

UNIVERSITAT DE BARCELONA

Treball final de grau

GRAU D'ENGINYERIA INFORMÀTICA

Facultat de Matemàtiques i Informàtica
Universitat de Barcelona

An Interactive LLM-based Conversational Agent for Complex Data Analysis

Author: Rubén Jurado González

Advisors:	**Dra. Anna Puig**
	Dra. Inmaculada Rodríguez
Conducted at:	**Departament de Matemàtiques i Informàtica**

Barcelona, **4 de juliol de 2025**

Abstract

Complex multivariate datasets—characterized by complex parent-child structures and rich attributes such as hierarchies and networks—pose challenges for intuitive exploration and analysis. This work presents an interactive visualization system integrated with a conversational agent (chatbot) to support natural language interaction with such data, especially for domain experts.

Users can upload datasets, issue natural language queries, manipulate interface elements (e.g., buttons, panels), and generate custom visualizations including force-directed graphs, circle-packing layouts, and tabular charts. These features enhance data interpretability and engagement.

The system includes a robust NLP pipeline based on DistilBERT for intent classification, optimized through data balancing and retraining. Visualizations, rendered in real time with Plotly and D3.js in a Dash interface, support interactions such as zooming, panning, node selection, and dynamic color mapping via language commands.

A Retrieval-Augmented Generation (RAG) pipeline enriches chatbot responses using contextual information from uploaded documents. The system also supports misclassification reporting to iteratively refine the NLP model. It handles large-scale hierarchical data efficiently and has been validated on examples like organizational charts and threaded discussions.

Notable features include real-time GUI customization, multi-turn conversational support, and popup visualizations from selected data subsets using intuitive queries (e.g., “Show toxicity distribution”). User testing showed high satisfaction among experts, while novices noted a steeper learning curve during onboarding.

Resumen

Los conjuntos de datos multivariantes complejos—caracterizados por estructuras complejas de padre-hijo y atributos ricos como jerarquías y redes—presentan retos significativos para su análisis intuitivo. Este trabajo presenta un sistema de visualización interactiva integrado con un agente conversacional (chatbot), diseñado para facilitar la exploración mediante lenguaje natural, especialmente por parte de usuarios expertos.

El sistema permite cargar conjuntos de datos, realizar consultas conversacionales, manipular elementos gráficos (botones, paneles) y generar visualizaciones personalizadas como grafos dirigidos por fuerza, empaquetamiento circular y gráficos tabulares. Estas funcionalidades mejoran la comprensión de los datos y la interacción del usuario.

El núcleo técnico se basa en un pipeline de PLN con un modelo DistilBERT para la clasificación de intenciones, optimizado mediante reentrenamiento. Las visualizaciones, generadas con Plotly y D3.js dentro de una interfaz Dash, permiten acciones como zoom, desplazamiento, selección de nodos y mapeo dinámico de colores mediante comandos en lenguaje natural.

Un pipeline de RAG enriquece las respuestas del chatbot con información contextual de documentos cargados. El sistema admite también la notificación de errores de clasificación, lo que contribuye a su mejora continua. Su rendimiento ha sido validado con estructuras jerárquicas como organigramas y hilos de discusión.

Entre sus funcionalidades clave se incluyen la personalización dinámica de la interfaz, soporte conversacional para consultas complejas, y generación de gráficos emergentes mediante comandos intuitivos (e.g., “Mostrar distribución de toxicidad”). Las pruebas con usuarios reflejaron una alta satisfacción entre perfiles expertos, mientras que los principiantes señalaron una curva de aprendizaje más pronunciada.

Resum

Els conjunts de dades multivariants complexos —caracteritzats per estructures complexes de pare-fill i atributs rics com jerarquies i xarxes—presenten reptes importants per a una anàlisi intuïtiva. Aquest treball presenta un sistema de visualització interactiva integrat amb un agent conversacional (xatbot), dissenyat per facilitar-ne l'exploració mitjançant llenguatge natural, especialment per part d'usuaris experts.

El sistema permet carregar dades, fer consultes conversacionals, manipular la interfície (botons, panells) i generar visualitzacions com gràfics de força, empaquetament circular i gràfics tabulars. Aquestes funcionalitats milloren la comprensió i la interacció amb les dades.

El nucli tècnic es basa en un pipeline de PLN amb el model DistilBERT per a la classificació d'intencions, optimitzat amb tècniques d'equilibri i reentrenament. Les visualitzacions es generen en temps real amb Plotly i D3.js dins d'una interfície Dash, amb suport per zoom, desplaçament, selecció de nodes i mapeig de colors dinàmic, tot controlable amb llenguatge natural.

Un pipeline RAG millora les respostes del xatbot amb informació contextual de documents pujats pels usuaris. El sistema també permet notificar errors de classificació, afavorint la seva millora contínua. Ha estat validat amb estructures jeràrquiques com organigrames i fils de discussió.

Entre les funcionalitats clau hi ha la personalització dinàmica de la interfície, el suport a consultes complexes i la generació de gràfics emergents amb ordres intuïtives (per exemple, "Mostra la distribució de toxicitat"). Les proves d'usuari mostren una alta satisfacció entre experts, mentre que els usuaris novells van assenyalar una certa corba d'aprenentatge inicial.

Acknowlegements

I would like to express my deepest gratitude to Dra. Anna Puig and Dra. Inmaculada Rodríguez for their unwavering support and insightful guidance throughout every stage of this project. Without their expertise and encouragement, this work would not have been possible.

I am also profoundly grateful to Cristina Burgos for her creative contributions to the graph designs and overall visual development; her artistic insight greatly enhanced the presentation and usability of the visualizations.

I would like to thank my parents Aurelio Jurado and Maria Antonia Gonzalez for their constant encouragement and support and my brother Ivan Jurado, which sustained me throughout this journey.

Finally, I thank the technical staff of the university for providing the computational resources necessary for development, and all early testers whose feedback helped refine the system's usability.

Contents

Introduction v

1 Problem Analysis 1

1.1 Background in Hierarchical Data Analysis . . . 1

1.2 Challenges . . . 2

1.2.1 NLP Challenges . . . 2

1.2.2 Visualization Challenges . . . 3

1.2.3 System Challenges . . . 3

1.3 Requirements . . . 4

1.3.1 Functional Requirements . . . 4

1.3.2 Non Functional Requirements . . . 5

2 Related Work 7

2.1 Existing V-NLIs . . . 7

2.1.1 FlowNL . . . 8

2.1.2 Evizeon . . . 9

2.1.3 Snowy . . . 10

2.1.4 Natural Language Interfaces for Specific Data Types . . . 11

2.2 NLP Techniques in Conversational Interfaces . . . 11

2.2.1 Intent Classification and Query Processing . . . 11

2.2.2 Contextual Enrichment via RAG . . . 12

2.3 Current Gaps and Motivation . . . 12

3 Interface Design 15

3.1 Introduction . . . 15

3.2 Design Proposal . . . 15
3.2.1 Design Choices for Accessibility and User-Friendliness . . . 16
3.2.2 Simplifying Complex Interactions . . . 17
3.2.3 Interaction Flow . . . 17
3.3 Welcome Page . . . 18
3.4 Dashboard Buttons . . . 20
3.5 Upload Additional Files . . . 20
3.6 Main Visualization . . . 21
3.7 Chatbot Sidebar . . . 22
3.8 PopUp Visualization . . . 24

4 Hierarchical Interactive Visualization Architecture **27**
4.1 System Overview . . . 27
4.2 NLP Parser . . . 28
4.2.1 Stage 1: Initialization . . . 31
4.2.2 Stage 2: Query Processor . . . 33
4.2.3 Stage 3: Cluster Prediction . . . 34
4.2.4 Stage 4: Intent and Label Classification . . . 34
4.3 Visualization Generator . . . 37
4.3.1 Hierarchical Visualizations . . . 39
4.3.2 Tabular Visualizations . . . 41
4.4 Reasoning Module . . . 43
4.5 Data Handling . . . 44
4.5.1 The Core of Data Management . . . 45
4.5.2 Metadata and Building the Structures . . . 45
4.6 Retrieval-Augmented Generation (RAG) Module . . . 47
4.6.1 Step 1: Document Processing and Text Extraction . . . 48
4.6.2 Step 2: Text Chunking . . . 49
4.6.3 Step 3: Embedding Generation . . . 50
4.6.4 Step 4: Vector Storage . . . 51
4.6.5 Step 5: Retrieval Mechanism . . . 51

5 Implementation **53**

5.1 System Overview Implementation . . . 53
5.2 NLP Parser . . . 54
5.2.1 Cluster and Classification Transformer . . . 55
5.2.2 Query Processing Pipeline . . . 56
5.2.3 Cluster Prediction and Classification . . . 58
5.2.4 Intent and Label Resolution via LLM . . . 60
5.3 Controller Implementation . . . 62
5.3.1 Key Components and Technologies . . . 63
5.3.2 Context Management . . . 64
5.3.3 Configuration Updates . . . 64
5.4 Visualization Generator Implementation . . . 65
5.4.1 Hierarchical Visualizations . . . 65
5.4.2 Tabular Visualizations . . . 68
5.5 Reasoning Module . . . 70
5.5.1 Explanation and Reasoning . . . 70
5.5.2 Goal Generation . . . 71
5.6 Data Handling . . . 72
5.6.1 Data Input and Metadata Management . . . 73
5.6.2 General Algorithm for Building a Hierarchical Graph . . . 73
5.7 RAG Module . . . 75
5.7.1 Step 1: Document Processing and Text Extraction . . . 75
5.7.2 Step 2: Text Chunking . . . 75
5.7.3 Step 3: Embedding Generation . . . 77
5.7.4 Step 4: Vector Storage . . . 77
5.7.5 Step 5: Retrieval Mechanism . . . 77
5.7.6 Step 6: Response Generation . . . 78

6 Evaluation **79**
6.1 Methodology . . . 79
6.2 Participants . . . 80
6.3 Procedure and Tasks . . . 80
6.4 Results . . . 81
6.4.1 Task 1 Results . . . 81

6.4.2 Task 2 Results 81
6.4.3 Task 3 Results 82
6.4.4 SUS Results 82
6.5 Discussion 83
6.6 Limitations 84

7 Conclusions and Future Work **85**

A Technical Manual **87**
A.0.1 Repository Access 87

A Intent And Label Recognition Ouput By LLM **89**

B Metadata Full Explanation **91**

C Full Hierarchical Algorithm Explanation **95**

Bibliography **97**

Introduction

Context

In today's world, we collect and store data at an unprecedented rate. Whether it's tracking sales figures in a company, mapping relationships in a social network, or monitoring ecosystems in biology, we often organize information in nested structures—where each element can have multiple properties. For example, an organizational chart might list employees, their roles, departments, and performance metrics. Exploring such hierarchical, multivariate datasets can quickly become overwhelming with conventional tools. Static menus and complex dropdowns make it hard to find the right view, and users without advanced technical skills can struggle to answer simple questions like "Who leads the marketing team?" or "Show me all departments with more than 50 employees."

Interactive visualization techniques—such as zooming, panning, selecting subsets of nodes, and dynamically generating related charts—offer a powerful way to make sense of complex data at a glance. By allowing users to click on a branch of a tree, zoom into a region of a force-directed graph, or generate a pie chart of a selected subset, interactive visuals transform raw numbers into an exploratory experience. However, building an interface that combines these interactions with the ability to ask questions in plain language adds a new layer of usability. This work sits at the intersection of two fields: interactive data visualization and natural language processing. The goal is to let users converse with a system—typing or speaking requests—and instantly see updated visuals, whether hierarchical graphs or tabular charts like bar plots or line graphs.

During my degree, I studied Machine Learning, which provided the foundations for teaching a model to interpret a wide range of user requests. Courses in Graphs and Data Visualization introduced me to layouts like circle-packing and force-directed diagrams. Software Engineering and Web Development classes taught me how to structure a Dash application, integrating back-end logic with front-end interactivity. Algorithms and Data Structures courses ensured that, behind the scenes, we could handle large hierarchies efficiently without long load times. All these experiences converge in this project, which aims to bridge user intent—expressed in natural language—with interactive, dynamic visual exploration.

