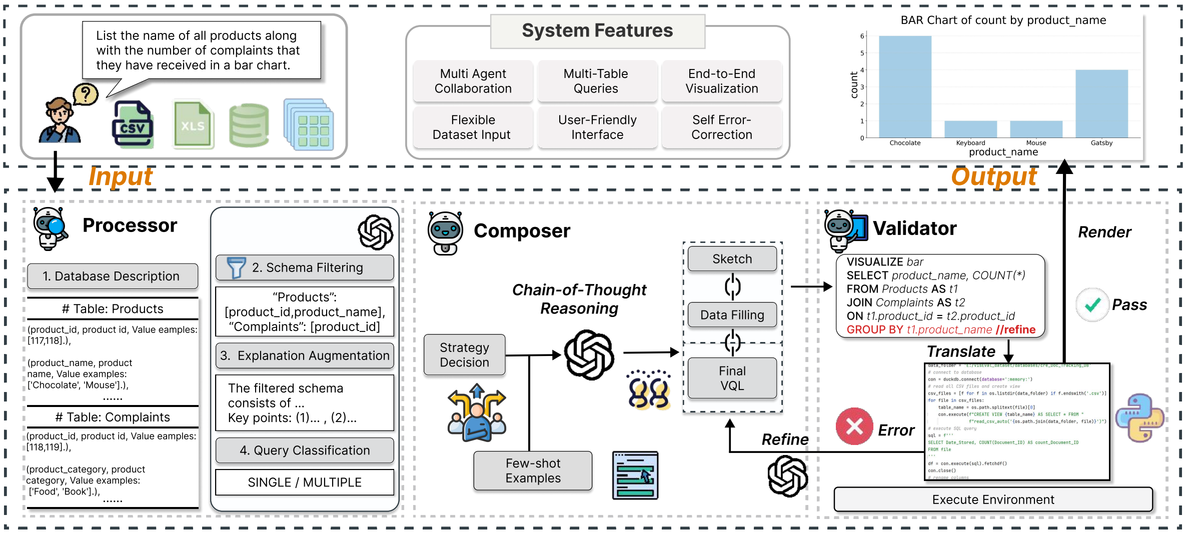


Figure 2: The overall pipeline of NVAGENT. We recommend a ***"Zoom in"*** to view its detailed design: **(1)** The *processor* agent performs schema filtering and context augmentation; **(2)** The *composer* agent generates structured VQL representations through sketch-and-fill reasoning; **(3)** The *validator* agent ensures visualization correctness via iterations of execution-guided validation and error-based refinement.

3.2 Processor Agent

To handle massive data and complex queries effectively, we design a *processor* agent that prepares and enriches input data. Specifically, the *processor* agent consists of four steps:

Database Description. The *processor* first constructs a comprehensive database description, which includes table and column schemas, with representative value examples. This provides the foundation for LLMs to understand the data structure and relationships. For instance, when processing a *"Products"* table, it extracts column details like *"product_id"*, and *"product_category"*, along with their value examples (*e.g.*, *"Chocolate"*, *"Book"*).

Schema Filtering. Subsequently, building on this foundation, the agent performs schema filtering to identify and extract tables and columns relevant to the user query (*e.g.*, filtering out unrelated columns like *"product_category"*), effectively reducing noise and preventing information overload.

Explanation Augmentation. To enable more accurate query interpretation, inspired by the self-augmented strategy (Sui et al., 2024), the *processor* generates augmented explanations for the filtered schema like *"Key points: (1) product_id in the table Products serves as a foreign key linking to the table Complaints"*. These explanations bring insights that provide additional context about table relationships and column semantics.

Query Classification. Finally, the agent classifies query complexity as either *single* or *multiple* based on the number of tables involved and the operations required. This classification guides subsequent agents in choosing appropriate strategies (*e.g.*, the *multiple* scenario requires join operations across tables or complex aggregations).

By providing a focused, well-explained schema and classification, the *processor* agent establishes a strong foundation of complex data understanding for the subsequent stages in our framework.

3.3 Composer Agent

The *composer* agent is designed to bridge the gap between natural language queries and visualization code, generating structured VQL queries through a step-by-step reasoning approach.

Strategy Decision. Based on the query classification from the *processor* agent, different strategies are adopted to plan the visualization generation. For example, *single* queries focus on basic aggregations, while *multiple* scenarios require more complex join operations.

Chain-of-Thought Reasoning. During the generation stage, the *composer* agent employs a chain-of-thought (Wei et al., 2022a) approach to break down the visualization process into manageable steps. This approach is complemented by providing few-shot examples for In-Context Learning, enhancing the model's adaptability to diverse query types.

Sketch-and-Fill Process. The reasoning process follows the *"sketch-and-fill"* paradigm and is structured into three steps, including sketch construction, data components filling, and final VQL composition (prompt shown in Appendix F).

Taking the query *"List the name of all products along with the number of complaints that they have received in a bar chart."* (shown in Figure 2) as an example, the *composer* initially determines the specific elements (*i.e.*, visualization type *"Bar"*) and constructs a VQL sketch (*e.g.*, *"Visualize bar SELECT _, COUNT(_) FROM _ JOIN _ ON _"*). Subsequently, it fills the data components (*e.g.*, the column *"product_name"*) into the sketch and then combines them to produce the complete VQL representation.

3.4 Validator Agent

The *validator* agent ensures the accuracy and executability of generated VQL queries through an iterative execution-guided validation and error-based refinement process.

Translation and Execution. When receiving a VQL representation, the *validator* first translates the query into executable Python code using visualization libraries like *"Matplotlib"*. The generated code is then executed in a sandboxed environment, where the agent captures either successful execution results or potential error messages.

Pass or Error. During the execution phase, the *validator* monitors the return information from the execution environment. If successful, it renders and returns the final visualization; otherwise, if errors occur (*e.g.*, syntax errors, or invalid column names), the agent captures specific error messages and routes them back to the *composer* agent, triggering the refinement process.

As illustrated in Figure 2(c), when the *validator* translates the VQL query *"VISUALIZE bar ... ON t1.product_id = t2.product_id"* into Python code and executes, it encounters an error message *"lack a 'group by' clause"*. This error message is then sent back to the *composer* agent, which refines the VQL query by adding *"GROUP BY product_name"* to ensure proper data aggregation.

Iterative Refinement. The *composer* agent iteratively refines its output based on feedback from the *validator* agent until a valid visualization is produced. If any errors are detected during validation, it receives error information and adjusts its output accordingly, ensuring the final VQL query is correct. Notably, we design the system to refine VQL query instead of Python code due to its simpler syntax for better correction.

4 Experiments and Analysis

4.1 Experimental Setup

Dataset. VisEval (Chen et al., 2024c) is a benchmark designed based on nvBench (Luo et al., 2021a) to assess the capabilities of LLMs in the NL2VIS task. It consists of 1,150 distinct visualizations (VIS) and 2,524 (NL, VIS) pairs across 146 databases, with accurately labeled ground truths and meta-information detailing feasible visualization options. The dataset is divided into *single-table* scenario and *multi-table* scenario. Moreover, visualizations are classified into four distinct levels of hardness: easy, medium, hard, and extra hard. Cases across different hardness levels can be found in Appendix E.

Baselines. We conduct our experiments compared with three formerly SOTA baselines[†]: Chat2Vis (Maddigan and Susnjak, 2023), which uses prompt engineering to generate visualizations from natural language descriptions; LIDA (Dibia, 2023), which employs a four-step process for incrementally translating natural language inputs into visualizations; and CoML4Vis (Zhang et al., 2023), which applies a few-shot prompt method integrating multiple tables for visualization tasks. More details can be found in Appendix B. We implement our approach and baselines using three different backbone models: GPT-4o (OpenAI, 2024b), GPT-4o-mini (OpenAI, 2024a), and GPT-3.5-turbo (OpenAI, 2022).

Evaluation Metrics. We evaluate the performance using both rule-based and model-based metrics for quantitative and qualitative assessment. **Invalid Rate** and **Illegal Rate** represent the percentages of visualizations that fail to render or meet query requirements, respectively. **Pass Rate** measures the proportion of valid and legal visualizations in the evaluation set. **Readability Score** is the average score ranging from 0 to 5 assigned by MLLM-as-a-Judge (Chen et al., 2024a; Ye et al.,

[†]We try the vanilla baseline similar to the GPT-4o with code interpreter in `https://platform.openai.com/docs/assistants/tools/code-interpreter`. Due to the API still in the beta stage and often failing, we do not include it as a baseline.

Method	Single-Table					Multi-Table				
	Invalid(↓)	Illegal(↓)	Pass(↑)	Read.(↑)	Qual.(↑)	Invalid(↓)	Illegal(↓)	Pass(↑)	Read.(↑)	Qual.(↑)
					GPT-4o					
CoML4Vis	**0.67%**	24.14%	75.17%	3.42	2.58	1.87%	26.27%	71.84%	3.45	2.48
LIDA	1.13%	21.20%	77.66%	2.53	1.99	14.80%	83.56%	1.62%	3.62	0.06
Chat2Vis	0.86%	21.37%	77.75%	**3.87**	3.02	38.74%	59.84%	1.40%	**3.76**	0.05
NVAGENT	0.72%	**13.63%**	**85.63%**	3.66	**3.13**	**1.34%**	**17.57%**	**81.07%**	3.61	**2.93**
Δ	-0.05%	+7.57%	+7.88%	-5.42%	+3.64%	+0.53%	+8.70%	+9.23%	-3.98%	+18.15%
					GPT-4o-mini					
CoML4Vis	**0.36%**	25.74%	73.88%	3.33	2.47	10.01%	33.06%	56.92%	3.24	1.86
LIDA	9.09%	23.04%	67.85%	3.10	2.12	17.61%	80.86%	1.51%	3.10	0.04
Chat2Vis	2.14%	25.92%	71.92%	**3.81**	2.76	35.78%	61.93%	2.27%	2.30	0.05
NVAGENT	1.97%	**22.86%**	**75.16%**	3.67	**2.77**	**8.15%**	**25.99%**	**65.85%**	**3.66**	**2.42**
Δ	-1.61%	+0.18%	+1.28%	-3.67%	+0.36%	+1.86%	+7.07%	+8.93%	+12.96%	+30.11%
					GPT-3.5-turbo					
CoML4Vis	6.17%	29.28%	64.54%	3.33	2.18	13.92%	30.09%	55.98%	3.37	1.93
LIDA	47.32%	15.84%	36.83%	3.32	1.23	62.57%	36.56%	0.86%	3.50	0.03
Chat2Vis	3.90%	28.11%	67.98%	3.03	2.08	40.77%	57.66%	1.55%	3.31	0.05
NVAGENT	**2.98%**	20.93%	**76.08%**	**3.58**	**2.72**	**7.18%**	**28.51%**	**64.29%**	**3.61**	**2.32**
Δ	+0.92%	-5.09%†	+8.10%	+7.51%	+24.77%	+6.74%	+1.58%	+8.11%	+3.14%	+20.21%

* Δ represents the percentage improvement or decrease of NVAGENT compared to the best-performing baseline for each metric.
For the first three columns, Δ is calculated using absolute differences, while for the last two columns, it is calculated as the relative change.
†: NVAGENT actually performs best, while LIDA has a lower Illegal due to its high Invalid rate.

Table 1: Performance of our approach with baselines using different backbone models.

