LLM-Powered Dashboard using Multi-Agent Framework

Luqman Hakim bin Noorazmi1, a) and Zarina binti Che Embi1, 2, b)

1Faculty of Computing and Informatics, Multimedia University, Persiaran Multimedia, 63100 Cyberjaya, Selangor, Malaysia

2 Center for Image and Vision Computing, CoE for Artificial Intelligence, Multimedia University, Persiaran Multimedia, 63100 Cyberjaya, Selangor, Malaysia

*b) Corresponding author: zarina.embi@mmu.edu.my*

*a) luq3973@gmail.com*

**Abstract.** Data visualization helps people understand large and complex datasets. It turns raw information into clear and useful visuals. Many tools and large language models, or LLMs, now support this task. However, current systems still have limits. They do not explore full chart libraries like Plotly. They cannot select from all chart functions or call them directly. This makes it hard to get the best visual for a given dataset. There is a need for systems that can handle more of the visualization process on their own. This project introduces a multi-agent system that automates the full process. It includes SQL query generation, data transformation, chart selection, and analysis writing. Each task is handled by a separate agent. These agents work together inside a graph-based framework using LangChain and LangGraph. The core language model is llama-3.3-70b, an open-source model from Meta. The agents follow clear instructions and pass their output to the next agent in the chain. The system starts when a user enters a prompt. The first agent writes an SQL query based on the prompt and the database schema. If the query runs correctly, the next agent transforms the result into a Python dictionary. Another agent then selects the best chart using Plotly functions. A final agent writes a summary of the data in plain text. Each tool is wrapped within a @tool decorator so agents can use it like a function call. The system was tested on few prompts. Most tasks were completed in less than one second per agent. The full process took about 4.24 seconds on average. The slowest steps were writing and improving SQL queries. The fastest was executing queries. Some prompts failed due to incorrect queries or bad formatting. These failures show that prompt quality and error checking are important. Still, most workflows ran smoothly. Most agents’ output was valid and could be converted into Python objects without extra steps. This project shows that multi-agent systems can support fast and flexible data visualization. The agents work together to turn a simple prompt into a complete visual and summary. This approach could be expanded in the future. More chart types, smarter prompts, and better validation could make the system more powerful. Support for larger databases and more tools would also help. This work lays the foundation for future systems that combine LLMs and code tools to support full automation in data analysis.

# INTRODUCTION

Data visualization tools are important in data analysis. They turn large and complex datasets into clear and readable visuals. Today, these tools help support data-driven decisions. Many options are available, each with its own strengths and weaknesses [1]. A common mistake in scientific charts is using the wrong chart type. This can confuse or mislead people [2]. Large language models (LLMs) are powerful but still face problems. They often struggle in tasks that need high accuracy or real-time results [3]. This issue can be seen in autonomous agents. These agents complete tasks by observing the environment, planning, and acting. There are many types of agents, from general use [4] to agent-based operating systems [5]. In past research, LLM chart systems either suggest a chart type or write code using one fixed library. These systems have a limited range of chart options [6]. This paper proposes a new method. It utilizes LLM agents to automate data visualization workflow. From writing structured query language (SQL) query, validating the query, generating DataFrame, and choosing the best visual, the agents pick the best chart based on the data and prompt. The chart options are given to the agent as Plotly functions. The agent receives a prompt and a dataset. It returns the complete function with all needed values. The goal is to make data visualization easier. The agent chooses the most suitable chart based on the prompt and the data.

Data visualization helps users understand large datasets through visual formats. It allows analysts to find patterns and make better decisions. Tools like Tableau, Power BI, and Qlik Sense support this by creating charts and dashboards. These tools have benefits but also face challenges. Some need technical skills, while others lack features like automation or easy updates. For example, Tableau does not support auto-scheduling. Power BI is hard to learn. D3.js and Matplotlib are complex for beginners. To make tools easier to use, some systems add features like natural language processing (NLP) and AI. Oracle Analytics Cloud allows users to ask questions in plain language. It also supports scenario simulations and multidimensional analysis. IBM Cognos includes an AI assistant. It suggests charts and shows trends in business data. These features help non-technical users explore data [1]. But they only improve certain parts of the visualization process. Language models have improved over time. Early models used statistics. Later models used neural networks. Pre-trained models like BERT and GPT-2 were a major step forward. Today, large language models (LLMs) like GPT-3 and GPT-4 can understand and generate human-like text [7]. They help with many tasks. However, they still have problems. LLMs are trained on past data. They may give wrong or outdated answers. This is a risk in fields that need high accuracy or real-time results [3]. To solve this, researchers use LLMs with agents [4, 5]. An agent is a software system that can plan and act on its own. In multi-agent systems, each agent has a special role. Agents work together to solve problems that one model cannot handle alone. These systems are useful for complex tasks like decision-making or analysis [8]. LLMs have also been used to suggest chart types. One example is LLM4Vis. It uses ChatGPT to recommend simple charts like line or bar graphs [9]. But these systems are limited. They do not explore full libraries like Plotly. They do not choose from all available chart functions or call them directly [10]. This shows a gap in current research. There is no system where LLM agents can browse a full visualization API like Plotly and select the best chart on their own. A new approach is needed. A system that uses LLM-powered agents to generate queries, transform data, select visualizations, and write insights could improve automation. This would allow users to get complete and accurate results from a single prompt.