Motivation

Traditional visualization platforms often fall short when handling nested, multivariate data. Users must navigate through multiple menus to drill down into deeper levels or switch views. Selecting a subset of interest might require writing complex queries or manually adjusting filters. Furthermore, when data changes—new employees join, departments reorganize—the user typically needs to reconfigure settings or reload static charts. For many professionals, this workflow is time-consuming and error-prone. More importantly, it excludes non-technical users who could benefit from instant insights but lack the coding skills or familiarity with the interface.

Our project addresses these shortcomings by embedding a conversational agent—a chatbot—directly into the visualization tool. Instead of hunting through menus, a user can type or speak "Zoom into the sales department," "Highlight teams with social media Managers," or "Generate a pie chart of department sizes." The system understands the request, updates the interactive graph (allowing zooming, panning, and node selection), and, if needed, spawns new tabular charts (bar, pie, or line graphs) that reflect the current focus. This approach reduces friction, speeds up analysis, and opens hierarchical data exploration to a broader audience.

The project's scope is deliberately focused. We concentrate on hierarchical, multivariate datasets—such as organizational charts or taxonomies—where each node can have several attributes (e.g., name, role, metric values). We do not attempt to visualize every possible data type or deliver a general-purpose analytics platform. Instead, we aim to create a highly responsive, conversational interface that makes nested data easy to navigate, inspect, and manipulate. By combining a chatbot's natural language understanding with fully interactive visual components, users can explore complex datasets in an intuitive, flexible way.

Main goal

The main goal of this project is to build a system that melds a conversational chatbot with interactive visualization tools, enabling experienced users to explore hierarchical, multivariate datasets through natural language commands. Users will be able to issue commands such as "Show the top three branches by revenue," "Zoom into the North America region," or "Color nodes by performance score," and immediately see a force-directed or circle-packing graph update accordingly. Beyond these hierarchical views, the system will support full interactivity: selecting individual or multiple nodes, panning and zooming within the graph, and spawning new tabular charts—pie, bar, or line graphs—based on the current selection. By combining natural-language interaction with real-time visuals, we aim to make data exploration more efficient and intuitive, particularly for professionals with domain expertise but without extensive programming skills.

Specific goals

To achieve this main objective, the project is structured around the following subgoals:

- **Integrate a conversational interface with NLP:** Build a chatbot powered by DistilBERT that understands plain-language requests—such as "Generate a pie chart of toxicity level," "Zoom into the most ," or "Highlight nodes with revenue above $1M"—and accurately translates them into visualization commands.
- **Develop interactive visualizations:** Use Plotly and D3.js within a Dash framework to render hierarchical graphs (force-directed, circle-packing) that support zooming, panning, node selection, dynamic color mapping, and spawning auxiliary charts (bar, pie, line) based on the current subset.
- **Provide context-aware responses:** Implement a Retrieval-Augmented Generation (RAG) module so the chatbot can reference uploaded documents or metadata, answering questions like "What was last quarter's revenue for the Marketing division?", while also allowing users to correct misinterpretations ("No, I meant 'region,' not 'department'") to refine intent classification over time.
- **Ensure performance and robustness:** Design data structures and back-end processes to handle large hierarchies (thousands of nodes) without noticeable lag, detect and handle malformed inputs (e.g., incomplete JSON).
- **Validate with targeted testing:** Conduct thorough evaluations with experienced users (the primary target audience) and a limited group of novice testers to ensure the interface remains responsive, user-friendly, and accurate across different data complexities.

Planning

The project is set to take six months, with tasks planned out to keep everything on track. A Gantt chart in Figure 1 shows the timeline and key steps. Here's how it breaks down:

- **Month 1:** Research existing tools and techniques for chatbots and data visuals to learn what works best.
- **Month 2:** Design the system's structure, including how the chatbot, visuals, and interface will connect.
- **Months 3-4:** Build the chatbot, set up the visualization tools, and link them in the Dash interface.
- **Month 5:** Test the system to check if it understands requests, shows data correctly, and runs smoothly.
- **Month 6:** Fix any issues found in testing, finish the documentation, and complete the project.

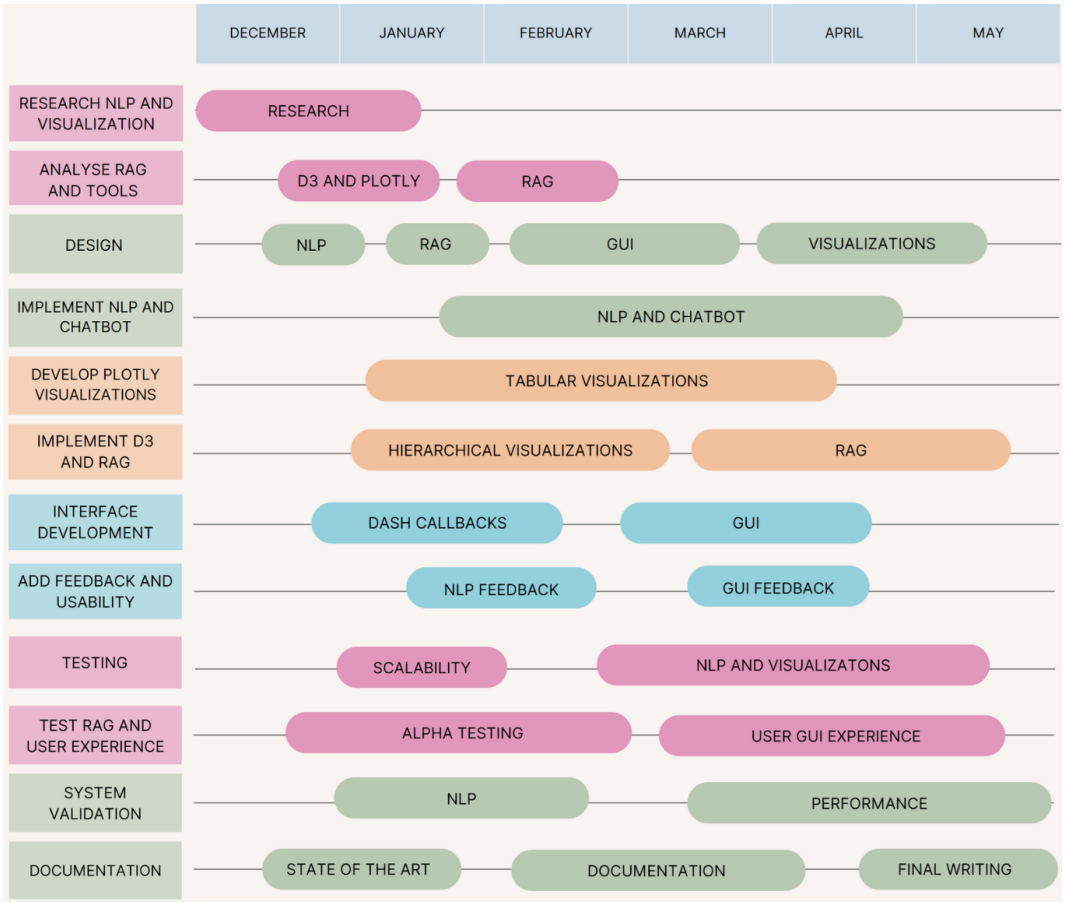


Figure 1: Gantt chart showing the project timeline and main tasks.

Organization of this document

This project is organized into these chapters:

- **Introduction** - Explains the project's background, why it matters, its goals, plan, and layout.
- **Chapter 1: Problem Analysis** - Looks at existing tools and research related to chatbots and visuals.
- **Chapter 2: Related Work** - Describes the research that has been done and shows the comparison between systems.
- **Chapter 3: System Architecture** - Describes how the system is built and how its parts work together.
- **Chapter 4: Implementation** - Detailed steps of the used technologies and the reasoning behind its implementation.

- **Chapter 5: Interface Design** - Details the steps taken to create the interface, the visuals and its reasoning.
- **Chapter 6: Evaluation** - Shows how the system was tested and what users thought of it.
- **Chapter 7: Conclusions and Future Work** - Wraps up the project's results and ideas for what's next.
- **Appendices** - Includes extra details like setup guides or code samples.

Chapter 1

Problem Analysis

In today's world, where data plays a key role, being able to understand and analyze complex datasets is more important than ever. But as these datasets become more detailed and include more types of information, traditional tools often are not enough. Hierarchical multivariate datasets, such as organizational charts, family trees, biological classifications, or social networks, present specific challenges due to their layered structure and the rich information each part contains. These are not just simple lists; they are systems in which the connections between items are just as important as the items themselves. Most common tools, designed for flat, table-like data, struggle with this complexity. They usually offer static graphics and fixed interfaces that limit how users can explore the data. To solve this, we suggest a new approach: adding a chatbot to a graphical user interface (GUI) designed to work with hierarchical data and tabular data. This setup allows users, whether they are analysts, researchers, or teachers, to interact with and adjust the visualizations using natural language, making the process easier and more accessible, even without technical skills. This chapter looks at the unique challenges of these types of data and the technical requirements needed to develop this solution.

1.1 Background in Hierarchical Data Analysis

Hierarchical data is all around us. It appears in things like company organization charts, family trees, biological classifications, and social networks. This kind of data is built on parent-child relationships, creating a layered structure where each point, or node, includes a mix of information such as numbers, categories, or text. For example, in an organizational chart, employees are shown as nodes connected to their managers, along with details like their name, job title, department, salary, or performance. A family tree is similar, showing individuals along with birth dates, jobs, and roles across generations. The challenge is to keep these complex relationships organized while making the structure and its details easy to understand and use.

Standard tools, such as spreadsheets or basic charting software, often struggle with this complexity. Simple tree diagrams might work for small datasets, but they quickly become messy when dealing with larger ones, such as a global company with thousands of employees. Branches can overlap, and labels become difficult to read. This reveals a clear need: interactive visualizations that let users zoom in on specific sections, such as the marketing department, or filter by roles, such as 'managers only'. Having this kind of flexibility is crucial for exploring the data, comparing different parts of the hierarchy, spotting unusual patterns, or adjusting the view as new questions arise. Without it, working with hierarchical data becomes much harder.

1.2 Challenges

To effectively manage hierarchical data, the proposed system must meet several technical requirements, including natural language processing (NLP), visualization, and broader system needs such as performance and user experience. These challenges are closely connected: how well the chatbot understands user input influences how the data is visualized, and both rely on a strong and efficient system foundation. This section breaks the challenges into three main categories: NLP, visualization, and system-level requirements, and explores each in detail.

1.2.1 NLP Challenges

The conversational agent is a key part of the system and is responsible for understanding natural language and turning it into meaningful actions. It needs to recognize that different sentences, such as “draw a tree,” “show the hierarchy,” or “make an org chart”, may all mean the same thing. The NLP component must be able to group these different expressions into a single intent.

More complex commands, such as “explain this node”, require the system to identify which node the user is referring to (based on context or previous interactions) and then display its details, such as attributes or its place in the hierarchy.

Ambiguous inputs add another layer of difficulty. A vague command like “tell me about it” could refer to the entire dataset, a specific node, or a recent action. In these cases, the system should either interpret the most likely meaning or ask the user to clarify. Advanced NLP models, such as those based on contextual embeddings (e.g., DistilBERT), help manage this by considering both the conversation history and the data context. Multipart queries, such as “build a chart and highlight key nodes”, are even more challenging, as the system must break them down into steps and respond accordingly.

Real-world use cases make things even richer. For example, if a user uploads a JSON file and asks: “who runs finance?”, the chatbot must read and understand the file using a retrieval-augmented generation (RAG) pipeline. It should support different file types (JSON, CSV, plain text) and give clear, accurate responses. Users may also ask for custom behavior, like “flag nodes with sales over a million,” so the NLP system should allow

for flexible, intuitive commands. If external AI services (e.g., OpenAI) are used to assist, such as choosing which attributes appear in tooltips, the system must handle these results smoothly and integrate them naturally into the interface.

1.2.2 Visualization Challenges

The visualization component must clearly show hierarchical structures while allowing users to interact with and modify the display. Choosing the right layout is the first step. For small datasets, tree diagrams work well to show parent-child relationships. But for larger hierarchies, such as the structure of a multinational company, they can become cluttered and hard to read. Force-directed graphs offer a more flexible layout by placing nodes based on relationships, though they may lose the sense of strict hierarchy. The system should allow switching between formats depending on the use case, balancing clarity and speed.