2024) to assess their visual clarity for legal visualization. We assess MLLM-scoring by calculating the similarity of GPT-4o-mini and GPT-4o with human-annotated scores in a subset with 500 samples. Empirically, we select GPT-4o-mini as the vision model for judgment. More details are referred to the Appendix B. **Quality Score** is 0 for invalid or illegal visualizations, otherwise equal to the readability score.

4.2 Overall Performance

Table 1 shows the performance across different methods and backbone models. Generally, our proposed method, NVAGENT, demonstrates significant improvements over existing approaches across all metrics in both *single-* and *multi-table* scenarios, particularly on pass rate and quality score. Furthermore, NVAGENT achieves an impressive 85.63% pass rate and a quality score of 3.13 in *single-table* scenarios using GPT-4o, surpassing all baseline methods. In more complex *multi-table* scenarios, NVAGENT maintains strong performance, significantly outperforming other approaches. Specifically, using GPT-4o, our method attains an 81.07% pass rate and a quality score of 2.93 for *multi-table* queries, exceeding the previous state-of-the-art by 18.15%. The minimal performance gap between *single-* and *multi-table* scenarios (85.63% vs. 81.07% pass rate) underscores NVAGENT's consistency and adaptability across varying query complexities, a crucial advantage in real-world applications where multi-table queries are common.

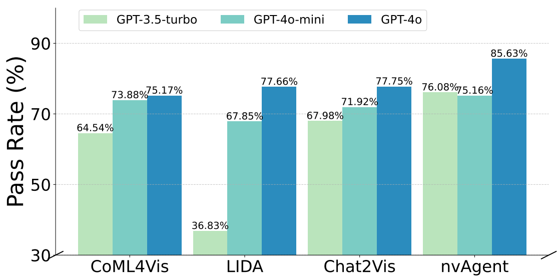


Figure 3: Integrating better LLMs as backbones (*i.e.*, GPT-4o) can bring higher pass rates.

4.3 Effectiveness of Each Agent

To evaluate the effectiveness of each component in NVAGENT, we conducted comprehensive ablation experiments. We perform agent workflow ablation studies with GPT-4o to assess the contributions of each agent, as shown in Table 2. From this table, we observe that the *composer* is the most critical component, as its removal leads to significant drops in the overall pass rate—22.39% with GPT-3.5-turbo and 59.81% with GPT-4o. The *validator* also proves vital, as its absence leads to a

Method	Single-Table			Multi-Table			Average
	Invalid	Illegal	Pass	Invalid	Illegal	Pass	Pass Rate
GPT-4o							
NVAGENT(4-shot)	0.72%	13.63%	**85.63%**	1.34%	17.57%	81.07%	83.80%
w/o Processor	0.62%	14.27%	85.09%	1.26%	16.42%	**82.31%**	**83.97%**
w/o Composer	1.20%	74.56%	24.22%	2.34%	74.00%	23.64%	23.99%
w/o Validator	5.80%	12.22%	81.96%	7.01%	15.95%	77.02%	79.98%
GPT-3.5-turbo							
NVAGENT(4-shot)	2.98%	20.93%	76.08%	7.18%	28.51%	**64.29%**	**71.35%**
w/o Processor	3.01%	20.15%	**76.82%**	9.38%	31.01%	59.60%	69.92%
w/o Composer	18.78%	30.97%	50.24%	25.02%	27.92%	47.05%	48.96%
w/o Validator	18.04%	17.50%	64.45%	22.64%	21.40%	55.94%	61.04%

Table 2: Ablation results of each agent within NVAGENT.

Method	Single-Table			Multi-Table			Average
	Invalid	Illegal	Pass	Invalid	Illegal	Pass	Pass Rate
nvAgent(4-shot)	2.98%	20.93%	76.08%	7.18%	28.51%	**64.29%**	**71.35%**
w/o schema filtering	3.36%	20.09%	**76.53%**	12.08%	30.14%	57.77%	69.01%
w/o aug. explanation	3.23%	20.69%	76.06%	7.10%	30.87%	62.01%	70.44%
w/o complex. classifi.	4.77%	21.42%	73.79%	7.50%	29.80%	62.69%	69.34%
w/o CoT	15.81%	16.91%	67.27%	17.73%	24.40%	57.86%	63.50%
w/o ICL	26.80%	24.92%	48.27%	31.91%	28.41%	39.66%	44.82%

Table 3: Ablation results of each module within NVAGENT's agentic workflow.

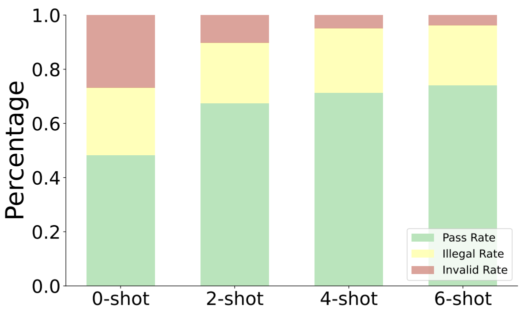


Figure 4: More examples for in-context learning bring higher pass rate, using GPT-3.5-turbo.

3.82% decrease for GPT-4o and a sharper decrease of 10.31% using GPT-3.5-turbo, primarily due to increased invalid rate, confirming the effectiveness of the post-processing stage.

Interestingly, while the *processor*'s removal shows only a slight overall performance decline (1.43%), its impact varies across scenarios: a marginal improvement in *single-table* cases but a notable decrease (4.69%) in *multi-table* scenarios. This pattern is particularly pronounced when using GPT-3.5-turbo, highlighting the *processor*'s critical role in handling complex database information. However, more capable models like GPT-4o may occasionally find this additional processing step redundant, as similarly observed in *"The Death of Schema Linking"* (Maamari et al., 2024).

4.4 Impact of LLM Backbones

Figure 3 illustrates the performance of different methods across three backbone LLMs in *single-table* scenarios. It can be observed that the pass rate positively correlates with the capacity of the backbone LLMs. However, an intriguing phenomenon was noted: using GPT-4o-mini resulted in a slight decrease in performance compared to GPT-3.5-turbo. This unexpected outcome suggests potential limitations in GPT-4o-mini's reasoning abilities for this specific task, despite its overall advancements.

4.5 Impact of Prompting Techniques

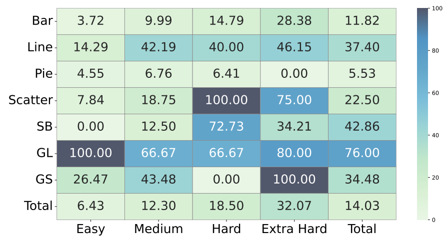
Further ablation results of individual prompting techniques within each agent using GPT-3.5-turbo are demonstrated in Table 3. From this table, we observe that all three techniques in *processor* show similar results. However, the schema filtering proves more beneficial for *multi-table* scenarios (6.52%), while complexity classification benefits *single-table* scenarios (2.29%). In the *composer* agent, the sharp decrease (26.53%) upon removal of in-context learning demonstrates the critical role of example-based prompts in task com-

Setting	Invalid	Illegal	Pass	Tokens
VQL Refine	4.66%	**23.97%**	**71.36%**	**1179**
Code Refine	**4.11%**	25.51%	70.35%	1365

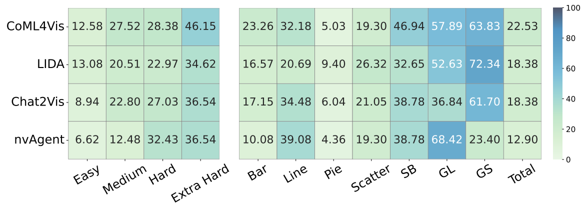
Table 4: Exploration study of Python code refinement. Tokens represent the usage in the refinement stage.

Method	Single-Table		Multi-Table	
	Elo	95% CI	Elo	95% CI
NVAGENT	**1538.27**	+2.95/-2.95	**1529.86**	+2.83/-2.84
CoML4Vis	1506.71	+3.00/-3.00	1514.96	+3.00/-3.00
Chat2Vis	1496.71	+3.05/-3.05	1499.44	+3.01/-3.01
LIDA	1458.31	+2.85/-2.85	1455.74	+2.94/-2.93

Table 5: Elo rankings on *single-* and *multi-table* test sets. NVAGENT scores the highest in both scenarios.



(a) Error distribution of NVAGENT.



(b) Errors of different models in *single-table* dataset.

Figure 5: Error distributions with hardness and chart type. SB, GL, and GS refer to Stacked Bar, Grouping Line, and Grouping Scatter, respectively.

prehension, and the significant increase in Invalid Rate also highlights the step-by-step VQL generation. Moreover, as shown in Table 4, we conduct an exploration study for *validator* to refine Python code directly and find that the pass rate decreased by 1.01%, indicating the effectiveness of using VQL for correction. We also include several exploration experiments in Appendix C.

We carefully design diverse examples including various visualization types (*e.g.*, grouping scatter) and binning operations (*e.g.*, Year, Weekday) for prompting LLM, and Figure 4 illustrates the impact of increasing the number of examples in the prompt. The observed improvement in pass rate suggests that the language model effectively leverages knowledge from few-shot prompts.

4.6 Qualitative Analysis

ELO Score. We adopt the ELO rating system (Elo and Sloan, 1978), a widely-used method for calculating relative skill levels, to evaluate model performance. We conduct this experiment in 1000 example pairs from *single-* and *multi-table* datasets with equal weights for different models, using human judgments to assess the accuracy of natural language queries. The results in Table 5 show that our NVAGENT outperforms other baselines, highlighting its capability to manage complex queries and produce relevant visualizations. Implementation details are in Appendix B.

Case Study. Figure 6 presents three cases illustrating NL queries and their visualizations generated by NVAGENT and baseline models. The examples showcase NVAGENT's superior performance. In the first case, NVAGENT correctly orders data by the X-axis, while Chat2Vis and CoML4Vis use the Y-axis. The second case highlights NVAGENT's accurate grouping in a stacked bar chart, unlike the baselines. In the third case, involving a *multi-table* query, NVAGENT effectively joins tables and groups data for a line chart, whereas Chat2Vis struggles with the structure, and CoML4Vis overlooks the where condition.

Error Analysis. As shown in Figure 5, NVAGENT's performance varies significantly across chart type and difficulty level, particularly with rare queries in temporal data, such as line charts. Our error analysis reveals that failures stem from insufficient handling of temporal information and an imperfect translate function for time-series binning operations. These challenges related to chart complexity and task difficulty underscore the need for better tabular data understanding in LLMs. Our future work can be focused on improving the reasoning abilities of LLMs in temporal information in tabular data.

5 Related Work

NL2VIS. NL2VIS research has evolved from rule-based systems (Narechania et al., 2020; Srinivasan and Stasko, 2017; Yu and Silva, 2019; Gao et al., 2015; Luo et al., 2018) and neural network-based approaches (Markel et al., 2002; Luo et al., 2021c; Song et al., 2022), to most recently to generated model enhanced systems (Hong et al., 2024). Current LLM-based approaches can be broadly categorized into two

Figure 6: Case study of visualization performed by NVAGENT and other baselines. The first two cases are from *single-table* dataset and the third from *multi-table* dataset. NVAGENT performed well in most complex cases (*e.g.*, stacked bar charts), while other baselines failed.

groups: (1) those utilizing prompt engineering techniques, such as Chat2Vis (Maddigan and Susnjak, 2023), Prompt4Vis (Li et al., 2024b), Mirror (Xu et al., 2023), LIDA (Dibia, 2023), and Data Formulator (Wang et al., 2024b), and (2) those involving fine-tuning of models specifically for NL2VIS tasks, like TableGPT (Zha et al., 2023; Su et al., 2024), ChartLlama (Han et al., 2023) and DataVis-T5 (Wan et al., 2024). This evolution marks significant progress in making data visualization more accessible and intuitive.