# METHODOLOGY

To add an LLM to the data visualization tool, this system uses LangChain, an open-source framework for integrating LLMs into software workflows. LangChain supports tasks like chat, summarization, and agent-based automation [11]. It also includes LangGraph, a module that lets agents interact using a graph structure [12]. The model used is llama-3.3-70b, an open-source LLM from Meta. This model handles complex tasks, solves problems, and performs data analysis across domains. To get accurate responses, the prompt must be clear and follow a specific format [13]. LangChain enables agents to perform tasks inside a graph. Each agent has a role. They pass outputs to each other to automate data visualization.

The tools used are Plotly graph functions [14]. These are wrapped with the @tool decorator, which lets agents treat them as callable tools inside LangChain. Each function includes annotations for data type and parameter description [15]. The supported graphs are: Scatter Plot, Bar Chart, Choropleth, Pie Chart, and Table. An agent will select the best chart for the data. It outputs the correct Plotly function call with the required parameters which is made possible because each tool has clear type annotations and descriptions. Agents must produce valid outputs in the correct format. Each agent receives instructions about its task, required input, expected output, and any additional constraints. Router agents control the flow. It checks if the output from an agent is valid. If not, it sends the output back for correction. Each agent has a clear purpose in the visualization workflow. Table 1 describes each agent’s purpose, input, and expected output.

Agent outputs are used by other agents and will be converted directly into a Python object, hence the outputs must follow a strict structure. Few-shot prompting is used to guide output formatting. It gives examples to the agent so it learns the correct format [16]. Table 2 displays the prompt tag to be included within each agents’ prompt instruction for language model llama-3.3-70b.

These outputs must be easily converted into Python objects. Some agents return strings, which are easy to convert. Others must return a string that represents a valid Python dictionary. If not, the system raises an error. The router agents check the output of its previous agent. If it is a valid dictionary, the flow continues. If not, it returns feedback to improve the dictionary. Table 3 displays the example of the expected structured output by the agents.

The full agent workflow starts when the user submits a prompt. The writeQuery agent creates an SQL query from the prompt and schema. The executeQuery agent then runs the query. It returns ‘valid’ if the query works, ‘invalid’ if it fails, or ‘end’ if the prompt and schema do not match. If the query is invalid, the improveQuery agent fixes it. This repeats for up to three attempts. When a query works, generateDF agent converts the result into a dictionary. If it fails, the validator sends it back for correction. Once validated, chooseVisualization agent selects the best Plotly chart and outputs the function with all required parameters. The final agent generates the analysis. It describes the data preview, trends, and key findings. Each output is converted using Python’s built-in functions inside a try block. If conversion fails, the system stops and returns an error. Figure 1 depicts the full agent workflow.

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| **TABLE 1.** Example agent output and data type | | | | | | |
| **Agent** | **Purpose** | | **Input** | **Output** | | |
| writeQuery | Write SQL query | | Database Schema | String of SQL query | | |
| User Prompt |
| executeQuery | Validates the SQL query by checking its result in MySQL. | | Database Schema | String of 'valid' or 'invalid' or 'end' | | |
| User Prompt |
| SQL Query |
| Queried Data |
| improveQuery | Improve existing SQL query | | Database Schema | String of improvised SQL query | | |
| SQL Query |
| Queried Data |
| generateDF | Generate Dictionary of Data | | User Prompt | String, formatted as dictionary, of data from database | | |
| SQL Query |
| Queried Data |
| chooseVisualization | Choose the Best Tool to Represent the Data | | User Prompt | String of the graph or chart function fully equipped with the parameter | | |
| Queried Data |
| Tools |
| generateAnalysis | Generate Analysis based on the Data | | SQL Query | String of the data analysis | | |
| Queried Data |
| dfValidator | Agent Router, responsible of verifying the dictionary output from agent *generateDF* and decide whether the output require improvement. | | SQL Query | String, formatted as dictionary, containing the name of the next agent, and the suggestion for improvement (if any) | | |
| Queried Data |
| Graph or chart function |
| Data Analysis |
| **TABLE 2.** Prompt tags and purposes | | | | | |
| **Tag** | | **Purpose** | | | **Cited Source** |
| <|begin\_of\_text|> | | Start of the prompt | | | [17] |
| <|start\_header\_id|> | | Enclose specific role for specific message: System, User, Assistant | | |
| <|end\_header\_id|> | |
| <|eot\_id|> | | End of turn | | |