Users should also be able to adjust visuals dynamically. Commands like "color nodes by department" or "highlight sales over 500" should update the view instantly. The system must interpret the user's command, find the right data attribute, apply logic, and refresh the visualization, even if some data is missing.

Other important features include filtering (e.g., "show only finance team"), zooming into specific branches (like one family line), and even generating code to customize visuals. Large language models (LLMs) could help generate code for tasks like coloring, but the system should always verify that the code is correct and fits the current visualization. Helpful tools such as an undo button or a legend that updates with the colors, make the interface more user-friendly but also add complexity. Grouping nodes into clusters (e.g., by role) can make the data easier to understand, but it can also be demanding on performance. Custom tooltips and pie charts for attributes like "role and salary" add flexibility, especially when working with non-numeric data. All of this requires a powerful and adaptable rendering engine.

1.2.3 System Challenges

The system as a whole must support smooth and consistent interactions. For example, if the user first says "show the org chart" and later says "highlight its top," the system should understand that "top" refers to the root nodes of the current chart. This requires a conversation tracker that keeps context over time.

Scalability is also essential. Large datasets, such as social networks or corporate structures with thousands of nodes, require optimized data handling and rendering to ensure the system stays responsive. This includes fast file parsing, real-time querying, and efficient visual updates.

Processing needs go beyond just showing visuals. Commands like "recolor by department" on massive datasets push the limits of performance. Due to the data including numbers, categories, and text, the system must be able to handle and display these mixed

types properly. In real-world scenarios, data is often incomplete, for example, missing salaries or parent links, so the system needs to handle gaps gracefully and still show what is available. For large operations, progress bars can help manage user expectations. Maintaining consistency between visual elements and chatbot responses, such as how attributes are named and displayed, also helps tie the experience together.

1.3 Requirements

In this section we summarize the requirements for the system, which are divided into functional and non-functional categories. Functional requirements specify what the system should do, while non-functional requirements outline how the system should perform.

1.3.1 Functional Requirements

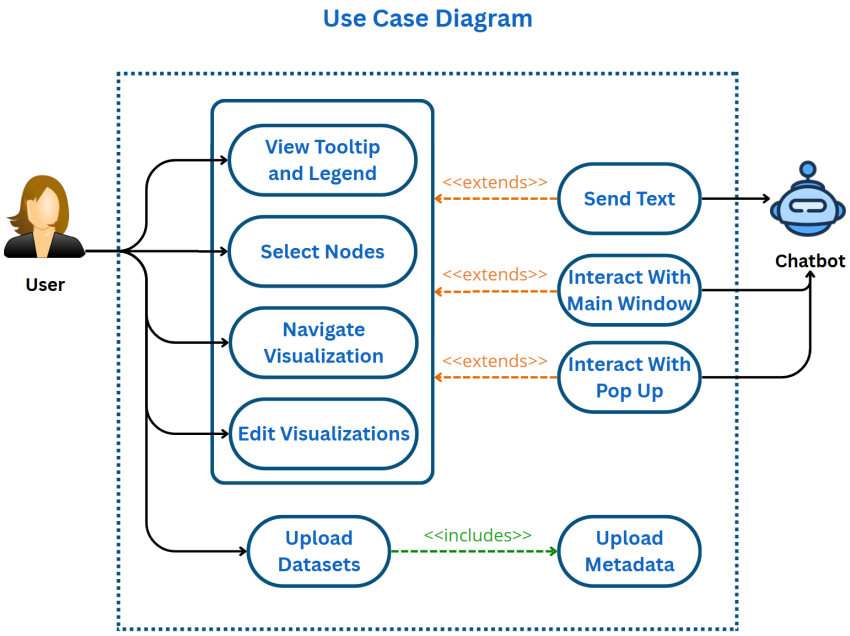


Figure 1.1: Use Case Diagram of the functional requirements.

Figure 1.1 represents the functional requirements of the system through a Use Case Diagram. It illustrates the interactions between two main actors: the **User** and the **Chatbot**, as well as the main functionalities they are involved in.

The **User** can interact directly with the data visualization interface using traditional graphical means. These include:

- **View Tooltip and Legend:** allows the user to access contextual information about elements in the visualization.
- **Select Nodes:** enables the user to highlight or select specific elements within the data visualization.
- **Navigate Visualization:** refers to the ability to move through the visualization space, zoom, or pan to explore the dataset.
- **Edit Visualizations:** allows the user to modify aspects of the visualization, such as styles or properties.
- **Upload Datasets:** enables the user to input new data into the system.
- **Upload Metadata:** involves providing descriptive information associated with the uploaded datasets and is marked as a required sub-function included when uploading datasets.

The **Chatbot** acts as a secondary interface that can extend certain functionalities. For instance, users can optionally:

- **Send Text:** communicate natural language queries or commands to the Chatbot, which can trigger interactions with the system.
- **Interact With Main Window / Pop Up:** through text input, the user can indirectly navigate or manipulate the main interface or associated pop-up dialogs.

1.3.2 Non Functional Requirements

The platform must exhibit high performance and scalability, efficiently handling large hierarchical datasets containing thousands of nodes without perceptible delays. All user interactions—whether dragging, zooming or resizing visual elements—must be reflected in real time, and the system should seamlessly adapt to changes in window dimensions or dataset updates without any loss of responsiveness. In parallel, the interface itself must remain intuitive and user-friendly across a variety of devices and screen sizes, offering context-aware, visually appealing animations (for example, an adaptive welcome screen) that enhance the user experience without introducing clutter or latency.

The system should detect and recover gracefully from edge cases such as malformed JSON hierarchies, missing fields, or incompatible metadata, providing clear and informative feedback rather than failing silently.

From a development standpoint, maintainability and extensibility are important. The codebase should be organized into modular components with comprehensive documentation, to help add or modify chatbot intents, visualization types, or data-processing actions simply by updating configuration files—without deep changes to core logic. The chatbot component, in particular, must sustain a high intent-classification accuracy after retraining to ensure a reliable interpretation of natural language commands.

Chapter 2

Related Work

This chapter reviews the state of the art in visualization-oriented natural language interfaces and examines the techniques reported in the literature for designing such interfaces. Section 2.1 reviews existing visualization tools and their limitations. Section 2.2 discusses the NLP techniques that underpin conversational interfaces. Finally, Section 2.3 highlights deficiencies that motivate our approach.

2.1 Existing V-NLIs

As data-driven decision-making becomes increasingly important, visualization platforms have evolved to make complex analysis more accessible—especially through natural language interfaces. By integrating conversational agents, these systems allow users, even those without technical expertise, to explore and understand data more intuitively. Foundational tools like NL4DV, LIDA, Haystack, and LangChain have set the stage, while more application-specific systems such as FlowNL, Evizeon, and Snowy demonstrate how natural language can drive data exploration. However, despite these advancements, most platforms still focus on simple, tabular data and offer limited support for hierarchical structures. Additionally, the visualizations they generate are typically static and non-interactive—users can issue natural language queries, but cannot, for example, click on a part of a chart to ask specific questions about it. A recent scoping review by Kavaz et al. [11] highlights that while many systems support natural language interaction, few handle hierarchical data effectively or allow dynamic chart manipulation. This chapter reviews the current landscape, including a specialized system for time series data, and highlights how our proposed approach fills these gaps by focusing on customizable hierarchical views that users can extend and refine through conversation.

2.1.1 FlowNL

FlowNL is a natural language-driven visualization tool designed to generate standard charts from tabular data through user queries [8]. For example, as shown in the provided image 2.1, a user can input a query like show the upward flow' or show flow between [region] and [region]' to visualize flow data, such as hurricane trajectories, with the tool producing corresponding flow maps alongside basic charts (e.g., bar charts). Its strength lies in its simplicity and accessibility, making it suitable for users who need quick visual representations of flat datasets. Additionally, FlowNL integrates spatial and numerical attributes into visualizations, as demonstrated by the image's depiction of various flow types (e.g., upward flow, tiny spiral flow) with color-coded patterns. This capability is underpinned by a declarative language that filters and combines data to render visualizations, translating natural language queries into structured specifications.

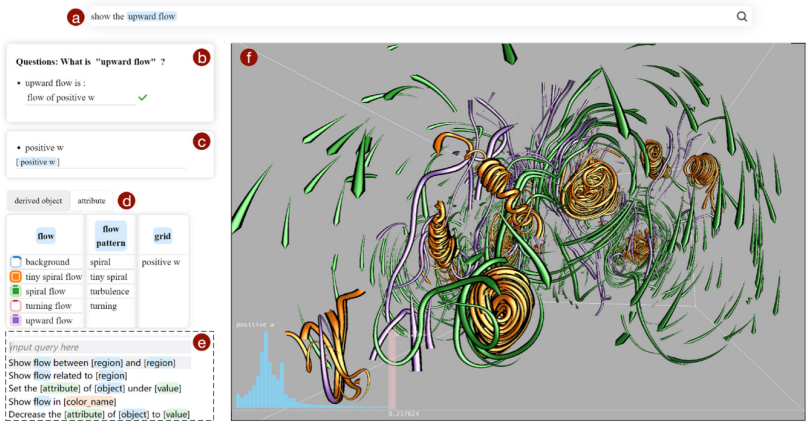


Figure 2.1: Sample interface of FlowNL, showing a flow map generated from user queries about flow data, such as upward flow and flow between regions, alongside a histogram, as depicted in the provided image [8].

However, FlowNL has significant limitations when it comes to more complex data structures and analytical needs. It is incapable of producing hierarchical visualizations, such as tree diagrams, network graphs, or any other form of nested or relational data representation [8]. This is a critical drawback for users dealing with data that inherently contains hierarchical relationships, such as organizational structures or interconnected systems, where flat representations obscure important insights. According to the scoping review from Kavaz et al. [11], FlowNL's focus remains on tabular and flow data with fixed visual mapping. Moreover, FlowNL lacks automated insight generation. While it can generate charts based on explicit user queries, as seen with the image's query options, it does not provide suggestions for further analysis or highlight key patterns or anomalies in the data. This forces users to manually decide what to visualize next, which can be inefficient and may lead to missed opportunities for deeper exploration. Additionally, FlowNL does not support the creation of derived tabular charts without manual intervention. For in-

stance, if a user visualizes flow by region and then wishes to see a breakdown by another attribute, they must issue separate queries rather than having the system automatically generate related charts [11].

2.1.2 Evizeon

Evizeon advances the field with a chatbot-like interface that allows users to interact with the system conversationally, auto-generating multiple chart types based on natural language queries [7]. For example, a user might ask, 'Visualize sales trends over the past year,' and Evizeon could produce a line chart. It also supports spatial queries, such as 'houses less than $1M in Ballard,' generating a map with clustered markers alongside a line chart of sales trends, and can adapt to follow-up queries, for instance refining the visualization to show townhome sales under $1M in the same neighborhood. See Figure 2.2. The system handles tabular data with attributes such as nominal, numerical, spatial, and temporal, producing basic visualizations like bar charts, scatter plots, line charts, and maps [11]. Evizeon leverages the CoreNLP toolkit for natural language processing, enhancing its ability to interpret user intents.

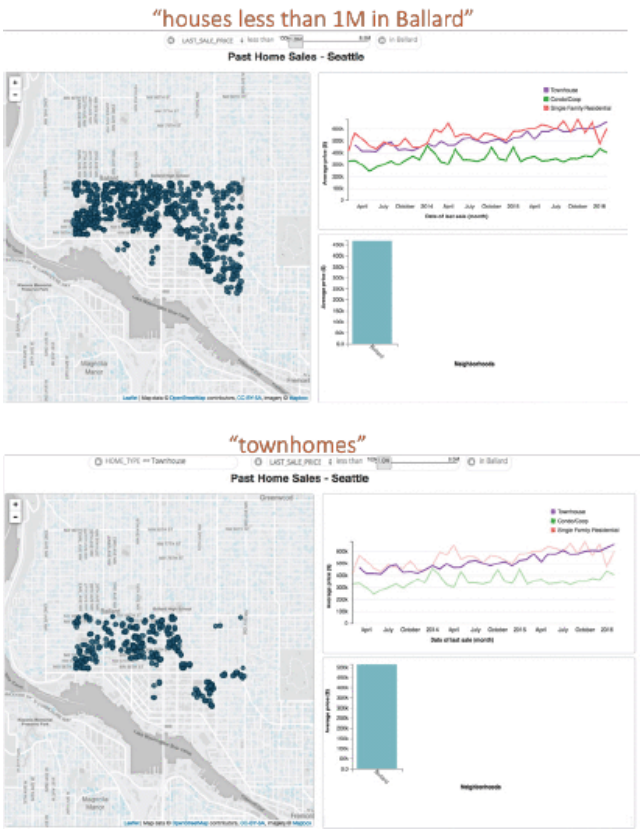


Figure 2.2: Evizeon's chatbot interface, displaying a line chart generated from a user query about sales trends [7].