LLM for Tabular Data. LLM-based approaches push the performance of tabular data processing to a new boundary (Liu et al., 2024). The emergent in-context learning capability (Dong et al., 2022) and chain-of-thought reasoning (Wei et al., 2022b) have significantly enhanced LLMs' ability to handle complex tabular tasks by mimicking examples and encouraging step-by-step thinking (Min et al., 2022; Zhang et al., 2022; Wu et al., 2024a). These advancements have been particularly impactful in several key tasks such as TableQA (Qiu et al., 2024; Xu et al., 2024), Text2SQL (Wu et al., 2024c; Pourreza and Rafiei, 2024), NL2Formula (Zhao et al., 2024) and NL2VIS (Yang et al., 2024; Li et al., 2024b; Tian et al., 2024).

Agentic Workflow. Agentic workflow leverages multiple LLM-based agents, each assigned different roles to tackle complex problems (Talebirad and Nadiri, 2023). These systems employ various interaction modes, such as collaboration (Chan et al., 2023; Li et al., 2023; Wu et al., 2023) or competition (Zhao et al., 2023), showing remarkable success in database query tasks (Wang et al., 2024a; Zhu et al., 2024; Cen et al., 2024), software development (Hong et al., 2023; Islam et al., 2024; Huang et al., 2024) and mathematical reasoning (Chen et al., 2024b). This success stems from the synergy of specialized agents working together to overcome individual limitations and solve complex tasks efficiently.

6 Conclusion

In this paper, we have proposed NVAGENT, a collaborative agent-based workflow to solve the challenging mult-table NL2VIS task and provide a *"turnkey solution"* for users. NVAGENT decomposes the process into atomic modules such as database preprocessing, visualization planning, and iterative optimization. Experimental results show that NVAGENT outperforms state-of-the-art baselines by 7.88% in single-table and 9.23% in multi-table scenarios, demonstrating the efficacy of NVAGENT.

Limitations

While NVAGENT demonstrates significant improvements in NL2VIS tasks, we acknowledge several limitations. Our reliance on proprietary APIs may constrain the system's reproducibility and adaptability. Additionally, utilizing large language models as both backbone and evaluator introduces potential biases that could affect output quality and evaluation accuracy. Moreover, our error analysis finds insufficient handling of temporal information, which underscores the need for better tabular data understanding capabilities of LLMs. Our prompting strategy and evaluation metrics may not fully capture the nuances of complex visualizations or semantic correctness. The current framework employs a simple function to translate VQL into Python code, which can be further optimized for better readability. Future work should address these limitations by exploring open-source alternatives, developing more sophisticated prompting and evaluation techniques, and integrating advanced tools like retrieval augmented generation to enhance the system's capabilities and mitigate biases.

References

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.

Jipeng Cen, Jiaxin Liu, Zhixu Li, and Jingjing Wang. 2024. Sqlfixagent: Towards semantic-accurate sql generation via multi-agent collaboration. *arXiv preprint arXiv:2406.13408*.

Chi-Min Chan, Weize Chen, Yusheng Su, Jianxuan Yu, Wei Xue, Shanghang Zhang, Jie Fu, and Zhiyuan Liu. 2023. Chateval: Towards better llm-based evaluators through multi-agent debate. *arXiv preprint arXiv:2308.07201*.

Dongping Chen, Ruoxi Chen, Shilin Zhang, Yinuo Liu, Yaochen Wang, Huichi Zhou, Qihui Zhang, Yao Wan, Pan Zhou, and Lichao Sun. 2024a. Mllm-as-a-judge: Assessing multimodal llm-as-a-judge with vision-language benchmark. *arXiv preprint arXiv:2402.04788*.

Justin Chih-Yao Chen, Archiki Prasad, Swarnadeep Saha, Elias Stengel-Eskin, and Mohit Bansal. 2024b. Magicore: Multi-agent, iterative, coarse-to-fine refinement for reasoning.

Nan Chen, Yuge Zhang, Jiahang Xu, Kan Ren, and Yuqing Yang. 2024c. Viseval: A benchmark for data visualization in the era of large language models. *IEEE Transactions on Visualization and Computer Graphics*.

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

Qingxiu Dong, Lei Li, Damai Dai, Ce Zheng, Jingyuan Ma, Rui Li, Heming Xia, Jingjing Xu, Zhiyong Wu, Tianyu Liu, et al. 2022. A survey on in-context learning. *arXiv preprint arXiv:2301.00234*.

Arpad E Elo and Sam Sloan. 1978. The rating of chessplayers: Past and present. *(No Title)*.

Carly Fiorina. The goal is to turn data into information, and information into insight.

Tong Gao, Mira Dontcheva, Eytan Adar, Zhicheng Liu, and Karrie G Karahalios. 2015. Datatone: Managing ambiguity in natural language interfaces for data visualization. In *Proceedings of the 28th annual acm symposium on user interface software & technology*, pages 489–500.

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*.

Sirui Hong, Yizhang Lin, Bangbang Liu, Binhao Wu, Danyang Li, Jiaqi Chen, Jiayi Zhang, Jinlin Wang, Lingyao Zhang, Mingchen Zhuge, et al. 2024. Data interpreter: An llm agent for data science. *arXiv preprint arXiv:2402.18679*.

Sirui Hong, Xiawu Zheng, Jonathan Chen, Yuheng Cheng, Jinlin Wang, Ceyao Zhang, Zili Wang, Steven Ka Shing Yau, Zijuan Lin, Liyang Zhou, et al. 2023. Metagpt: Meta programming for multi-agent collaborative framework. *arXiv preprint arXiv:2308.00352*.

Dong Huang, Jie M. Zhang, Michael Luck, Qingwen Bu, Yuhao Qing, and Heming Cui. 2024. Agentcoder: Multi-agent-based code generation with iterative testing and optimisation.

Md. Ashraful Islam, Mohammed Eunus Ali, and Md Rizwan Parvez. 2024. Mapcoder: Multi-agent code generation for competitive problem solving.

Arshad Khan. 2024. Data visualization. In *Visual Analytics for Dashboards: A Step-by-Step Guide to Principles and Practical Techniques*, pages 67–73. Springer.

Guohao Li, Hasan Hammoud, Hani Itani, Dmitrii Khizbullin, and Bernard Ghanem. 2023. Camel: Communicative agents for" mind" exploration of large language model society. *Advances in Neural Information Processing Systems*, 36:51991–52008.

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.

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. *arXiv preprint arXiv:2402.07909*.

Xinyu Liu, Shuyu Shen, Boyan Li, Peixian Ma, Runzhi Jiang, Yuyu Luo, Yuxin Zhang, Ju Fan, Guoliang Li, and Nan Tang. 2024. A survey of nl2sql with large language models: Where are we, and where are we going? *arXiv preprint arXiv:2408.05109*.

Weizheng Lu, Jing Zhang, Ju Fan, Zihao Fu, Yueguo Chen, and Xiaoyong Du. 2024. Large language model for table processing: A survey. *arXiv preprint arXiv:2402.05121*.

Yuyu Luo, Xuedi Qin, Nan Tang, and Guoliang Li. 2018. Deepeye: Towards automatic data visualization. In *2018 IEEE 34th international conference on data engineering (ICDE)*, pages 101–112. IEEE.

Yuyu Luo, Jiawei Tang, and Guoliang Li. 2021a. nvbench: A large-scale synthesized dataset for cross-domain natural language to visualization task. *arXiv preprint arXiv:2112.12926*.

Yuyu Luo, Nan Tang, Guoliang Li, Chengliang Chai, Wenbo Li, and Xuedi Qin. 2021b. Synthesizing natural language to visualization (nl2vis) benchmarks from nl2sql benchmarks. In *Proceedings of the 2021 International Conference on Management of Data, SIGMOD Conference 2021, June 20–25, 2021, Virtual Event, China*. ACM.

Yuyu Luo, Nan Tang, Guoliang Li, Chengliang Chai, Wenbo Li, and Xuedi Qin. 2021c. Synthesizing natural language to visualization (nl2vis) benchmarks from nl2sql benchmarks. In *Proceedings of the 2021 International Conference on Management of Data*, pages 1235–1247.

Karime Maamari, Fadhil Abubaker, Daniel Jaroslawicz, and Amine Mhedhbi. 2024. The death of schema linking? text-to-sql in the age of well-reasoned language models. *arXiv preprint arXiv:2408.07702*.

Paula Maddigan and Teo Susnjak. 2023. Chat2vis: generating data visualizations via natural language using chatgpt, codex and gpt-3 large language models. *Ieee Access*, 11:45181–45193.

Tony Markel, Aaron Brooker, Terry Hendricks, Valerie Johnson, Kenneth Kelly, Bill Kramer, Michael O'Keefe, Sam Sprik, and Keith Wipke. 2002. Advisor: a systems analysis tool for advanced vehicle modeling. *Journal of power sources*, 110(2):255–266.

Sewon Min, Xinxi Lyu, Ari Holtzman, Mikel Artetxe, Mike Lewis, Hannaneh Hajishirzi, and Luke Zettlemoyer. 2022. Rethinking the role of demonstrations: What makes in-context learning work? *arXiv preprint arXiv:2202.12837*.

Arpit Narechania, Arjun Srinivasan, and John Stasko. 2020. Nl4dv: A toolkit for generating analytic specifications for data visualization from natural language queries. *IEEE Transactions on Visualization and Computer Graphics*, 27(2):369–379.

OpenAI. 2022. Chatgpt (gpt-3.5). `https://openai.com/index/chatgpt/`.

OpenAI. 2024a. Gpt-4o mini: Advancing cost-efficient intelligence. `https://openai.com/index/gpt-4o-mini-advancing-cost-efficient-intelligence/`. Accessed: 2024-09-04.

OpenAI. 2024b. Hello gpt-4o. Accessed: 2024-06-06.

Mohammadreza Pourreza and Davood Rafiei. 2024. Din-sql: Decomposed in-context learning of text-to-sql with self-correction. *Advances in Neural Information Processing Systems*, 36.

Zipeng Qiu, You Peng, Guangxin He, Binhang Yuan, and Chen Wang. 2024. Tqa-bench: Evaluating llms for multi-table question answering with scalable context and symbolic extension. *arXiv preprint arXiv:2411.19504*.

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*.

Yuanfeng Song, Xuefang Zhao, Raymond Chi-Wing Wong, and Di Jiang. 2022. Rgvisnet: A hybrid retrieval-generation neural framework towards automatic data visualization generation. In *Proceedings of the 28th ACM SIGKDD Conference on Knowledge Discovery and Data Mining*, pages 1646–1655.

Arjun Srinivasan and John Stasko. 2017. Orko: Facilitating multimodal interaction for visual exploration and analysis of networks. *IEEE transactions on visualization and computer graphics*, 24(1):511–521.