# RESULT AND DISCUSSION

To evaluate the system, the output of each agent was analyzed. The syntax and semantics of each agent’s output are evaluated to ensure its correctness and alignment with the expected format. If it fails to be converted directly into a Python object, the output of the agent is considered invalid. The database contains information about a singer’s songs, albums, popularity by country, and song lyrics. A sample prompt was given to begin the agent workflow to generate a visual of the most popular album from each Southeast Asian country. The writeQuery agent generated a valid SQL query. It used nested subqueries, joins, and filters correctly. The query matched the user’s request and the database schema. It was executable and showed that the agent could handle complex instructions. The output from the executeQuery agent returns a list of country and album name pairs. The result matched the schema and the query logic. The query ran without errors. This confirms that the agent can interact with the database reliably.

The improveQuery agent fixed errors in a flawed SQL query. The original query generated by the writeQuery agent used the GROUP BY clause incorrectly. The agent explained the problem and suggested a fix. It rewrote the query using a subquery to find the top albums per country. The final result was correct and logical. The generateDF agent returns a string formatted as a Python dictionary object. The keys were column names and the values were lists of the corresponding column’s value. The structure matched the query result and was directly converted into a Python dictionary object. This shows that the agent can format data properly. The chooseVisualization agent selected a choropleth map using the output from generateDF agent. The Python dictionary object is converted into a DataFrame, which is then passed as a parameter into the tool chosen by the agent. The agent was able to generate output formatted as an executable Python function based on the query and data. The generateAnalysis agent created a detailed summary. The output was a single string, divided into sections: data preview, trends, and conclusion. It clearly referenced album names and countries from the dataset. The output from the dfValidator agent returns a dictionary with two keys: agentName and improvementMessage. Both values were strings. The structure was simple and easy to read. It helped manage the workflow by directing the next step. Overall, the multi-agent system worked as expected. Each agent completed its task. The outputs were valid, correctly formatted, and followed the user’s intent. From query creation to visualization and analysis, the process was smooth and reliable. Figure 2 shows the chart generated from the user prompt.

|  |  |  |
| --- | --- | --- |
| **TABLE 3.** Example agent output and data type | | |
| **Agent** | **Example Structured Output String** | **Target Data Type** |
| writeQuery | SELECT \* FROM employees; | String |
| executeQuery | valid | String |
| generateDF | '{  'id': [1, 2],  'name': ['Alice Smith', 'Bob Jones'],  'department': ['Engineering', 'Marketing'],  'salary': [80000, 60000]  }' | Dictionary |
| chooseVisualization | Figure({  'data': [{'hovertemplate': 'name=%{x}<br>salary=%{y}<extra></extra>',  'marker': {'color': '#636efa', 'pattern': {'shape': ''}},  'orientation': 'v',  'showlegend': False,  'textposition': 'auto',  'type': 'bar',  'x': [Alice Smith, Bob Jones],  'y': [80000, 60000]}],  'layout': {'barmode': 'relative',  'legend': {'tracegroupgap': 0},  'margin': {'t': 60},  'template': '...',  'title': {'text': 'Employee Salaries'},  'xaxis': {'title': {'text': 'Employee Name'}},  'yaxis': {'title': {'text': 'Salary'}}}  })' | plotly.graph\_objs.\_figure.Figure |
| dfValidator | '{"agentName": "chooseVisualization",  "improvementMessage": ""  }' | Python dictionary |