Despite these strengths, Evizeon cannot create hierarchical visualizations, such as tree maps or network graphs, limiting its utility for nested or relational data [11]. This restriction confines users to flat charts, which are inadequate for representing complex relationships. Furthermore, while its chatbot interface is interactive, Evizeon does not support full UI customization through conversational commands; modifications like layout changes require manual menu interventions, disrupting the conversational flow [7]. According to the scoping review from Kavaz et al. [11], Evizeon's visual mapping is fixed, lacking adaptability to diverse data structures beyond basic layouts. Evizeon also falls short in deriving new insights or automatically generating related tables from an initial visualization. For example, after displaying a sales trend chart for houses in Ballard, it does not suggest or create additional charts (e.g., sales by product within each region) without explicit user prompts [7]. This lack of proactive insight generation mirrors findings in the literature, where V-NLIs often require users to drive the analysis manually [11].

2.1.3 Snowy

Snowy distinguishes itself by focusing on user guidance, recommending suitable chart types based on the user's input query [19]. For instance, asking 'What's the best chart for sales data?' might prompt Snowy to suggest a bar chart or line chart, depending on the data context. Designed for tabular data with nominal, numerical, and temporal attributes, Snowy produces basic visualizations such as bar charts, scatter plots, line charts, and maps, using a rule-based visual mapping approach [11]. It employs the CoreNLP toolkit and provides intent recommendations to assist users, particularly non-experts, in formulating effective queries [19] (see Figure 2.3).

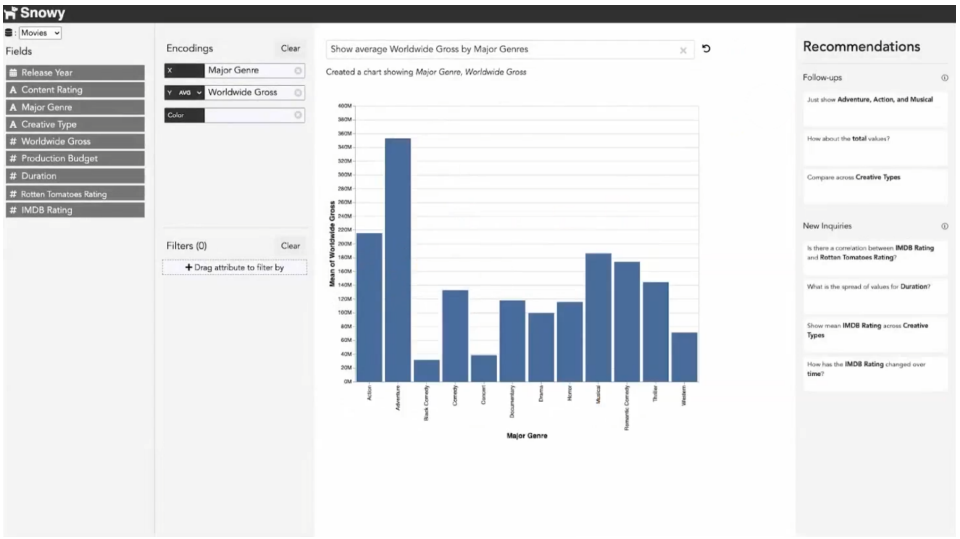


Figure 2.3: Snowy's recommendation interface, suggesting a bar chart for visualizing genres data [19].

However, Snowy does not support hierarchical visualizations like tree maps or network graphs, a limitation shared with FlowNL and Evizeon [11]. Its suggestions are confined to conventional charts, restricting its applicability to flat data structures.

2.1.4 Natural Language Interfaces for Specific Data Types

Beyond general visualization tools, some systems are tailored for specific data types. For instance, Wang et al. [21] presented NLTime, a system for interactive visual analysis of multivariate time series data using natural language queries. This system allows users to ask questions about their data in natural language and receive interactive visualizations as responses, demonstrating the potential of NLP in handling time series data, which is common in domains like finance and healthcare. NLTime combines natural language processing with visualization techniques to generate charts that answer user queries, such as time-based trends or correlations [21]. However, like other systems, it focuses on flat data structures and does not support hierarchical visualizations, limiting its applicability to complex relational datasets [11].

2.2 NLP Techniques in Conversational Interfaces

Conversational visualization systems rely on robust NLP pipelines to interpret user intents and generate appropriate visual or textual responses. In Evizeon, intent classification combines keyword matching with NLP toolkits like CoreNLP to identify user requests and generate charts [7]. FlowNL uses a declarative grammar to parse queries into visualization specifications, translating natural language into structured filters and chart parameters [8]. Snowy applies rule-based visual mapping combined with CoreNLP to recommend chart types based on query context [19]. Despite these efforts, such approaches often struggle with ambiguous or incomplete queries, and they generally rely on fixed visual mappings without deeper contextual understanding [11]. This project builds on these foundations by incorporating transformer-based models and advanced query-enrichment techniques to improve both intent classification and query processing.

2.2.1 Intent Classification and Query Processing

Many existing V-NLIs perform intent classification using rule-based or statistical methods. For example, Evizeon employs CoreNLP to extract entities and map them to visualization actions, while FlowNL relies on a domain-specific declarative language to translate user queries into chart specifications [7, 8]. Snowy's recommendation engine uses a combination of intent rules and basic NLP to suggest suitable chart types, but this can fail when queries do not match predefined patterns [19]. More recent research in related domains has shown that transformer-based classifiers, such as BERT, outperform traditional methods in intent detection by capturing semantic nuances.

In this project, transformer-based classifiers (e.g., BERT) are fine-tuned on a labeled corpus of visualization-related queries to identify user intents—such as requesting a hierarchical view, generating a specific popup chart, or filtering by spatial or temporal attributes. The classifier analyzes the query text and maps it to one of several predefined intents, improving accuracy over keyword-matching approaches. To the best of our knowledge, query augmentation methods have been used in information retrieval to refine vague inputs into structured commands but have not been explored in visualization chatbots. In this context, we implement a query augmentation step that, for instance, takes a user query like "show sales" and augments it to "show sales by region in a bar chart" based on session context and historical user preferences, thereby improving the mapping from natural language to visualization actions.

2.2.2 Contextual Enrichment via RAG

Retrieval-Augmented Generation (RAG) originates from the NLP literature as a technique that combines external data retrieval with generative models to produce contextually rich outputs. In conversational QA systems, RAG retrieves relevant documents or knowledge snippets before generating a response, ensuring factual accuracy and richer content. However, to the best of our knowledge, RAG has not been applied in visualization chatbots to integrate external metrics or documentation into dialog-driven visual analysis.

In this project, RAG is used to fetch relevant metrics, documentation, or previously generated visualizations (e.g., performance figures from uploaded files or metadata about selected subtrees). A retrieval component queries an index of available assets (such as CSV metadata or user-uploaded reports), and the generative component synthesizes explanations or suggestions. For example, when a user drills down into a hierarchy, RAG can retrieve the latest performance metrics for that subtree and generate a response like "According to the retrieved report, the selected subtree shows a 15% uptick in activity compared to last quarter." This approach addresses the static nature of existing systems like FlowNL and Evizeon by providing dynamic, context-sensitive insights and personalized recommendations [11].

2.3 Current Gaps and Motivation

Existing platforms lack unified support for hierarchical exploration, dynamic insights, and seamless transition between a **main hierarchical visualization** and auxiliary charts. Applications like FlowNL and Evizeon demonstrate how conversational interfaces can be embedded into user-facing dashboards; many of these applications rely on foundational tools—such as NL4DV—to support their functionality [11, 8, 7]. Table 2.1 summarizes the features covered in the analysis of the state of the art. As can be observed, only the proposed system supports hierarchical data handling and graph-based visualizations, among other advanced capabilities.

Table 2.1: Comparison of Mounted System Features

Feature	FlowNL	Evizeon	Snowy	Proposal
Natural Language Queries	✓	✓	✓	✓
Hierarchical Data Handling	✗	✗	✗	✓
Automatic Visualization Generation	✓	✓	✓	✓
Conversational Interaction	✓	✓	✓	✓
Multi-Visualization Support	—	✓	—	✓
Interactive Modifications	✗	✓	✗	✓
Graph-based Visualizations	✗	✗	✗	✓
RAG Integration for Contextual Insights	✗	✗	✗	✓
Multi-Modal Data Support	✗	—	✗	✓
User Context Awareness	✗	✓	✗	✓
Adaptive UI/UX	✓	✓	✓	✓
Personalized Recommendations	✓	✓	✓	✓

Hierarchical Exploration Limitations: None of the reviewed systems—FlowNL, Evizeon, Snowy, or NLTime—natively create or allow conversational edits of hierarchical force graphs [11, 8, 7, 19, 21]. Users cannot select nodes and request comparative tables or change visual encodings by attribute without leaving the **main hierarchical visualization**, limiting their ability to explore complex relational data through **interactive selection** and conversational commands.

Static Insight and Extension: Existing tools produce static charts without automated insights or follow-up visualizations. For example, after generating a chart, systems like FlowNL and Snowy do not suggest related analyses or derive new charts automatically [8, 19]. By treating the **main hierarchical visualization** as a dynamic interface, this project continuously extends it via conversational commands to generate suggestions, objectives, or pop-up tables. By organizing the system in this way, all derived tabular views ("popup charts") originate from the main hierarchical visualization, maintaining context and enabling an uninterrupted exploratory flow. Additionally, the concept of task-centric management, as explored by Chang et al. [3], could inspire visualization systems to organize multiple views efficiently, addressing the cognitive overload experienced in current tools.

Proposed Approach: Unlike existing tools, this project proposes a system that centers on hierarchical visualizations as the primary view. Users can fully customize this visualization through conversational commands covering two main categories:

- **Chart Creation and Extension**: Commands to create alternative visualizations on demand, such as circle packing charts for hierarchy or tabular charts (bar, line, pie, scatter, histogram, box, heatmap, bubble, sunburst, sankey).

- **Chart Editing and Data Interaction**: Fine-grained edits of node/link appearance (size, color, opacity, shape, force parameters), **interactive selection** and highlighting of nodes, zoom controls, and explanation requests that produce pop-up tabular charts for selected nodes.

Foundation-Level Features and Comparative Capabilities: Foundational tools underpin many user-facing conversational visualization applications. Table 2.2 compares the base capabilities of four such tools—NL4DV [20], LIDA [15], Haystack [5], and LangChain [12]—against the system proposed in this project. These tools vary in scope: NL4DV and LIDA focus on direct visualization generation; Haystack and LangChain specialize in Retrieval-Augmented Generation (RAG) pipelines. None, however, unify advanced natural language understanding, dynamic visualization creation, RAG-based insights, and hierarchical data support in a single system.

Table 2.2: Comparison of Base System Features

Feature	**NL4DV**	**LIDA**	**Haystack (RAG)**	**LangChain (RAG)**	**Proposal**
Natural Language Queries	✓	✓	✓	✓	✓
Hierarchical Data Handling	✗	~	✗	✗	✓
Automatic Visualization Generation	✓	✓	✗	✗	✓
Conversational Interaction	✗	~	✓	✓	✓
Multi-Visualization Support	~	✓	✗	✗	✓
User Context Awareness	✗	✗	✓	✓	✓
Graph-based Visualizations	✗	✗	✗	✗	✓
Interactive Modifications	✗	✗	✗	✗	✓
RAG Integration for Contextual Insights	✗	✗	✓	✓	✓
Multi-Modal Data Support	✗	✗	✗	✗	✓
Adaptive UI/UX	✗	✗	✗	✗	✓
Personalized Recommendations	✗	✗	✗	✗	✓

Chapter 3

Interface Design

3.1 Introduction

This chapter details the various components of the user interface, including the welcome page, dashboard controls, file upload mechanisms, main visualization area, chatbot sidebar, and popup visualizations. Each element has been carefully designed to provide an intuitive and seamless user experience, allowing users to explore and analyze complex datasets with ease. The interface is built using the Dash framework, which facilitates the creation of interactive web applications with Python, while the visualizations are powered by D3.js and Plotly.