Aofeng Su, Aowen Wang, Chao Ye, Chen Zhou, Ga Zhang, Guangcheng Zhu, Haobo Wang, Haokai Xu, Hao Chen, Haoze Li, Haoxuan Lan, Jiaming Tian, Jing Yuan, Junbo Zhao, Junlin Zhou, Kaizhe Shou, Liangyu Zha, Lin Long, Liyao Li, Pengzuo Wu, Qi Zhang, Qingyi Huang, Saisai Yang, Tao Zhang, Wentao Ye, Wufang Zhu, Xiaomeng Hu, Xijun Gu, Xinjie Sun, Xiang Li, Yuhang Yang, and Zhiqing Xiao. 2024. Tablegpt2: A large multimodal model with tabular data integration.

Yuan Sui, Mengyu Zhou, Mingjie Zhou, Shi Han, and Dongmei Zhang. 2024. Table meets llm: Can large language models understand structured table data? a benchmark and empirical study. In *Proceedings* of the 17th ACM International Conference on Web Search and Data Mining*, pages 645–654.

Yashar Talebirad and Amirhossein Nadiri. 2023. Multi-agent collaboration: Harnessing the power of intelligent llm agents. *arXiv preprint arXiv:2306.03314*.

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*.

Manasi Vartak, Silu Huang, Tarique Siddiqui, Samuel Madden, and Aditya Parameswaran. 2017. Towards visualization recommendation systems. *Acm Sigmod Record*, 45(4):34–39.

Zhuoyue Wan, Yuanfeng Song, Shuaimin Li, Chen Jason Zhang, and Raymond Chi-Wing Wong. 2024. Datavist5: A pre-trained language model for jointly understanding text and data visualization. *arXiv preprint arXiv:2408.07401*.

Bing Wang, Changyu Ren, Jian Yang, Xinnian Liang, Jiaqi Bai, Linzheng Chai, Zhao Yan, Qian-Wen Zhang, Di Yin, Xing Sun, and Zhoujun Li. 2024a. Mac-sql: A multi-agent collaborative framework for text-to-sql.

Chenglong Wang, John Thompson, and Bongshin Lee. 2024b. Data formulator: Ai-powered concept-driven visualization authoring. *IEEE Transactions on Visualization and Computer Graphics*, 30(1):1128–1138.

Michael L Waskom. 2021. Seaborn: statistical data visualization. *Journal of Open Source Software*, 6(60):3021.

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

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

Qingyun Wu, Gagan Bansal, Jieyu Zhang, Yiran Wu, Shaokun Zhang, Erkang Zhu, Beibin Li, Li Jiang, Xiaoyun Zhang, and Chi Wang. 2023. Autogen: Enabling next-gen llm applications via multi-agent conversation framework. *arXiv preprint arXiv:2308.08155*.

Xianjie Wu, Jian Yang, Linzheng Chai, Ge Zhang, Jiaheng Liu, Xinrun Du, Di Liang, Daixin Shu, Xianfu Cheng, Tianzhen Sun, et al. 2024a. Tablebench: A comprehensive and complex benchmark for table question answering. *arXiv preprint arXiv:2408.09174*.

Yang Wu, Yao Wan, Hongyu Zhang, Yulei Sui, Wucai Wei, Wei Zhao, Guandong Xu, and Hai Jin. 2024b. Automated data visualization from natural language via large language models: An exploratory study. *Proceedings of the ACM on Management of Data*, 2(3):1–28.

Zhenhe Wu, Zhongqiu Li, Jie Zhang, Mengxiang Li, Yu Zhao, Ruiyu Fang, Zhongjiang He, Xuelong Li, Zhoujun Li, and Shuangyong Song. 2024c. Rb-sql: A retrieval-based llm framework for text-to-sql. *arXiv preprint arXiv:2407.08273*.

Canwen Xu, Julian McAuley, and Penghan Wang. 2023. Mirror: A natural language interface for data querying, summarization, and visualization. In *Companion Proceedings of the ACM Web Conference 2023*, WWW '23, page 49–52. ACM.

Yao Xu, Shizhu He, Zeng Xiangrong, Jiabei Chen, Guang Liu, Bingning Wang, Jun Zhao, and Kang Liu. 2024. Llasa: Large language and structured data assistant.

Hao Yang, Zhaoyong Yang, Ruyang Zhao, Xiaoran Li, and Gaoqi Rao. 2024. The implementation solution for automatic visualization of tabular data in relational databases based on large language models. In *2024 International Conference on Asian Language Processing (IALP)*, pages 175–180. IEEE.

Jiayi Ye, Yanbo Wang, Yue Huang, Dongping Chen, Qihui Zhang, Nuno Moniz, Tian Gao, Werner Geyer, Chao Huang, Pin-Yu Chen, et al. 2024. Justice or prejudice? quantifying biases in llm-as-a-judge. *arXiv preprint arXiv:2410.02736*.

Shoulin Yin, Hang Li, Yang Sun, Muhammad Ibrar, and Lin Teng. 2024. Data visualization analysis based on explainable artificial intelligence: A survey. *IJLAI Transactions on Science and Engineering*, 2(2):13–20.

Bowen Yu and Cláudio T Silva. 2019. Flowsense: A natural language interface for visual data exploration within a dataflow system. *IEEE transactions on visualization and computer graphics*, 26(1):1–11.

Liangyu Zha, Junlin Zhou, Liyao Li, Rui Wang, Qingyi Huang, Saisai Yang, Jing Yuan, Changbao Su, Xiang Li, Aofeng Su, et al. 2023. Tablegpt: Towards unifying tables, nature language and commands into one gpt. *arXiv preprint arXiv:2307.08674*.

Lei Zhang, Yuge Zhang, Kan Ren, Dongsheng Li, and Yuqing Yang. 2023. Mlcopilot: Unleashing the power of large language models in solving machine learning tasks. *arXiv preprint arXiv:2304.14979*.

Zhuosheng Zhang, Aston Zhang, Mu Li, and Alex Smola. 2022. Automatic chain of thought prompting in large language models. *arXiv preprint arXiv:2210.03493*.

Qinlin Zhao, Jindong Wang, Yixuan Zhang, Yiqiao Jin, Kaijie Zhu, Hao Chen, and Xing Xie. 2023. Competeai: Understanding the competition behaviors in large language model-based agents. *arXiv preprint arXiv:2310.17512*.

Wei Zhao, Zhitao Hou, Siyuan Wu, Yan Gao, Haoyu Dong, Yao Wan, Hongyu Zhang, Yulei Sui, and Haidong Zhang. 2024. Nl2formula: Generating spreadsheet formulas from natural language queries. *arXiv preprint arXiv:2402.14853*.

Jun-Peng Zhu, Peng Cai, Kai Xu, Li Li, Yishen Sun, Shuai Zhou, Haihuang Su, Liu Tang, and Qi Liu. 2024. Autotqa: Towards autonomous tabular question answering through multi-agent large language models. *Proceedings of the VLDB Endowment*, 17(12):3920–3933.

A Framework Details

Our framework is described in Algorithm 1, and compared with former baselines in Table 6. Distinct with several methods generating Python code for visualization directly, we use VQL as an intermediate representation to bridge natural language queries and visualization code. Additionally, our framework can be easily optimized by adding some useful tools such as Retrieval Augmented Generation. Moreover, our method supports handling multi-table data and the visualization can be customized according to humans' preferences. Our framework utilizes the agent-based collaborative workflow, which consists of data preprocessing, generation, and error correction, organized with the modular design.

Algorithm 1 NVAGENT Framework

```
function NL2VIS(Q, S)
    Initialize Mem ← {Q, S}
    (S′, A) ← PROCESSOR(Mem)
    Mem.update(S′, A)
    V ← COMPOSER(Mem)
    Mem.update(V)
    Chart, isValid ← VALIDATOR(Mem)
    while not isValid do
        V ← REFINE(Mem)
        Mem.update(V)
        Chart, isValid ← VALIDATOR(Mem)
    end while
    return Chart
end function
```

B Detailed Experiment Setups

Baselines. We implemented our experiment compared with three recent baselines. (We also tried to use Code Interpreter as a baseline, but due to the rate limit of API constraint, the evaluation failed to generate visualizations via direct .csv files)

- **Chat2Vis** (Maddigan and Susnjak, 2023): This approach generates data visualizations by leveraging prompt engineering to translate natural language descriptions into visualizations. It uses a language-based table description, which includes column types and sample values, to inform the visualization generation process.
- **LIDA** (Dibia, 2023): This tool structures visualization generation as a four-step process, where each step builds on the previous one to incrementally translate natural language inputs into visualizations. It uses a JSON format to describe column statistics and samples, making it adaptable across various visualization tasks.
- **CoML4Vis** (Zhang et al., 2023): Building on a data science code generation framework, CoML4Vis utilizes a few-shot prompt that integrates multiple tables into a single visualization task. It summarizes data table information, including column names and samples, and then applies a few-shot prompt to guide visualization generation.

Metrics. Our evaluation framework involves five main metrics:

- **Invalid Rate** represents the percentage of visualizations that fail to render due to issues like incorrect API usage or other code errors.
- **Illegal Rate** indicates the percentage of visualizations that do not meet query requirements, which can include incorrect data transformations, mismatched chart types, or improper visualizations.
- **Readability Score** is the average score (range 1-5) assigned by a vision language model, like GPT-4V, for valid and legal visualizations, assessing their visual clarity and ease of interpretation.
- **Pass Rate** measures the proportion of visualizations in the evaluation set that are both valid (able to render) and legal (meet the query requirements).
- **Quality Score** is set to 0 for invalid or illegal visualizations; otherwise, it is equal to the readability score, providing an overall assessment of visualization quality factoring in both functionality and clarity.

Following metrics are detailed evaluations of each metric above:

- **Code Execution Check** verifies that the Python code generated by the model can be successfully executed.
- **Surface-form Check** ensures that the generated code includes necessary elements to produce a visualization like function calls to display the chart.
- **Chart Type Check** verifies whether the extracted chart type from the visualization matches the ground truth.

Framework	System Features		Visualization Capabilities		Agentic Workflow		
	VQL as Thoughts	Extensible Optimization	Multi-Table Support	Customizable Styling	Data Preprocess	Modular Design	Error-Correction
Chat2VIS (Maddigan and Susnjak, 2023)	✘	✘	✘	✘	✔	✘	✘
Mirror (Xu et al., 2023)	✘	✘	✘	✘	✘	✔	✘
LIDA (Dibia, 2023)	✘	✔	✘	✔	✔	✔	✘
CoML4VIS (Zhang et al., 2023)	✘	✘	✔	✘	✔	✘	✘
Prompt4VIS (Li et al., 2024b)	✔	✘	✔	✘	✔	✔	✘
CoT-Vis (Yang et al., 2024)	✔	✘	✘	✘	✔	✘	✘
NVAGENT (Ours)	✔	✔	✔	✔	✔	✔	✔

Table 6: Comparison of Different NL2VIS Frameworks.

- **Data Check** assesses if the data used in the visualization matches the ground truth, taking into consideration potential channel swaps based on specified channels.
- **Order Check** evaluates whether the sorting of visual elements follows the specified query requirements.
- **Layout Check** examines issues like text overflow or element overlap within visualizations.
- **Scale & Ticks Check** ensures that scales and ticks are appropriately chosen, avoiding unconventional representations.
- **Overall Readability Rating** integrates various readability checks to provide a comprehensive score considering layout, scale, text clarity, and arrangement.