The system can also handle different types of charts. If a specific chart is mentioned in the prompt, the agent will follow the request. Some prompts caused the system to fail. This happened at different stages of the workflow. For example, prompts about track features like danceability and acousticness sometimes failed. The SQL query could not be executed or did not match the schema. In other cases, the generateDF or dfValidator agents returned wrong outputs. Some outputs were empty or not in the right format. Prompts that asked for data from multiple albums also had issues. These failures show that the system is sensitive to how the prompt is written. Better prompts, more checks, and stronger error handling could improve the results. The system was tested on 10 prompts to measure performance. Due to model limitations in the free tier, the number of tests was small. The full workflow took between 3.93 and 4.45 seconds per prompt. The average was 4.24 seconds. This shows the system is able to complete a full cycle quickly. The writeQuery and improveQuery agents were the slowest, each taking about 0.9 seconds. The executeQuery agent was the fastest, averaging 0.27 seconds. The visualization and chart selection agents took around 0.41 and 0.61 seconds. The time was mostly balanced across agents. The slower ones may be improved by using better prompt designs or reducing generation steps. The system worked well on the small test set. More testing with larger data is needed to measure scalability. A stronger model or paid-tier access could improve future performance.

A diagram of a software

AI-generated content may be incorrect., Picture

**FIGURE 1.** Full agents’ workflow

A map of the world

AI-generated content may be incorrect., Picture**FIGURE 2.** Generated visual from the agents’ workflow

# CONCLUSION

This project presents an automated data visualization tool using a multi-agent framework. The agent was able to produce the expected results in near real-time. While the system meets its goals, there is room for improvement. The SQL generation agent depends on the database schema to create SQL queries. The test used a small and simple database. If tested on a larger and more complex database, the results may not be the same. The system also relies heavily on the agent to create dictionaries. This can be a problem because LLMs may generate incorrect or made-up data. A verification step should be added to verify the dictionary against the actual data after it is created. The system also requires charts and graphs to be coded manually to be made available for the agent to choose. This limits scalability when adding new visual types. The performance of the LLM can be improved with better prompts, retrieval-based methods, or possibly fine-tuning. This project presents a first step toward automating data visualization using a multi-agent system.

# Acknowledgments

The authors would like to thank Multimedia University for the support given to this work.

# REFERENCES

1. A. Lavanya, S. Sindhuja, L. Gaurav, and W. Ali, “A Comprehensive Review of Data Visualization Tools: Features, Strengths, and Weaknesses,” IJCERT **10**(1), 10–20 (2023).
2. Y. Xie, Y. Luo, G. Li, and N. Tang, “HAIChart: Human and AI Paired Visualization System,” (2024).
3. Z. Shen, “LLM With Tools: A Survey,” (2024).
4. J. Ruan, Y. Chen, B. Zhang, Z. Xu, T. Bao, H. Mao, and R. Zhao, NeurIPS 2023 Foundation Models for Decision Making Workshop (2023).
5. K. Mei, X. Zhu, W. Xu, W. Hua, M. Jin, Z. Li, S. Xu, R. Ye, Y. Ge, and Y. Zhang, “AIOS: LLM Agent Operating System,” (2025).
6. F. Skender and V. Manevska, "Data visualization tools-preview and comparison," Journal on Emerging Computing Technologies **2**(1), 30-35 (2022).
7. W.X. Zhao, K. Zhou, J. Li, T. Tang, X. Wang, Y. Hou, Y. Min, B. Zhang, J. Zhang, Z. Dong, Y. Du, C. Yang, Y. Chen, Z. Chen, J. Jiang, R. Ren, Y. Li, X. Tang, Z. Liu, P. Liu, J.-Y. Nie, and J.-R. Wen, “A Survey of Large Language Models,” arXiv.Org, (2023).
8. Y. Talebirad, and A. Nadiri, “Multi-Agent Collaboration: Harnessing the Power of Intelligent LLM Agents,” arXiv.Org, (2023).
9. L. Wang, S. Zhang, Y. Wang, E.-P. Lim, and Y. Wang, “LLM4Vis: Explainable Visualization Recommendation using ChatGPT,” (2023).
10. K. Goswami, P. Mathur, R. Rossi, and F. Dernoncourt, “PlotGen: Multi-Agent LLM-based Scientific Data Visualization via Multimodal Feedback,” (2025).
11. “Introduction | LangChain,” (n.d.).
12. “LangGraph,” LangChain Blog, (2024).
13. “Llama-3.3-70B-Versatile - GroqDocs,” (n.d.).
14. “Plotly Python Graphing Library,” Plotly, (2025).
15. “How-to guides | LangChain,” (n.d.).
16. “Few-Shot Prompting – Nextra,” (n.d.).
17. “Model cards and prompt formats,” Llama 3.1, (2024).