3.2 Design Proposal

The design proposal outlines the conceptualization and planning of the system's user interface, user experience, and interaction flow, tailored to enable users to explore and manipulate hierarchical data visualizations effectively. This system, a web-based platform, aims to simplify complex data analysis by allowing users to interact with it using everyday language. The design prioritizes accessibility and intuitiveness, ensuring smooth transitions between user's interactions.

The platform's interface centers around a dashboard that integrates two primary components: the visualization area and the chatbot sidebar. The visualization area dominates the screen, displaying interactive representations of hierarchical or tabular data, such as force-directed graphs or bar charts, based on user queries. This section supports functionalities like zooming, panning, and node selection to enhance exploration. Adjacent to it, the chatbot sidebar resides on the left, providing a collapsible panel for users to input natural language queries and review conversation history. This sidebar can be hidden to maximize the visualization space when desired.

A header bar spans the top of the dashboard, housing essential controls such as buttons for uploading datasets and metadata, toggling the chatbot sidebar, and expanding the visualization view. This layout ensures that critical functions remain accessible without overcrowding the interface. For a general overview of the dashboard structure, refer to Figure 3.1. Detailed visuals and explanations of each component are provided in the following sections.

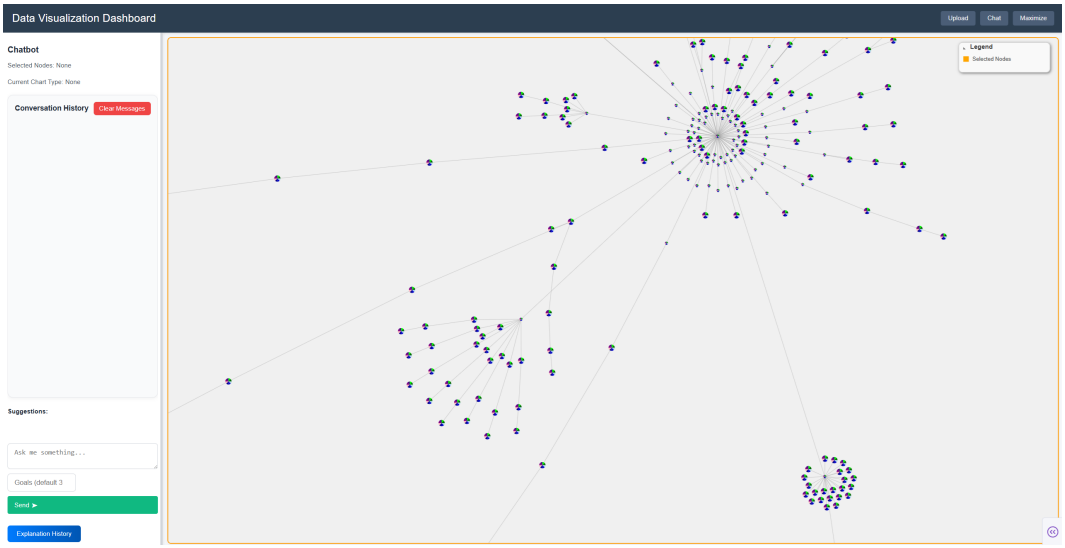


Figure 3.1: General Dashboard Layout Overview

3.2.1 Design Choices for Accessibility and User-Friendliness

Several targeted design decisions, informed by our example screenshots (Figures 3.1 and 3.10), work together to ensure the interface is both inclusive and intuitive:

- **Color and Focus Indicators**: Beyond a simple high-contrast palette for text and backgrounds, we use accent colors (e.g,bright orange borders) to indicate focus—whether on the main force-directed graph (Figure 3.1) or on a popup chart (Figure 3.10). This helps users quickly orient themselves, even in visually complex scenes.

- **Consistent Typography**: All labels, tooltips, and chat text use a clear, sans-serif font with adjustable size settings. In the chatbot sidebar (Figure 3.1), timestamps and user messages remain legible down to small sizes, while still fitting comfortably in the scrollable history pane.

- **Descriptive Iconography**: Every interactive control (e.g. Upload, Chat, Maximize buttons in the header) combines an icon with a text label and a hover tooltip. This redundant signaling supports both novice users and those relying on assistive technologies.

- **Legend and Contrast**: Chart legends (top-right of the main visualization) use distinct shapes and high-contrast borders to differentiate selected nodes from the rest, ensuring that color-blind or low-vision users can tell them apart by outline alone.
- **Responsive Layout**: The dashboard flexibly resizes—collapsing the chatbot sidebar to maximize visualization space on narrower viewports, or expanding pop-up overlays without obscuring the context underneath (see Figure 3.10). All controls remain reachable via keyboard navigation and maintain consistent relative positioning.
- **Contextual Suggestions**: The "Suggestions" area beneath the chat history (Figure 3.1) surfaces example queries based on current data focus, reducing cognitive load and guiding users toward meaningful interactions.

Together, these refinements balance visual clarity with functional depth, making complex data exploration approachable for users of all ability levels.

3.2.2 Simplifying Complex Interactions

To streamline complex tasks, the system incorporates UX principles such as:

- **Progressive Disclosure**: Options and information unfold gradually, preventing users from feeling overwhelmed.
- **Feedback Mechanisms**: Immediate responses, like confirming query comprehension or indicating visualization generation, keep users informed.
- **Error Handling**: The chatbot addresses ambiguous or erroneous inputs gracefully, offering clarifications or suggestions to maintain engagement.

These features ensure that users can navigate the system effortlessly, regardless of their technical background.

3.2.3 Interaction Flow

The interaction flow delineates the steps users take to engage with the system, from data input to insight generation.

Users initiate interaction by uploading a dataset (in CSV format) and corresponding metadata (in JSON format) via the welcome screen. Clear instructions and real-time feedback during this process confirm successful uploads or highlight issues, setting the stage for exploration.

Post-upload, users interact with the chatbot through the sidebar, posing natural language questions like "What's the distribution of toxicity levels?" or requesting specific visualizations, such as "Generate a force-directed graph of comment threads." The chatbot interprets these inputs, coordinating with visualization and data handling components

to produce results. Users can also refine existing outputs, for example, by saying "Adjust the node color to red."

The system delivers insights through interactive visualizations—such as hierarchical graphs or tabular charts—and textual explanations. Visual outputs appear in the visualization area, while explanations may accompany them or appear in the chatbot sidebar, enhancing user understanding. Figure 3.2 depicts this sequence, showing the transition from upload to query to insight.

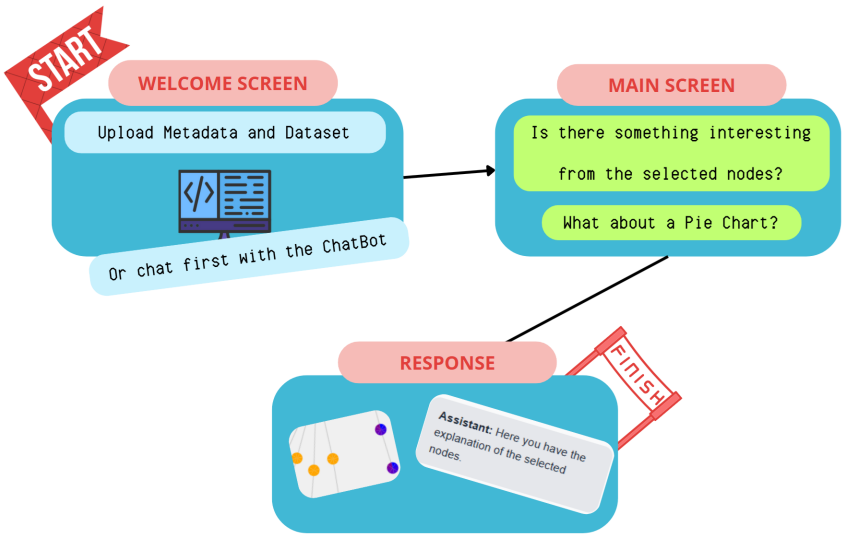


Figure 3.2: Interaction Flow Diagram

3.3 Welcome Page

The welcome page serves as the initial point of interaction for users, offering a clear and inviting entry into the system. As shown in Figure 3.3, the page features a prominent 'WELCOME' message created using a D3.js graph, which not only enhances visual appeal but also subtly introduces users to the system's data visualization capabilities. Below this, options for uploading datasets and metadata are presented, each accompanied by intuitive icons and brief descriptions. This design ensures that users can quickly grasp how to begin using the system, with clear visual cues guiding them through the initial steps. Accessibility is prioritized through the use of high-contrast text and icons, ensuring readability for all users. Additionally, the layout is responsive, adapting seamlessly to various screen sizes to provide an optimal experience on both desktop and mobile devices.

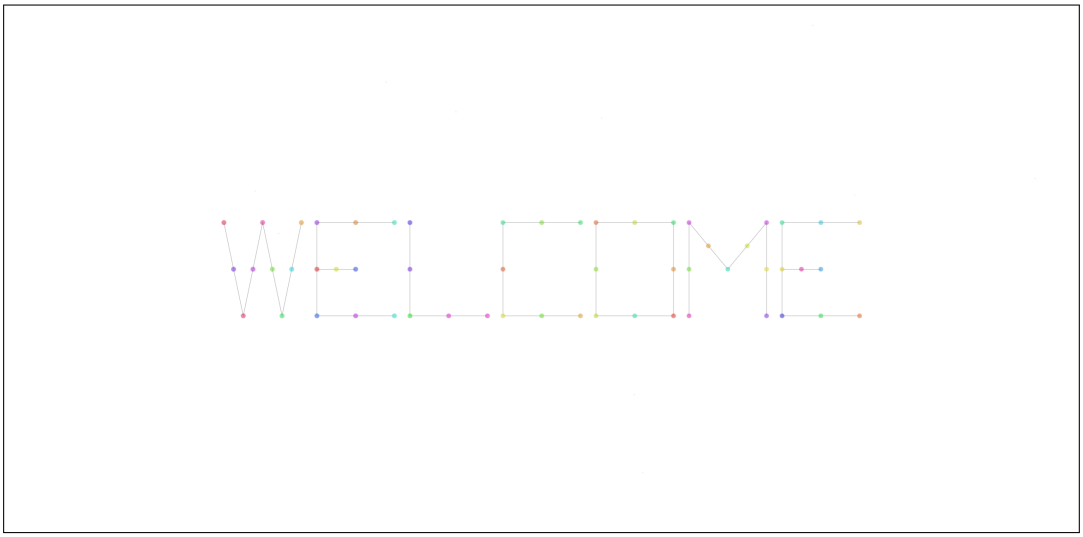


Figure 3.3: Welcome Page Design

Following the initial animation, the welcome page transitions to emphasize the upload options more prominently, as depicted in Figure 3.4. This gradual reveal focuses the user's attention on the next required action, reducing cognitive load and enhancing the manageability of the process.



Figure 3.4: Welcome Page Step 2

3.4 Dashboard Buttons

Once the necessary files are uploaded, users are directed to the main dashboard, which features a header with several control buttons. These buttons, illustrated in Figure 3.5, include options to open the upload modal, toggle the chatbot sidebar, and maximize the visualization view. Designed with a consistent style, the buttons employ a muted color palette that complements the overall interface without overshadowing the main content. Each button is clearly labeled and equipped with tooltips for additional guidance, ensuring that users can effortlessly access the system's features. The 'Upload' button facilitates the addition of more files to enhance the chatbot's context, the 'Chat' button controls the visibility of the chatbot sidebar, and the 'Maximize' button expands the visualization area for a more immersive experience.