For all evaluation results, these metrics are averaged across the dataset to provide an overarching view of model performance. These metrics collectively ensure that visualizations are not only correct in terms of execution but also effective in communicating the intended data narratives.

Model	P-corr	P-value
GPT-4o-mini	**0.6503**	0.000
GPT-4o	0.5648	0.000

Table 7: The Pearson correlations of GPT-4o-mini and GPT-4o with human judgments on readability scores.

Implement Details. Our system is implemented in Python 3.9, utilizing GPT-4o (OpenAI, 2024b), GPT-4o-mini (OpenAI, 2024a), and GPT-3.5-turbo (OpenAI, 2022) as the backbone model for all approaches, with the temperature set to 0 for consistent outputs. GPT-4o-mini serves as the vision language model for readability evaluation. We interact with these models through the AzureOpenAI API. The specific prompt templates for each agent, crucial for guiding their respective roles in the visualization generation process, are detailed in Appendix F. Token usages of NVAGENT and baselines are demonstrated in Table 8, and usage for each agent in our NVAGENT is shown in Table 9. Additionally, our evaluations are conducted in VisEval Benchmark (with MIT licsense).

Human Annotation. The annotation is conducted by 5 authors of this paper independently. As acknowledged, the diversity of annotators plays a crucial role in reducing bias and enhancing the reliability of the benchmark. These annotators have knowledge in the data visualization domain, with different genders, ages, and educational backgrounds. The educational backgrounds of annotators are above undergraduate. To ensure the annotators can proficiently mark the data, we provide them with detailed tutorials, teaching them how to judge the quality of data visualization. We also provide them with detailed criteria and task requirements in each annotation process shown in Figure 9. Two experiments requiring human annotation are detailed as follows:

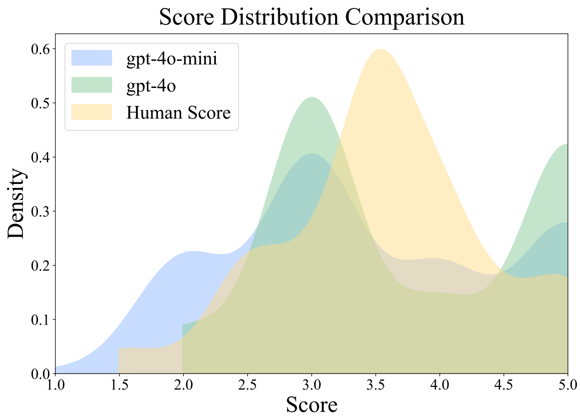


Figure 7: Comparison of score density distribution between GPT-4o, GPT-4o-mini and human average score.

- **Pearson Correlation of Visual Language Model.** We conduct human annotation frame-

Method	Single Table			Multiple Tables		
	prompt	response	total	prompt	response	total
LIDA	1386.23	237.90	1624.13	N/A		
Chat2VIS	414.35	451.30	865.65	N/A		
CoML4VIS	2614.76	279.86	2894.62	3069.62	307.67	3377.29
nvAgent	5122.99	777.63	5900.62	5613.96	1014.10	6628.06

Table 8: Token usage comparison for different methods. N/A indicates that LIDA and Chat2Vis cannot handle multiple table scenarios.

Agent	#Input	#Output	#Total
Processor	1486.07	569.58	1755.65
Composer	3268.32	221.74	3490.07
Validator	1051.82	127.85	1179.67

Table 9: Token usage of three agents in nvAgent.

works to compare the ability of the visual language model for MLLM-as-a-Judge, providing the readability score. Our annotation framework is shown in Figure 9. The final Pearson scores are demonstrated in Table 7, with its density distribution in Figure 7. The detailed instructions can be found in Figure 10.

- **Qualitative comparison to calculate ELO Scores.** We conduct human-judgments evaluations to compare which visualization generated by different models meets the query requirement more precisely. The leaderboard is shown in Table 5, and Figure 11 shows the judgment framework. Each model starts with a base ELO score of 1500. After each pairwise comparison, the scores are updated based on the outcome and the current scores of the models involved. The hyperparameters are set as follows: the K-factor is set to 32, which determines the maximum change in rating after a single comparison. We conducted two sets of evaluations: one for single-table queries and another for multiple-table queries, with 1000 bootstrap iterations for each set to ensure statistical robustness. The evaluation process involved presenting human judges with a query and two visualizations, asking them to select the one that better meets the query requirements. This process was repeated across all model pairs and queries in our test set. The detailed guidance provided to the human evaluators can be found in Figure 12, which outlines the criteria for judging visualization quality and relevance to the given query.

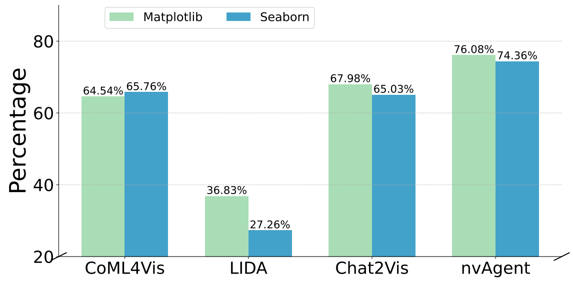


Figure 8: Performance of different models using Matplotlib and Seaborn library, using GPT-3.5-turbo.

C Additional Experiment Results

We also conducted a comparison experiment of different methods using matplotlib or seaborn library. Figure 8 demonstrates the results, indicating that our method outperforms obviously other baselines not only with matplotlib but also seaborn.

In addition, we test techniques in the Validator Agent, such as Chain-of-Thought. As is shown in Table 10, integrating Chain-of-Thought reasoning, may affect its performance badly, likely due to the simple refining task with complex reasoning. Moreover, using the original schema to check for false schema filtering seems to be useless in this case.

D Evaluation Results with Detailed Metrics

We demonstrated the main results in Table 1, and here we reported more detailed results of other metrics in Table 11, which underscored the error rates for each stage, including *Invalid*, *Illegal*, and *Low Readability*.

E Case Study

Figure 13 shows an example of a natural language query with its corresponding VQL representation. The output Python code for visualization and the final bar chart are demonstrated in Figure 14 and

	Invalid Rate	Illegal Rate	Pass Rate
NVAGENT	4.66%	23.97%	71.35%
w. CoT for Validator	5.82%	23.39%	70.78%
w. original schema for Validator	4.80%	24.22%	70.97%

Table 10: Additional exploration for Validator (using GPT-3.5-turbo)

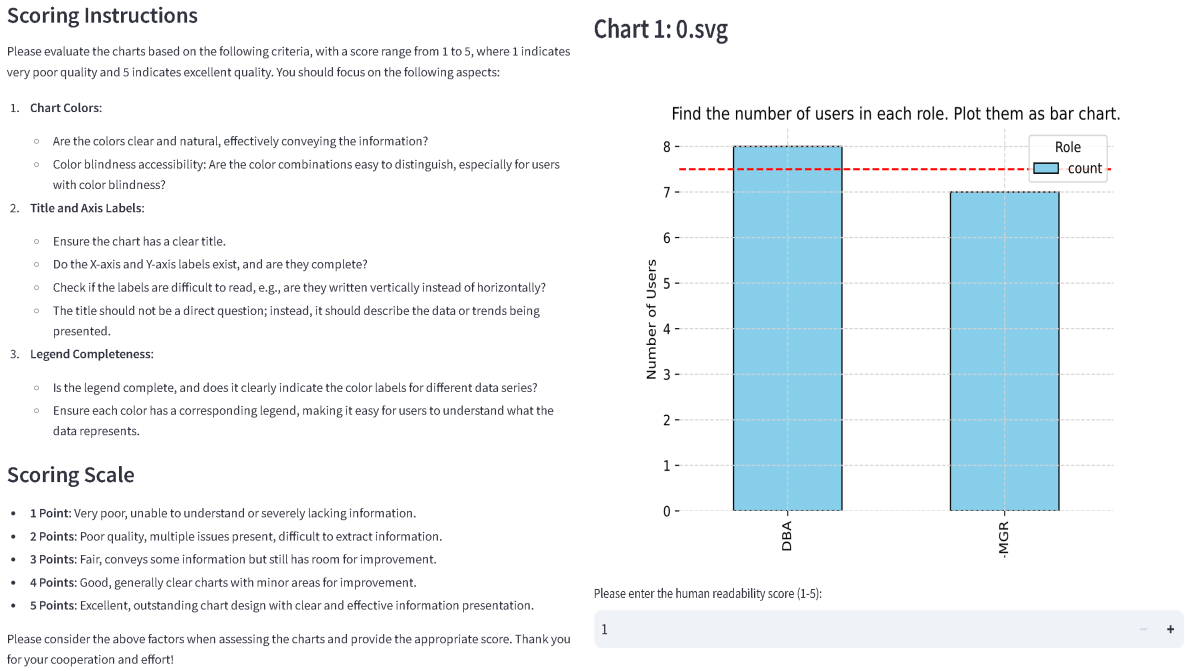


Figure 9: Screenshot of human annotation process in providing readability score.

Figure 15, respectively. Furthermore, we provide a case study of NVAGENT performance on four hardness-level NL2Vis problems in VisEval in Figure 16.

Readability Scoring Instruction

Scoring Instructions: Please evaluate the charts based on the following criteria, with a score range from 1 to 5, where 1 indicates very poor quality and 5 indicates excellent quality. You should focus on the following aspects:

1. Chart Colors:

- Are the colors clear and natural, effectively conveying the information?
- Color blindness accessibility: Are the color combinations easy to distinguish, especially for users with color blindness?

2. Title and Axis Labels:

- Ensure the chart has a clear title.
- Do the X-axis and Y-axis labels exist, and are they complete?
- Check if the labels are difficult to read, e.g., are they written vertically instead of horizontally?
- The title should not be a direct question; instead, it should describe the data or trends being presented.

3. Legend Completeness:

- Is the legend complete, and does it clearly indicate the color labels for different data series?
- Ensure each color has a corresponding legend, making it easy for users to understand what the data represents.

Scoring Scale:

- **1 Point:** Very poor, unable to understand or severely lacking information.
- **2 Points:** Poor quality, multiple issues present, difficult to extract information.
- **3 Points:** Fair, conveys some information but still has room for improvement.
- **4 Points:** Good, generally clear charts with minor areas for improvement.
- **5 Points:** Excellent, outstanding chart design with clear and effective information presentation.

Please consider the above factors when assessing the charts and provide the appropriate score. Thank you for your cooperation and effort!

Figure 10: Human Annotation Readability Scoring Instructions.

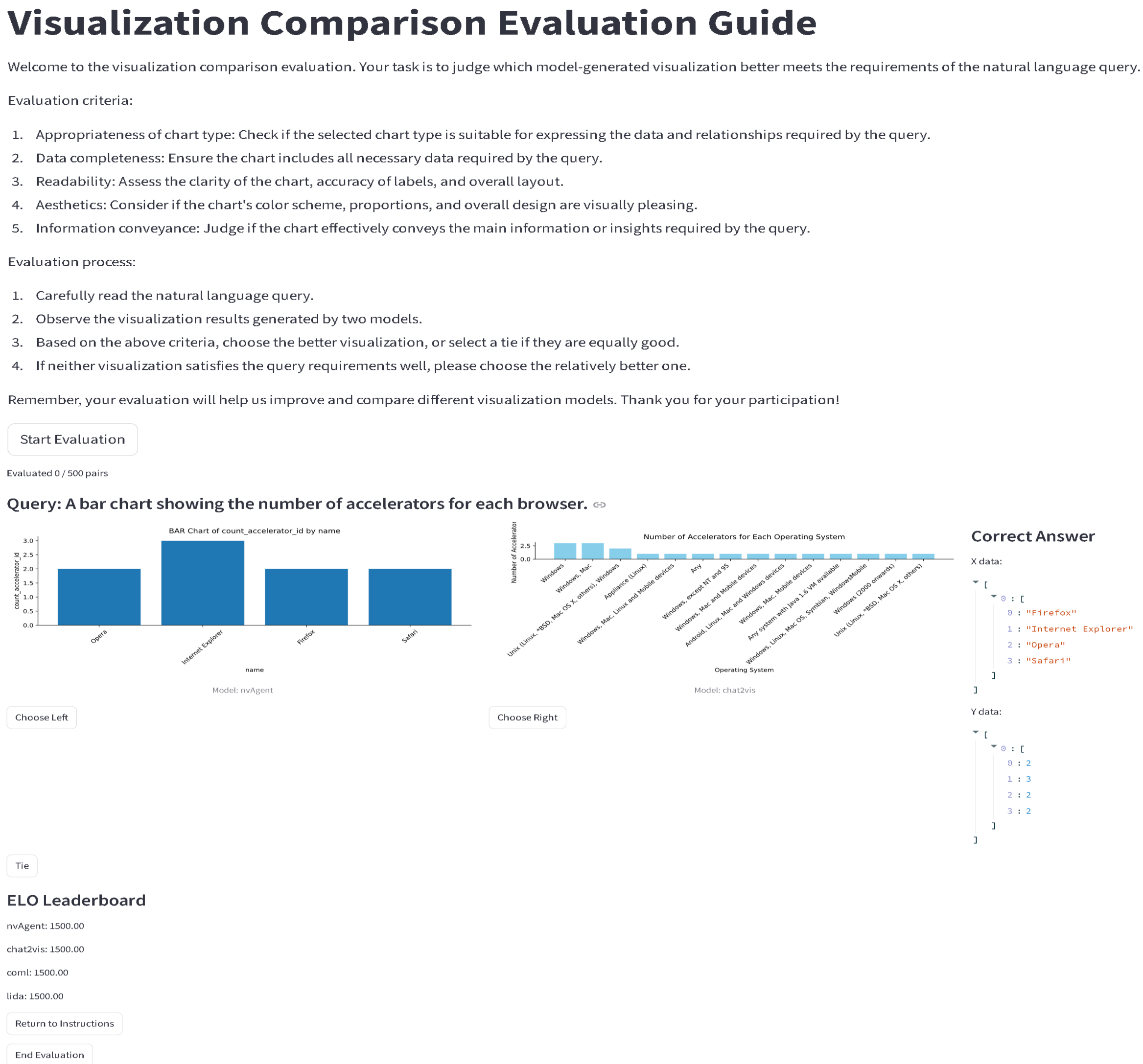


Figure 11: Screenshot of ELO Score Evaluation Framework for Human-as-a-Judge

Visualization Comparison Guidance

Welcome to the visualization comparison evaluation. Your task is to judge which model-generated visualization better meets the requirements of the natural language query.

Evaluation criteria:

1. **Appropriateness of chart type:** Check if the selected chart type is suitable for expressing the data and relationships required by the query.
2. **Data completeness:** Ensure the chart includes all necessary data required by the query.
3. **Readability:** Assess the clarity of the chart, accuracy of labels, and overall layout.
4. **Aesthetics:** Consider if the chart's color scheme, proportions, and overall design are visually pleasing.
5. **Information conveyance:** Judge if the chart effectively conveys the main information or insights required by the query.

Evaluation process:

1. Carefully read the natural language query.
2. Observe the visualization results generated by two models.
3. Based on the above criteria, choose the better visualization or select a tie if they are equally good.
4. If neither visualization satisfies the query requirements well, please choose the relatively better one.

Remember, your evaluation will help us improve and compare different visualization models. Thank you for your participation!

Figure 12: Visualization Comparison Evaluation Instructions.

An Example of Natural Language Query and Corresponding VQL

Natural Language Query:
How many documents are stored? Bin the store date by weekday in a bar chart.

Corresponding VQL:

```
Visualize BAR
SELECT Date_Stored, COUNT(Document_ID)
FROM All_Documents
GROUP BY Date_Stored
BIN Date_Stored BY WEEKDAY
```

Figure 13: The natural language query case and its corresponding output VQL representation.

Method	Dataset	Invalid		Illegal				Low Readability	
		Execution	Surface.	Decon.	Chart Type	Data	Order	Layout	Scale&Ticks
GPT-4o									
CoML4Vis	All	1.15	0.00	0.26	1.75	14.28	10.36	32.02	32.55
	Single	0.67	0.00	0.43	1.93	13.54	10.16	31.08	32.76
	Multiple	1.87	0.00	0.00	1.48	15.39	10.66	33.43	32.23
LIDA	All	6.61	0.00	1.60	3.24	40.53	4.07	32.68	15.77
	Single	1.13	0.00	2.11	0.89	12.26	6.79	53.93	26.22
	Multiple	14.80	0.00	0.79	8.51	80.53	0.00	1.24	0.21
Chat2Vis	All	16.05	0.00	0.62	3.99	30.14	5.96	2.37	20.88
	Single	0.86	0.00	0.75	2.30	10.78	9.73	3.97	34.63
	Multiple	38.74	0.00	0.43	6.51	59.08	0.32	0.00	0.34
nvAgent	All	0.97	0.00	0.08	1.28	11.07	4.05	5.07	40.03
	Single	0.72	0.00	0.14	1.27	9.88	3.60	3.92	39.36
	Multiple	1.34	0.00	0.00	1.30	12.84	4.73	6.79	41.03
GPT-4o-mini									
CoML4Vis	All	4.23	0.00	0.20	2.31	16.64	11.83	35.23	29.35
	Single	0.36	0.00	0.26	2.32	13.80	11.67	35.92	32.22
	Multiple	10.01	0.00	0.10	2.31	20.87	12.07	34.19	25.05
LIDA	All	12.50	0.00	0.40	4.92	40.02	5.80	27.87	17.05
	Single	9.09	0.00	0.44	2.53	12.91	9.68	45.69	28.32
	Multiple	17.61	0.00	0.33	8.51	80.53	0.00	1.24	0.21
Chat2Vis	All	15.45	0.17	0.17	4.21	31.90	8.20	2.14	18.97
	Single	2.14	0.29	0.41	2.53	11.99	9.68	45.69	28.32
	Multiple	35.78	0.00	0.00	6.70	61.66	0.00	0.92	0.32
nvAgent	All	5.14	0.00	0.00	2.40	16.33	10.61	41.06	27.00
	Single	1.97	0.00	0.14	2.97	15.21	7.49	39.30	32.39
	Multiple	8.15	0.00	0.00	2.31	20.87	12.07	34.19	25.05
GPT-3.5-turbo									
CoML4Vis	All	9.28	0.00	0.62	1.91	15.83	12.86	25.09	27.73
	Single	6.17	0.00	0.89	2.50	14.71	13.20	26.10	29.93
	Multiple	13.92	0.00	0.21	1.04	17.51	12.36	23.57	24.43
LIDA	All	53.43	0.00	1.27	3.56	22.33	0.53	14.90	6.62
	Single	47.32	0.00	1.91	2.81	13.03	0.89	24.43	11.05
	Multiple	62.57	0.00	0.32	4.68	36.23	0.00	0.65	0.00
Chat2Vis	All	18.68	0.00	0.28	3.66	32.47	7.20	25.45	20.15
	Single	3.90	0.00	0.47	2.78	15.62	12.01	41.74	33.38
	Multiple	40.77	0.00	0.00	4.97	57.66	0.00	1.12	0.37
nvAgent	All	4.66	0.00	0.08	3.06	18.24	5.64	5.25	35.34
	Single	2.98	0.00	0.14	2.84	15.08	5.69	3.62	37.57
	Multiple	7.18	0.00	0.00	3.38	22.95	5.56	7.69	32.02

Table 11: Detailed Error Rates (%) for Different Methods and Models

```python
import matplotlib.pyplot as plt
import pandas as pd
import os
import duckdb

# Set data folder path
data_folder = 'E:/visEval_dataset/databases/cre_Doc_Tracking_DB'

# Connect to database
con = duckdb.connect(database=':memory:')

# Read all CSV files and create view
csv_files = [f for f in os.listdir(data_folder) if f.endswith('.csv')]
for file in csv_files:
    table_name = os.path.splitext(file)[0]
    con.execute(f"CREATE VIEW {table_name} AS SELECT * FROM read_csv_auto('{os.path.join(data_folder, file)}')")

# Execute SQL query
sql = f'''
    SELECT Date_Stored, COUNT(Document_ID) AS count_Document_ID
    FROM All_Documents
    GROUP BY Date_Stored
'''
df = con.execute(sql).fetchdf()
con.close()

# Rename columns
df.columns = ['Date_Stored','count_Document_ID']

# Apply binning operation
flag = True
df['Date_Stored'] = pd.to_datetime(df['Date_Stored'])
df['Date_Stored'] = df['Date_Stored'].dt.day_name()

# Group by and calculate count
if flag:
    df = df.groupby('Date_Stored').sum().reset_index()

# Ensure all seven days of the week are included
weekday_order = ['Monday', 'Tuesday', 'Wednesday', 'Thursday',
                 'Friday', 'Saturday', 'Sunday']
df = df.set_index('Date_Stored').reindex(weekday_order, fill_value=0).reset_index()
df['Date_Stored'] = pd.Categorical(df['Date_Stored'],
                                   categories=weekday_order, ordered=True)
df = df.sort_values('Date_Stored')

# Create visualization
fig, ax = plt.subplots(1, 1, figsize=(10, 4))
ax.spines['top'].set_visible(False)
ax.spines['right'].set_visible(False)
ax.bar(df['Date_Stored'], df['count_Document_ID'])
ax.set_xlabel('Date_Stored')
ax.set_ylabel('count_Document_ID')
ax.set_title(f'BAR Chart of count_Document_ID by Date_Stored')
plt.xticks(rotation=45)
plt.tight_layout()
plt.show()
```

Figure 14: Python Code for Visualization

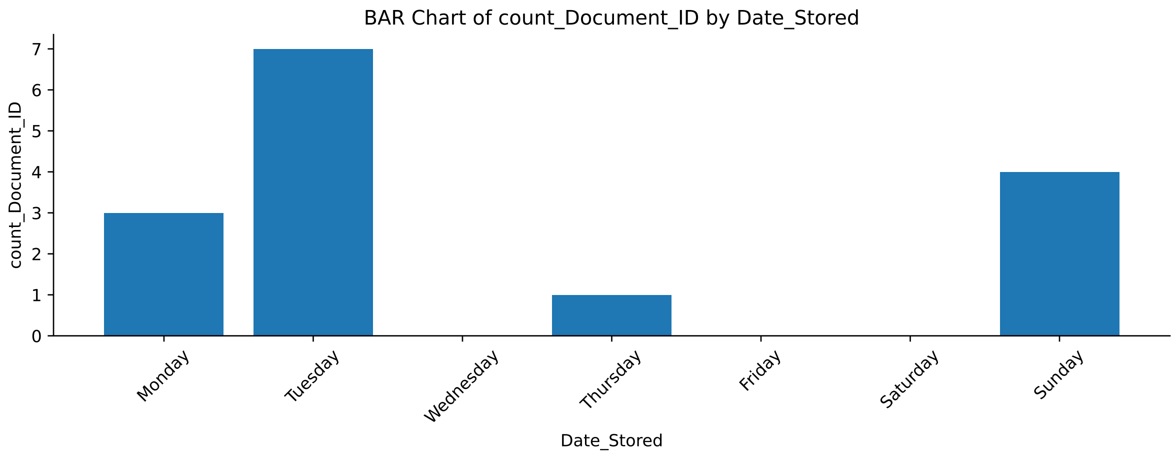


Figure 15: Final Bar Chart

Examples of NVAGENT performance on different hardness levels

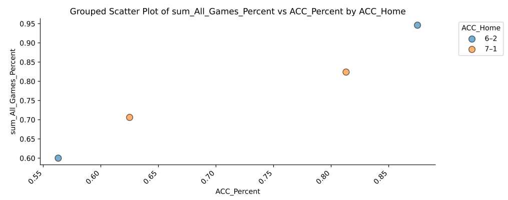
Hardness Level: Easy

Dataset: *Single*

Input Tables: basketball_match

Input Query: Show the relation between acc percent and all_games_percent for each ACC_Home using a grouped scatter chart.