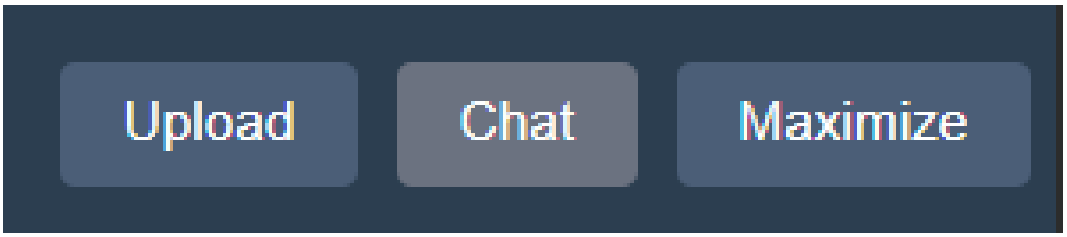


Figure 3.5: Dashboard Buttons

3.5 Upload Additional Files

To improve the chatbot's understanding of the dataset, users can upload supplementary files such as PDFs, Word documents, or Excel sheets. The interface for this feature, shown in Figure 3.6, is accessed via the 'Upload' button in the dashboard header. The upload modal provides explicit instructions and supports multiple file types, simplifying the process of adding extra information. The design prioritizes simplicity and clarity, featuring a clean layout and a prominent upload button. Upon uploading, the system processes these files to extract relevant data, subsequently enhancing the chatbot's response accuracy and enriching its knowledge base to provide more informed answers.

For example, if a user uploads a PDF containing guidelines on comment toxicity, the chatbot can reference this document when responding to queries about toxicity levels in comments. This feature not only broadens the scope of information available to the chatbot but also allows users to leverage external resources effectively, making the system more versatile and powerful in handling complex queries.

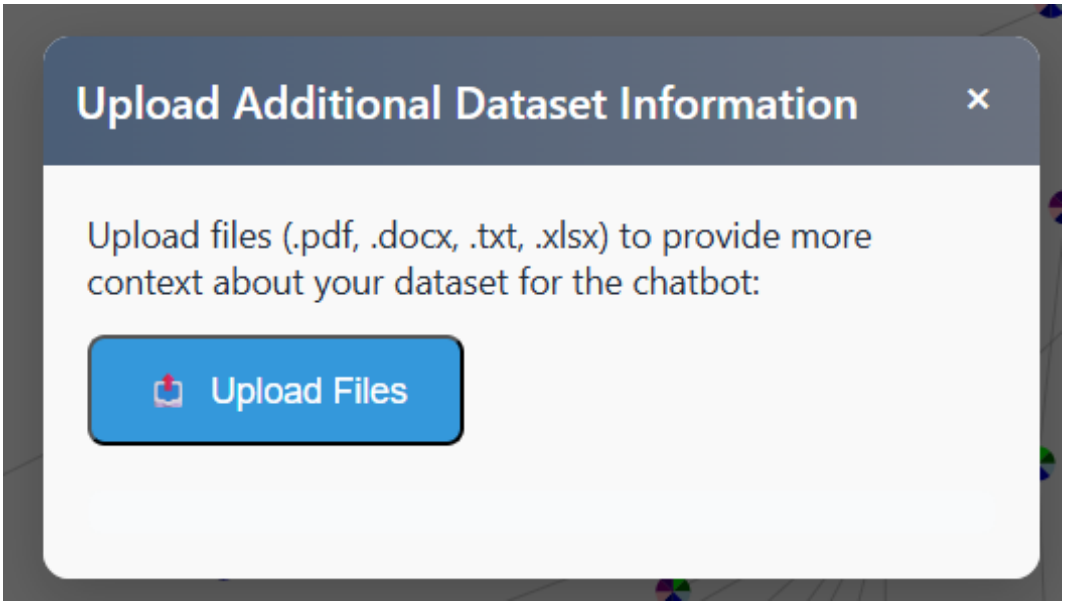


Figure 3.6: Upload Additional Files

3.6 Main Visualization

The main visualization area forms the core of the dashboard, enabling users to view and interact with hierarchical data representations. An example of a force-directed graph, one of the supported visualization types, is presented in Figure 3.7. This area is designed to be highly interactive, allowing users to zoom, pan, and select nodes, hover the nodes to show a tooltip to reveal additional information to explore the data in detail. The use of color and node size conveys additional information, such as toxicity levels or other attributes, rendering the visualization both informative and visually engaging. The force-directed layout elucidates the structure and relationships within the data, with nodes clustering based on their connections.

The main visualization has its own edges orange when the user is focusing on it, and the chatbot knows that the user is focusing on this visualization, so in future queries, it can provide more context-aware responses. This feature enhances the user experience by allowing the chatbot to tailor its replies based on the current focus of the user, making interactions more relevant and efficient. Also a legend is included to help users understand the meaning of different colors and node sizes, ensuring that the visualization is not only aesthetically pleasing but also informative. The design of the main visualization area emphasizes clarity and usability, with a layout that supports both novice and experienced users in navigating complex datasets.

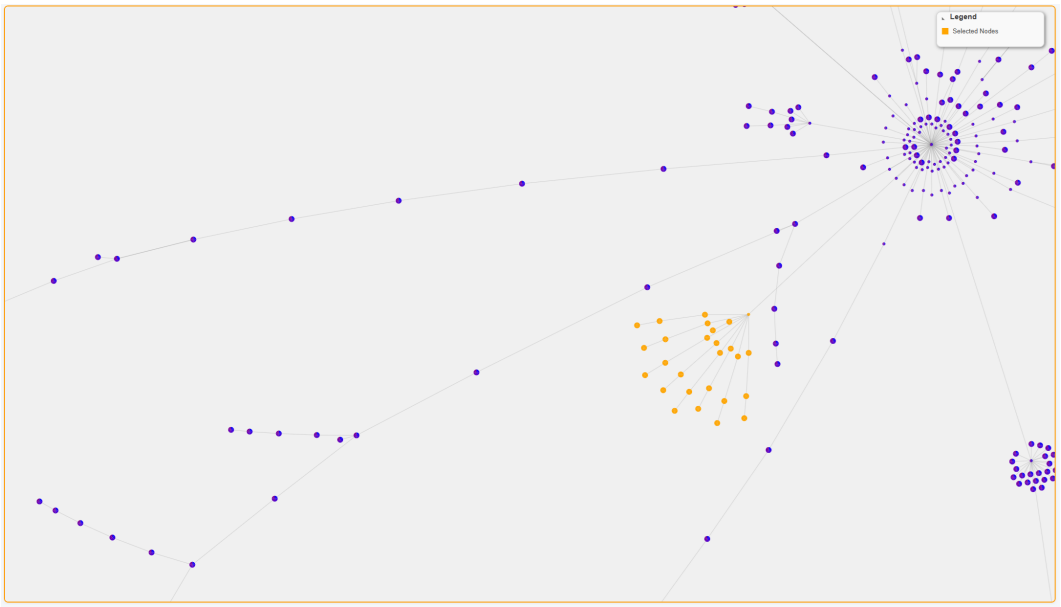


Figure 3.7: Main Visualization

3.7 Chatbot Sidebar

The chatbot sidebar, depicted in Figure 3.8, offers a conversational interface where users can query the system using natural language. It includes a text input area, a conversation history display, and buttons for clearing messages and accessing explanation history. The design emphasizes ease of use, featuring a clean and modern aesthetic that integrates seamlessly with the dashboard. The conversation history is housed in a scrollable area, enabling users to review past interactions effortlessly. Features such as auto-suggestions for queries and real-time feedback on query processing enhance user engagement, while the collapsible nature of the sidebar provides flexibility, allowing users to maximize the visualization area when needed.

The suggestion feature that is included, provides users with potential queries based on the current context of the conversation. This functionality is designed to assist users in formulating effective queries, particularly those who may be less familiar with the system's capabilities or the underlying data. A section where the user can request how many goals does he want to generate is also included, allowing the user to specify the number of goals he wants to generate.

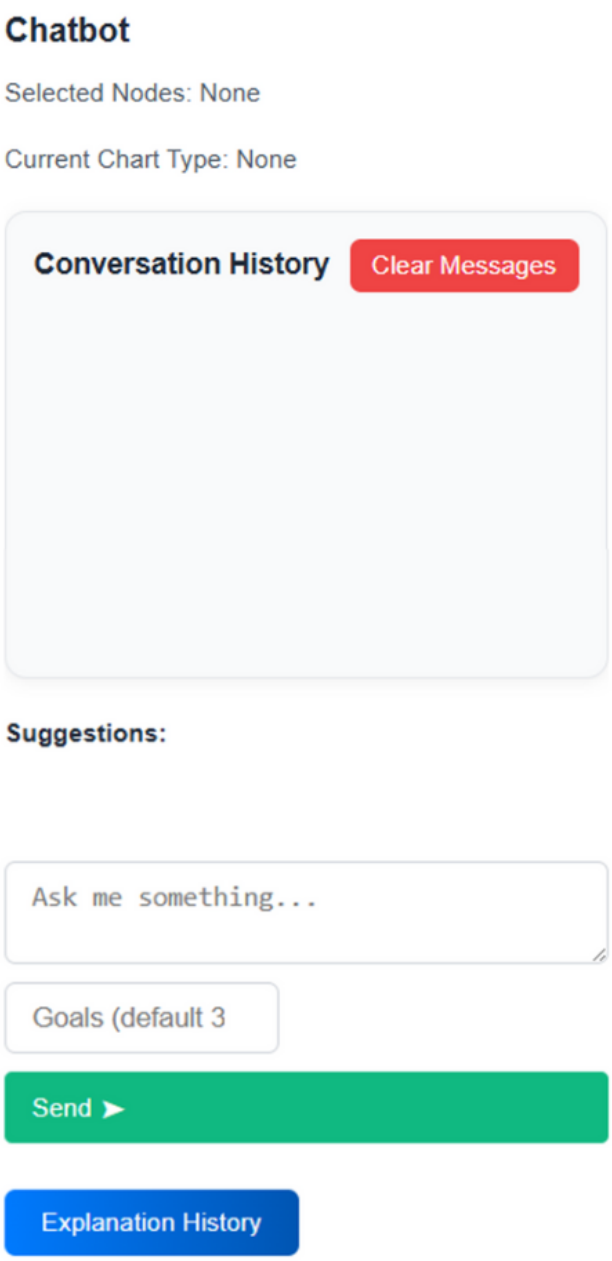


Figure 3.8: Chatbot Sidebar

Moreover, the chatbot maintains a history of explanations provided during the session, accessible via the 'Explanation History' button. This feature, illustrated in Figure 3.9, enables users to revisit previous insights and analyses, fostering a deeper understanding of the data.

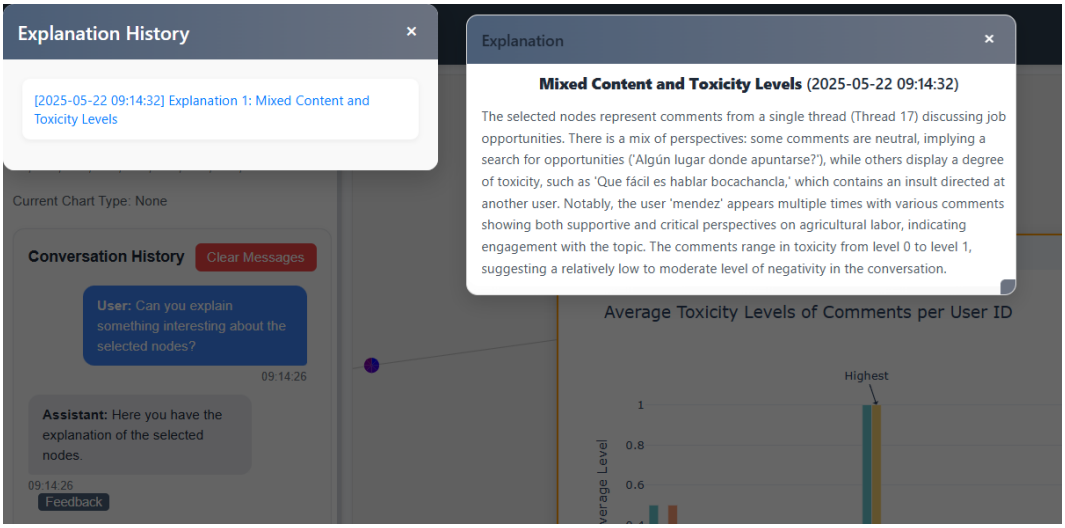


Figure 3.9: Chatbot Explanation History

3.8 PopUp Visualization

In addition to the main visualization, the system generates pop-up visualizations, such as tabular charts, based on user queries to the chatbot. Figure 3.10 presents a pop-up showing a bar chart of average toxicity levels. These popups are designed to deliver quick insights without interrupting the main workflow, appearing as overlay windows that can be easily dismissed or interacted with. The design ensures clarity and conciseness, with appropriate scaling to fit within the popup window, and includes interactive elements, such as the ability to select data points for further analysis.