Response:



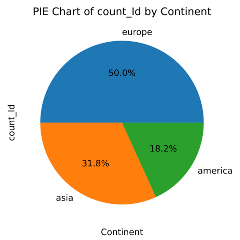
Hardness Level: Medium

Dataset: *Multiple*

Input Tables: car_makers, car_names, cars_data, continents, countries, model_list

Input Query: Display a pie chart for what is the name of each continent and how many car makers are there in each one?

Response:



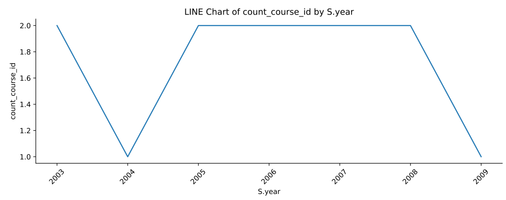
Hardness Level: Hard

Dataset: *Multiple*

Input Tables: advisor, classroom, course, department, instructor, prereq, section, student, takes, teaches, time_slot

Input Query: Find the number of courses offered by Psychology department in each year with a line chart.

Response:



Hardness Level: Extra Hard

Dataset: *Multiple*

Input Tables: Accounts, Documents, Documents_with_Expenses, Projects, Ref_Budget_Codes, Ref_Document_Types, Statements

Input Query: How many documents are created in each day? Bin the document date by weekday and group by document type description with a stacked bar chart, I want to sort Y in desc order.

Response:

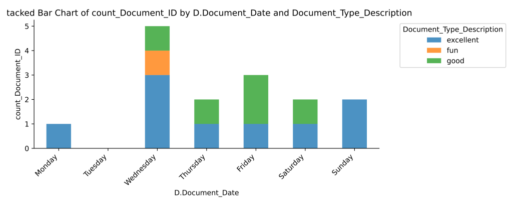


Figure 16: Examples of NVAGENT performing the different hardness levels, including Easy, Medium, Hard, and Extra Hard.

F Prompts Details

We provide detailed prompt design of our NVAGENT.

Prompt template for Processor Agent

You are an experienced and professional database administrator. Given a database schema and a user query, your task is to analyze the query, filter the relevant schema, generate an optimized representation, and classify the query difficulty.

Now you can think step by step, following these instructions below.
[Instructions]
1. Schema Filtering:
   - Identify the tables and columns that are relevant to the user query.
   - Only exclude columns that are completely irrelevant.
   - The output should be {{tables: [columns]}}.
   - Keep the columns needed to be primary keys and foreign keys in the filtered schema.
   - Keep the columns that seem to be similar with other columns of another table.

2. New Schema Generation:
   - Generate a new schema of the filtered schema, based on the given database schema and your filtered schema.

3. Augmented Explanation:
   - Provide a concise summary of the filtered schema to give additional knowledge.
   - Include the number of tables, total columns, and any notable relationships or patterns.

4. Classification:
For the database new schema, classify it as SINGLE or MULTIPLE based on the tables number.
   - if tables number >= 2: predict MULTIPLE
   - elif only one table: predict SINGLE

==============================
Here is a typical example:
[Database Schema]
[DB_ID] dorm_1
[Schema]
Table: Student
[
    (stuid, And This is a id type column),
    (lname, Value examples: ['Smith', 'Pang', 'Lee', 'Adams', 'Nelson', 'Wilson'].),
    (fname, Value examples: ['Eric', 'Lisa', 'David', 'Sarah', 'Paul', 'Michael'].),
    (age, Value examples: [18, 20, 17, 19, 21, 22].),
    (sex, Value examples: ['M', 'F'].),
    (major, Value examples: [600, 520, 550, 50, 540, 100].),
    (advisor, And this is a number type column),
    (city code, Value examples: ['PIT', 'BAL', 'NYC', 'WAS', 'HKG', 'PHL'].)
]
Table: Dorm
[
    (dormid, And This is a id type column),

(dorm name, Value examples: [’Anonymous Donor Hall’, ’Bud Jones Hall’, ’Dorm-plex 2000’, ’Fawlty Towers’, ’Grad Student Asylum’, ’Smith Hall’].),
(student capacity, Value examples: [40, 85, 116, 128, 256, 355].), (gender, Value examples: [’X’, ’F’, ’M’].)
]
Table: Dorm_amenity
[
(amenid, And This is a id type column),
(amenity name, Value examples: [’4 Walls’, ’Air Conditioning’, ’Allows Pets’, ’Carpeted Rooms’, ’Ethernet Ports’, ’Heat’].)
]
Table: Has_amenity
[
(dormid, And This is a id type column),
(amenid, And This is a id type column)
]
Table: Lives_in
[
(stuid, And This is a id type column),
(dormid, And This is a id type column),
(room number, And this is a number type column)
]

[Query]
Find the first name of students who are living in the Smith Hall, and count them by a pie chart

Now we can think step by step
[Filtered Schema]
{
"Student": ["stuid", "fname"],
"Dorm": ["dormid", "dorm name"],
"Lives_in": ["stuid", "dormid"]
}

[New Schema]
Table: Student
[
(stuid, And This is a id type column),
(fname, Value examples: [’Eric’, ’Lisa’, ’David’, ’Sarah’, ’Paul’, ’Michael’].),
]
Table: Dorm [
(dormid, And This is a id type column),
(dorm name, Value examples: [’Anonymous Donor Hall’, ’Bud Jones Hall’, ’Dorm-plex 2000’, ’Fawlty Towers’, ’Grad Student Asylum’, ’Smith Hall’].),
]
Table: Lives_in
[
(stuid, And This is a id type column),
(dormid, And This is a id type column),

]
[Augmented Explanation]
The filtered schema consists of 3 tables (Student, Dorm, and Lives_in) with a total of 6 relevant columns. There is a many-to-one relationship between Student and Dorm through the Lives_in junction table. The query involves joining these three tables to find students living in a specific dorm (Smith Hall).

Key points:
1. The Lives_in table acts as a bridge between Student and Dorm, allowing for the association of students with their dorms.
2. The 'dorm name' column in the Dorm table is crucial for filtering the specific dorm (Smith Hall).
3. The 'fname' column from the Student table is required for the final output.

[Classification]
MULTIPLE

==============================
Here is a new question:

[DB_ID] {db_id}
[Database Schema]
{db_schema}

[Query]
{query}

Now give your answer following this format strictly without other explanation:

[Filtered Schema]

[New Schema]

[Augmented Explanation]

[Classification]

Prompt template for multiple classification

Given a [Database schema] with [Augmented Explanation] and a [Question], generate a valid VQL (Visualization Query Language) sentence. VQL is similar to SQL but includes visualization components.

Now you can think step by step, following these instructions below.
[Background]
VQL Structure:
Visualize [TYPE] SELECT [COLUMNS] FROM [TABLES] [JOIN] [WHERE] [GROUP BY]

[ORDER BY] [BIN BY]

You can consider a VQL sentence as "VIS TYPE + SQL + BINNING"
You must consider which part in the sketch is necessary, which is unnecessary, and construct a specific sketch for the natural language query.

Key Components:
1. Visualization Type: bar, pie, line, scatter, stacked bar, grouped line, grouped scatter
2. SQL Components: SELECT, FROM, JOIN, WHERE, GROUP BY, ORDER BY
3. Binning: BIN [COLUMN] BY [INTERVAL], [INTERVAL]: [YEAR, MONTH, DAY, WEEKDAY]

When generating VQL, we should always consider special rules and constraints:
[Special Rules]
a. For simple visualizations:
  - SELECT exactly TWO columns, X-axis and Y-axis(usually aggregate function)
b. For complex visualizations (STACKED BAR, GROUPED LINE, GROUPED SCATTER):
  - SELECT exactly THREE columns in this order!!!:
    1. X-axis
    2. Y-axis (aggregate function)
    3. Grouping column
c. When "COLORED BY" is mentioned in the question:
  - Use complex visualization type(STACKED BAR for bar charts, GROUPED LINE for line charts, GROUPED SCATTER for scatter charts)
  - Make the "COLORED BY" column the third SELECT column
  - Do NOT include "COLORED BY" in the final VQL
d. Aggregate Functions:
  - Use COUNT for counting occurrences
  - Use SUM only for numeric columns
  - When in doubt, prefer COUNT over SUM
e. Time based questions:
  - Always use BIN BY clause at the end of VQL sentence
  - When you meet the questions including "year", "month", "day", "weekday"
  - Avoid using window function, just use BIN BY to deal with time base queries
[Constraints]
- In SELECT <column>, make sure there are at least two selected!!!
- In FROM <table> or JOIN <table>, do not include unnecessary table
- Use only table names and column names from the given database schema
- Enclose string literals in single quotes
- If [Value examples] of <column> has 'None' or None, use JOIN <table> or WHERE <column> is NOT NULL is better
- Ensure GROUP BY precedes ORDER BY for distinct values
- NEVER use window functions in SQL

Now we could think step by step:
1. First choose visualize type and binning, then construct a specific sketch for the natural language query
2. Second generate SQL components following the sketch.
3. Third add Visualize type and BINNING into the SQL components to generate final VQL

==============================
Here is a typical example:
[Database Schema]
Table: Orders, (orders)
[
(order_id, order id, And this is a id type column),
(customer_id, customer id, And this is a id type column),
(order_date, order date, Value examples: ['2023-01-15', '2023-02-20', '2023-03-10'].),
(total_amount, total amount, Value examples: [100.00, 200.00, 300.00, 400.00, 500.00].)
]
Table: Customers, (customers)
[
(customer_id, customer id, And this is a id type column),
(customer_name, customer name, Value examples: ['John', 'Emma', 'Michael', 'Sophia', 'William'].),
(customer_type, customer type, Value examples: ['Regular', 'VIP', 'New'].)
]
[Augmented Explanation]
The filtered schema consists of 2 tables (Orders and Customers) with a total of 7 relevant columns. There is a one-to-many relationship between Customers and Orders through the customer_id foreign key.