Continuing in Figure 3.10, we see an example of a pop-up visualization that displays a tabular representation of data related to the user's queries. The bar chart within the pop-up provides an average toxicity levels of comments per User ID (in this case, from the user uploaded dataset). We note that the generated bar chart provides significant information as the highest toxicity of a user within all the users in the chart. Furthermore, it handles various features from the dataset and colours the bars with different colour contrasts to enhance readability and visual appeal. Looking at the edges of the pop-up, we can see that there is a thin orange line that tells the user that he is focusing on the pop-up visualization, and then the chatbot knows that the user is focusing on this pop-up and can provide more information about it. This is a feature that we have implemented to enhance the user experience, as it allows the user to focus on a specific visualization without losing the context of the conversation with the chatbot. The pop-up is above the main visualization, allowing the user to see both the pop-up and the main visualization at the same time. Under the pop-up, in the main visualization, we see some purple and blue colour fade nodes that are not selected, and the orange ones that are selected, they are the ones whose information is shown in the pop-up.

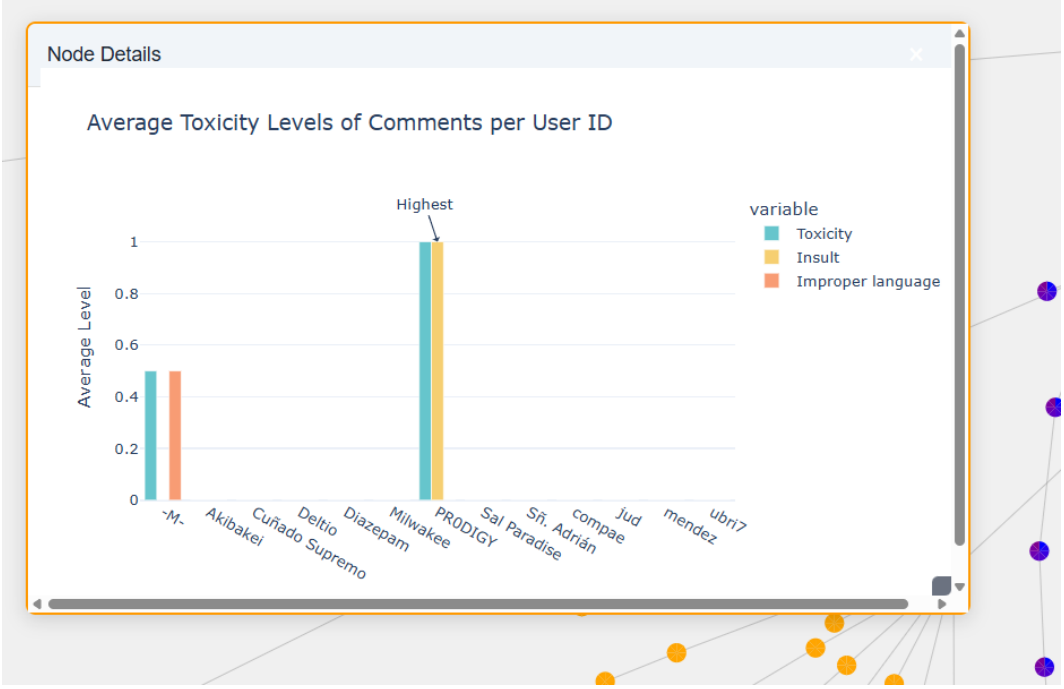


Figure 3.10: PopUp Visualization Example

Chapter 4

Hierarchical Interactive Visualization Architecture

This chapter explores the architecture and design of the system developed for this final degree project, providing a comprehensive overview of its structure and operational framework.

4.1 System Overview

Our architecture shown in Figure 4.1 is a web-based platform that transforms complex hierarchical data into intuitive, interactive visualizations, enabling users to explore and manipulate data using natural language. By integrating advanced Natural Language Processing (NLP), it processes user queries—such as requests for charts or display modifications—into dynamic visual outputs, making data analysis more comprehensible.

The main components are:

- **NLP Parser:** Interprets enriched user queries, translating them into actionable commands for the visualization framework.
- **Data Handler:** Manages and processes hierarchical datasets, ensuring efficient storage, retrieval, and manipulation.
- **RAG Module:** Enhances the system with external data retrieval and generative models, providing contextual insights based on user queries.
- **Reasoning Module:** Executes logical reasoning to support the generation of accurate responses and visualizations.
- **Visual Generator:** Creates dynamic visual representations, including charts and graphs, from processed data.

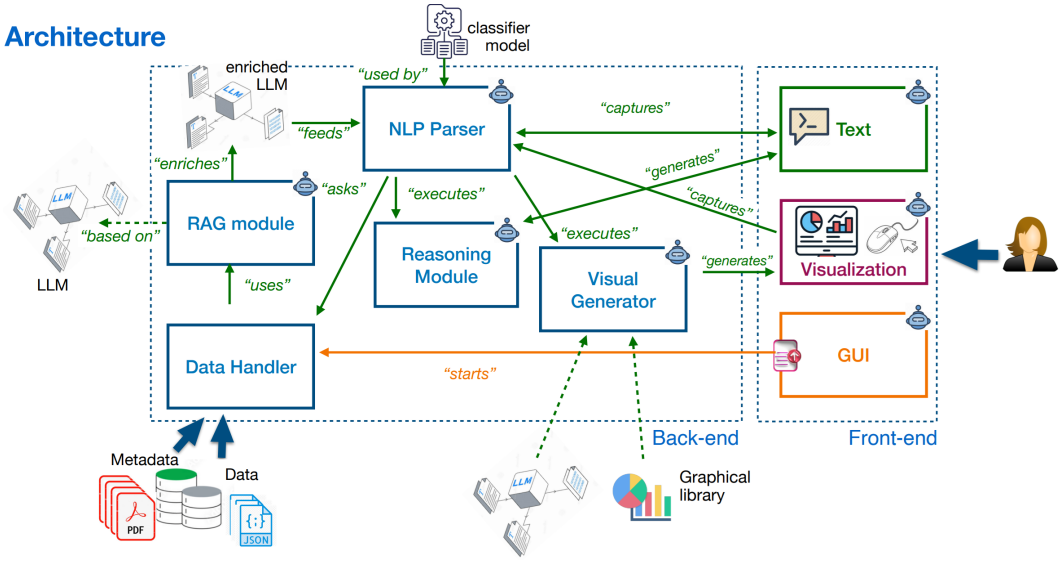


Figure 4.1: Diagram illustrating the full architecture.

The architecture is initiated by the Data Handler, which processes metadata and data (e.g., PDFs, JSON) and feeds it to the NLP Parser and RAG Module. The NLP Parser, enriched by LLMs, interprets user inputs and collaborates with the Reasoning Module to execute commands. The Visual Generator then produces visualizations, which are displayed via the GUI. The system is divided into back-end (handling data and processing) and front-end (user interaction) components.

4.2 NLP Parser

The NLP module operates as a structured pipeline, forming the core of the system's ability to interpret and act on user queries. It processes raw input, such as a user's command, through a series of refined stages to produce executable instructions.

The pipeline includes four key steps:

1. **Initialization**: Prepares the system by loading configuration files and setting up the tokenizer model.
2. **Query Processor**: Segments and enhances the user query, such as "Show me a chart," into a structured format.
3. **Cluster Predictor**: Identifies the query's general category, like "ChartCreation" or "ChartEditing," using a softmax model.
4. **Intent and Label Classifier**: Determines the precise action and labels, such as "CreateBarChart" and "GeneratePopUpChart" uses an enriched LLM.

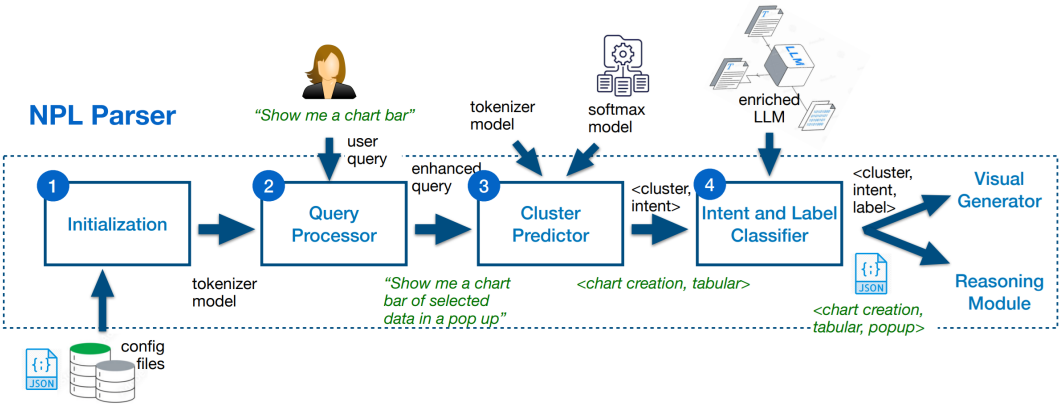


Figure 4.2: Diagram illustrating the query classification process.

This pipeline integrates rule-based techniques, machine learning, and language models to accurately handle varied or imperfect inputs, feeding results to the Visual Generator and Reasoning Module.

To effectively interpret user queries, the pipeline relies on four key concepts: (a) clusters, (b) sub-clusters, (c) intents, and (d) labels. These concepts are organized hierarchically to categorize and process queries systematically.

- **Clusters**: High-level categories that group related tasks the system can perform. For example:
  - "ChartCreation" groups all actions for creating new charts.
  - "ChartEditing" includes all actions for modifying existing charts.
  - "DataInteraction" covers interactions with data, such as selecting or zooming.
- **Subclusters**: Subcategories within a cluster that provide a more specific focus. For instance:
  - Under "ChartCreation", the subcluster "Hierarchical" includes actions for creating hierarchical charts like force-directed graphs.
  - The subcluster "Tabular" under "ChartCreation" groups actions for creating tabular charts like bar or pie charts.
  - Under "ChartEditing", the subcluster "MainChart" focuses on edits specific to the main chart, such as resizing nodes.
- **Intents**: Specific actions or goals within a cluster or subcluster. Examples include:
  - "CreateForceChart" (under "ChartCreation > Hierarchical") to create a force-directed chart.
  - "EditNodeSize" (under "ChartEditing > MainChart") to adjust node sizes in the main chart.

  - "GeneratePopupChart" (under "PopupChartGeneration") to create a new popup chart.

- **Labels**: Tags that describe the general purpose of an action, often tied to the cluster or subcluster level, and sometimes identical to intents. For example:
  - "CreateBarChart" (from "ChartCreation > Tabular") is a label for creating a bar chart.
  - "EditMainChart" (from "ChartEditing > MainChart") is a label for any main chart editing action, such as "EditNodeSize".
  - "ExplainFromSelection" (from "DataInteraction > Explanation") labels the action of explaining selected data.

Figure 4.3 shows this hierarchy: clusters (a) branch into subclusters (b), each containing specific intents (c), which are paired with corresponding labels (d).The role of this strategy in query classification is detailed in Section 4.2.3.