Key points:
1. The Orders table contains information about individual orders, including the order date and total amount.
2. The Customers table contains customer information, including their name and type (Regular, VIP, or New).
3. The customer_id column links the two tables, allowing us to associate orders with specific customers.
4. The order_date column in the Orders table will be used for monthly grouping and binning.
5. The total_amount column in the Orders table needs to be summed for each group.
6. The customer_type column in the Customers table will be used for further grouping and as the third dimension in the stacked bar chart.

The query involves joining these two tables to analyze order amounts by customer type and month, which requires aggregation and time-based binning.

[Question]
Show the total order amount for each customer type by month in a stacked bar chart.

Decompose the task into sub tasks, considering [Background] [Special Rules] [Constraints], and generate the VQL after thinking step by step:

Sub task 1: First choose visualize type and binning, then construct a specific sketch for the natural language query
Visualize type: STACKED BAR, BINNING: True
VQL Sketch:
Visualize STACKED BAR SELECT _ , _ , _ FROM _ JOIN _ ON _ GROUP BY _ BIN _ BY MONTH

1056

Sub task 2: Second generate SQL components following the sketch.
Let’s think step by step:
1. We need to select 3 columns for STACKED BAR chart, order_date as X-axis, SUM(total_amout) as Y-axis, customer_type as group column.
2. We need to join the Orders and Customers tables.
3. We need to group by customer type.
4. We do not need to use any window function for MONTH.

sql
“‘sql
SELECT O.order_date, SUM(O.total_amount), C.customer_type
FROM Orders AS O
JOIN Customers AS C ON O.customer_id = C.customer_id
GROUP BY C.customer_type
“‘

Sub task 3: Third add Visualize type and BINNING into the SQL components to generate final VQL
Final VQL:
Visualize STACKED BAR SELECT O.order_date, SUM(O.total_amount), C.customer_type FROM Orders O JOIN Customers C ON O.customer_id = C.customer_id GROUP BY C.customer_type BIN O.order_date BY MONTH

==============================
Here is a new question:

[Database Schema]
{desc_str}

[Augmented Explanation]
{augmented_explanation}

[Query]
{query}

Now, please generate a VQL sentence for the database schema and question after thinking step by step.

Prompt template for single classification

Given a [Database schema] with [Augmented Explanation] and a [Question], generate a valid VQL (Visualization Query Language) sentence. VQL is similar to SQL but includes visualization components.

Now you can think step by step, following these instructions below.
[Background]
VQL Structure:
Visualize [TYPE] SELECT [COLUMNS] FROM [TABLES] [JOIN] [WHERE] [GROUP BY]

[ORDER BY] [BIN BY]

You can consider a VQL sentence as "VIS TYPE + SQL + BINNING"
You must consider which part in the sketch is necessary, which is unnecessary, and construct a specific sketch for the natural language query.

Key Components:
1. Visualization Type: bar, pie, line, scatter, stacked bar, grouped line, grouped scatter
2. SQL Components: SELECT, FROM, JOIN, WHERE, GROUP BY, ORDER BY
3. Binning: BIN [COLUMN] BY [INTERVAL], [INTERVAL]: [YEAR, MONTH, DAY, WEEKDAY]

When generating VQL, we should always consider special rules and constraints:
[Special Rules]
a. For simple visualizations:
  - SELECT exactly TWO columns, X-axis and Y-axis(usually aggregate function)
b. For complex visualizations (STACKED BAR, GROUPED LINE, GROUPED SCATTER):
  - SELECT exactly THREE columns in this order!!!:
    1. X-axis
    2. Y-axis (aggregate function)
    3. Grouping column
c. When "COLORED BY" is mentioned in the question:
  - Use complex visualization type(STACKED BAR for bar charts, GROUPED LINE for line charts, GROUPED SCATTER for scatter charts)
  - Make the "COLORED BY" column the third SELECT column
  - Do NOT include "COLORED BY" in the final VQL
d. Aggregate Functions:
  - Use COUNT for counting occurrences
  - Use SUM only for numeric columns
  - When in doubt, prefer COUNT over SUM
e. Time based questions:
  - Always use BIN BY clause at the end of VQL sentence
  - When you meet the questions including "year", "month", "day", "weekday"
  - Avoid using window function, just use BIN BY to deal with time base queries
[Constraints]
- In SELECT <column>, make sure there are at least two selected!!!
- In FROM <table> or JOIN <table>, do not include unnecessary table
- Use only table names and column names from the given database schema
- Enclose string literals in single quotes
- If [Value examples] of <column> has 'None' or None, use JOIN <table> or WHERE <column> is NOT NULL is better
- Ensure GROUP BY precedes ORDER BY for distinct values
- NEVER use window functions in SQL

Now we could think step by step:
1. First choose visualize type and binning, then construct a specific sketch for the natural language query
2. Second generate SQL components following the sketch.
3. Third add Visualize type and BINNING into the SQL components to generate final VQL

==============================
Here is a typical example:
[Database Schema]
Table: course, (course)
[
(course_id, course id, Value examples: [101, 696, 656, 659]. And this is an id type column),
(title, title, Value examples: ['Geology', 'Differential Geometry', 'Compiler Design', 'International Trade', 'Composition and Literature', 'Environmental Law'].),
(dept_name, dept name, Value examples: ['Cybernetics', 'Finance', 'Psychology', 'Accounting', 'Mech. Eng.', 'Physics'].),
(credits, credits, Value examples: [3, 4].)
]
Table: section, (section)
[
(course_id, course id, Value examples: [362, 105, 960, 468]. And this is an id type column),
(sec_id, sec id, Value examples: [1, 2, 3]. And this is an id type column),
(semester, semester, Value examples: ['Fall', 'Spring'].),
(year, year, Value examples: [2002, 2006, 2003, 2007, 2010, 2008].),
(building, building, Value examples: ['Saucon', 'Taylor', 'Lamberton', 'Power', 'Fairchild', 'Main'].),
(room_number, room number, Value examples: [180, 183, 134, 143].),
(time_slot_id, time slot id, Value examples: ['D', 'J', 'M', 'C', 'E', 'F']. And this is an id type column)
]
[Augmented Explanation]
The filtered schema consists of 2 tables (course and section) with a total of 11 relevant columns. There is a one-to-many relationship between course and section through the course_id foreign key.

Key points:
1. The course table contains information about individual courses, including the course title, department, and credits.
2. The section table contains information about specific sections of courses, including the semester, year, building, room number, and time slot.
3. The course_id column links the two tables, allowing us to associate sections with specific courses.
4. The dept_name column in the course table will be used to filter for Psychology department courses.
5. The year column in the section table will be used for yearly grouping and binning.
6. We need to count the number of courses offered each year, which requires aggregation and time-based binning.

The query involves joining these two tables to analyze the number of courses offered by the Psychology department each year, which requires aggregation and time-based binning.

[Question]
Find the number of courses offered by Psychology department in each year with a line chart.

Decompose the task into sub tasks, considering [Background] [Special Rules] [Constraints], and generate the VQL after thinking step by step:

Sub task 1: First choose visualize type and binning, then construct a specific sketch for the natural language query
Visualize type: LINE, BINNING: True
VQL Sketch:
Visualize LINE SELECT _ , _ FROM _ JOIN _ ON _ WHERE _ BIN _ BY YEAR

Sub task 2: Second generate SQL components following the sketch.
Let's think step by step:
1. We need to select 2 columns for LINE chart, year as X-axis, COUNT(year) as Y-axis.
2. We need to join the course and section tables to get the number of courses offered by the Psychology department in each year.
3. We need to filter the courses by the Psychology department.
4. We do not need to use any window function for YEAR.

sql
‘‘‘sql
SELECT S.year, COUNT(S.year)
FROM course AS C
JOIN section AS S ON C.course_id = S.course_id
WHERE C.dept_name = ’Psychology’
‘‘‘

Sub task 3: Third add Visualize type and BINNING into the SQL components to generate final VQL
Final VQL:
Visualize LINE SELECT S.year, COUNT(S.year) FROM course C JOIN section S ON C.course_id = S.course_id WHERE C.dept_name = ’Psychology’ BIN S.year BY YEAR

==============================
Here is a new question:

[Database Schema]
{desc_str}

[Augmented Explanation]
{augmented_explanation}

[Query]
{query}

Now, please generate a VQL sentence for the database schema and question after thinking step by step.

Prompt template for Validator Agent

As an AI assistant specializing in data visualization and VQL (Visualization Query Language), your task is to refine a VQL query that has resulted in an error. Please approach this task systematically, thinking step by step.

[Background]
VQL Structure:
Visualize [TYPE] SELECT [COLUMNS] FROM [TABLES] [JOIN] [WHERE] [GROUP BY] [ORDER BY] [BIN BY]

You can consider a VQL sentence as "VIS TYPE + SQL + BINNING"

Key Components:
1. Visualization Type: bar, pie, line, scatter, stacked bar, grouped line, grouped scatter
2. SQL Components: SELECT, FROM, JOIN, WHERE, GROUP BY, ORDER BY
3. Binning: BIN [COLUMN] BY [INTERVAL], [INTERVAL]: [YEAR, MONTH, DAY, WEEKDAY]

When refining VQL, we should always consider special rules and constraints:
[Special Rules]
a. For simple visualizations:
  - SELECT exactly TWO columns, X-axis and Y-axis(usually aggregate function)
b. For complex visualizations (STACKED BAR, GROUPED LINE, GROUPED SCATTER):
  - SELECT exactly THREE columns in this order!!!:
    1. X-axis
    2. Y-axis (aggregate function)
    3. Grouping column
c. When "COLORED BY" is mentioned in the question:
  - Use complex visualization type(STACKED BAR for bar charts, GROUPED LINE for line charts, GROUPED SCATTER for scatter charts)
  - Make the "COLORED BY" column the third SELECT column
  - Do NOT include "COLORED BY" in the final VQL
d. Aggregate Functions:
  - Use COUNT for counting occurrences
  - Use SUM only for numeric columns
  - When in doubt, prefer COUNT over SUM
e. Time based questions:
  - Always use BIN BY clause at the end of VQL sentence
  - When you meet the questions including "year", "month", "day", "weekday"
  - Avoid using time function, just use BIN BY to deal with time base queries

[Constraints]
- In FROM <table> or JOIN <table>, do not include unnecessary table
- Use only table names and column names from the given database schema
- Enclose string literals in single quotes
- If [Value examples] of <column> has 'None' or None, use JOIN <table> or WHERE <column> is NOT NULL is better
- ENSURE GROUP BY clause cannot contain aggregates
- NEVER use date functions in SQL

[Query]
{query}

[Database info]

{db_info}

[Current VQL]
{vql}

[Error]
{error}

Now, please analyze and refine the VQL, please provide:

[Explanation]
[Provide a detailed explanation of your analysis process, the issues identified, and the changes made. Reference specific steps where relevant.]

[Corrected VQL]
[Present your corrected VQL here. Ensure it's on a single line without any line breaks.]

Remember:
- The SQL components must be parseable by DuckDB.
- Do not change rows when you generate the VQL.
- Always verify your answer carefully before submitting.