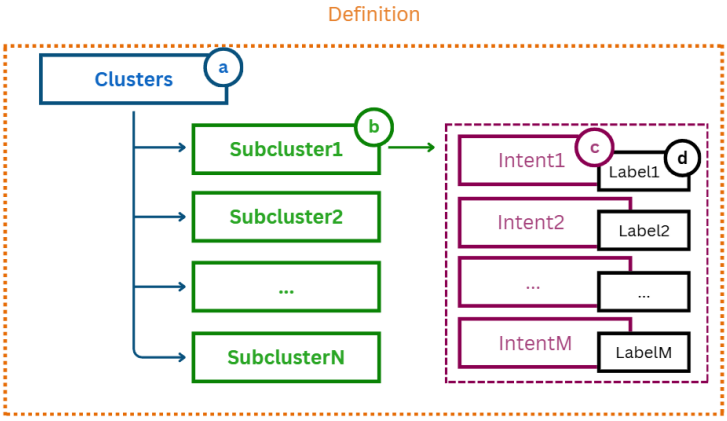


Figure 4.3: Diagram defining the clusters, subcluster, labels and intents.

For example Figure 4.4 illustrates the hierarchical organization of the "ChartCreation" cluster and its associated intents. The cluster branches into two subclusters: "Hierarchical" and "Tabular". The "Hierarchical" subcluster groups intents for creating hierarchical visualizations, such as "CreateForceChart" for force-directed graphs and "CreateCirclePacking" for circle packing charts. The "Tabular" subcluster includes intents for tabular visualizations, like "CreateLineChart" for line charts and "CreatePieChart" for pie charts, both of which can also trigger "GeneratePopUpChart" for additional popup displays. This structure demonstrates how clusters and subclusters organize related intents to handle specific visualization tasks efficiently.

Another example can be seen in Figure 4.5, that illustrates the structure of the "ChartEditing" cluster and its associated intents. The cluster divides into two subclusters: "MainChart" and "PopUpChart". The "MainChart" subcluster includes intents for editing pri-

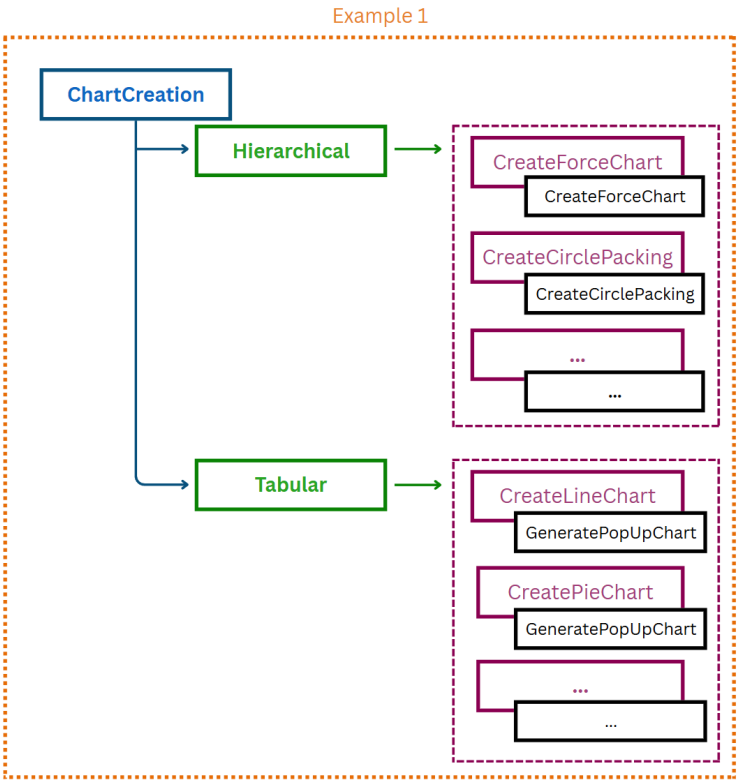


Figure 4.4: Diagram exemplifying the behaviour of the clusters and intents.

mary visualizations, such as "EditNodeSize" and "EditLinkColor", both labeled as "EditMainChart". The "PopUpChart" subcluster focuses on editing popup visualizations, with intents like "EditBarChart" and "EditLineChart", both labeled as "EditPopUpChart". This organization shows how the "ChartEditing" cluster categorizes editing tasks, enabling precise modifications to different chart types.

4.2.1 Stage 1: Initialization

The initialization stage prepares all components so the system can interpret and respond to user inputs. It involves:

- Loading configuration files that specify available chart types, term mappings, and permissible actions, providing the system with the background knowledge needed to interpret queries.
- Activating the language-processing module, which segments each user query into its constituent parts and analyzes their grammatical relationships—this enables the system to grasp the structure and semantics of the input.

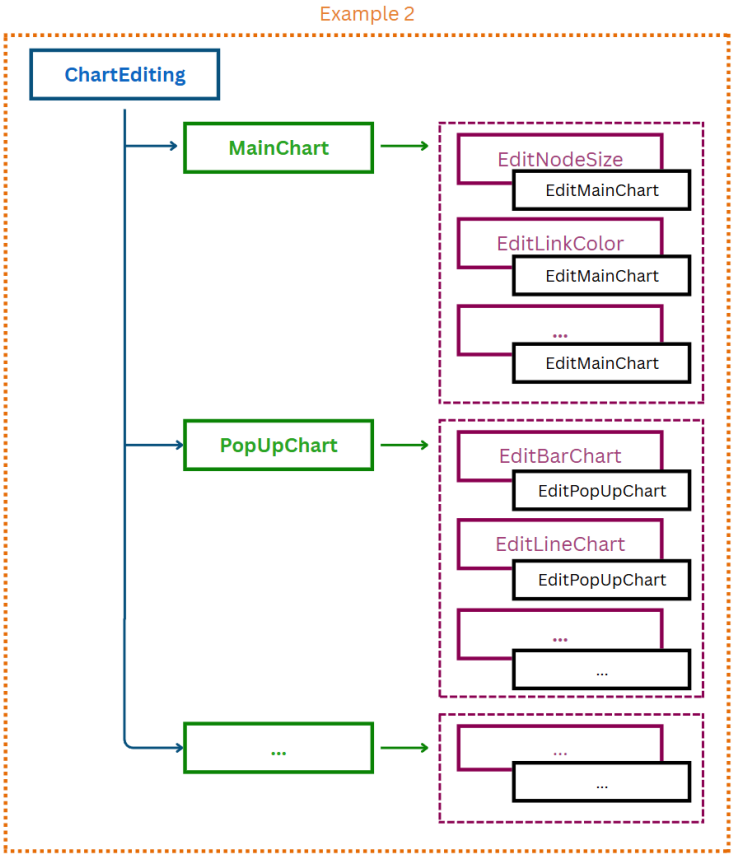


Figure 4.5: Second diagram exemplifying the behaviour of the clusters and intents.

- Initializing the intent-recognition engine by checking for an existing trained model in persistent storage. **If a suitable model is found**, it is loaded directly to expedite startup. **If no model is available**—such as on first deployment or after a reset—the system automatically trains a new one using the loaded configuration data and past interaction examples. Once training is complete, this model is saved for subsequent use.

- Performing a final validation of all components and resources to ensure the system is fully operational and ready to process queries without interruption.

This stage is important because it guarantees that the system's language-analysis and intent-detection capabilities are correctly configured and synchronized before any user requests are handled.

4.2.2 Stage 2: Query Processor

This stage processes a raw query, such as "Show me a barchart of sales data," to prepare it for analysis.

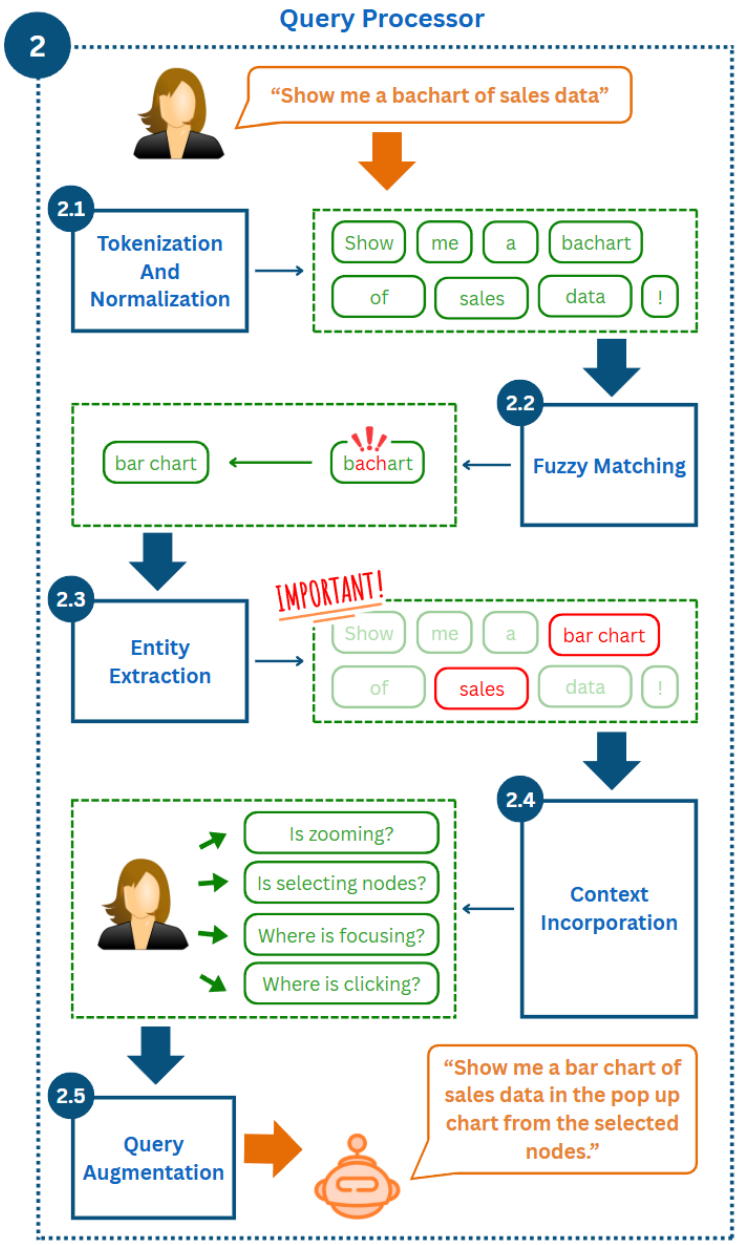


Figure 4.6: Diagram illustrating the query preprocessing steps.

The process unfolds as follows:

1. **Tokenization and Normalization**: Splits the query into tokens (e.g., [show", me", a", barchart", of", sales", "data"]) and normalizes them to lowercase.

2. **Fuzzy Matching**: Adjusts typos or variations, like correcting "barchart" to "bar chart" if the words are related.

3. **Entity Extraction**: Detects key elements, such as "sales" as a data column, when identified.

4. **Context Incorporation**: Integrates contextual cues, such as "popup" indicating a popup chart context.

5. **Query Augmentation**: Refines the query with added context, e.g., "Show me a bar chart of sales data in the popup chart from the selected nodes" if applicable.

4.2.3 Stage 3: Cluster Prediction

This stage classifies the query into appropriate categories through a structured process, as depicted in Figure 4.7.

The diagram illustrates the flow for a query like "Show me a bar chart of sales data in the pop up chart from the selected nodes."

- **Tokenization**: In step 3.1, the query is converted into tokens (e.g., [101, 226, 2304, 3673, 1997, 4341, 102, ...]), preparing it for analysis.

- **Probability Calculation**: Step 3.2 computes logits for each subcluster (e.g., [2.5, 0.9, 0.1, 0.05, ...]), which represent the likelihood (probabilities for model understanding) of the query belonging to categories like "ChartCreation > Tabular" or "ChartEditing > MainChart."

- **Softmax**: In step 3.3, these logits are transformed into probabilities for human understanding (e.g., [0.85, 0.15, 0.02, 0.03, ...]), with "ChartCreation > Tabular" scoring the highest at 0.85.

- **Cluster Selection**: Step 3.4 selects the subcluster with the highest probability, confirming "ChartCreation > Tabular: 0.85" as the query's category.

4.2.4 Stage 4: Intent and Label Classification

This final stage of the NLP pipeline is important for translating a categorized query into a precise, actionable command that the system can execute. It uses a large language model (LLM) to determine the exact intent behind the user's query and assign a corresponding label, ensuring that the system responds accurately to the user's request. The process is designed to handle nuanced queries by considering the query's context, the identified subcluster, and predefined system rules, such as constraints on visualization types.

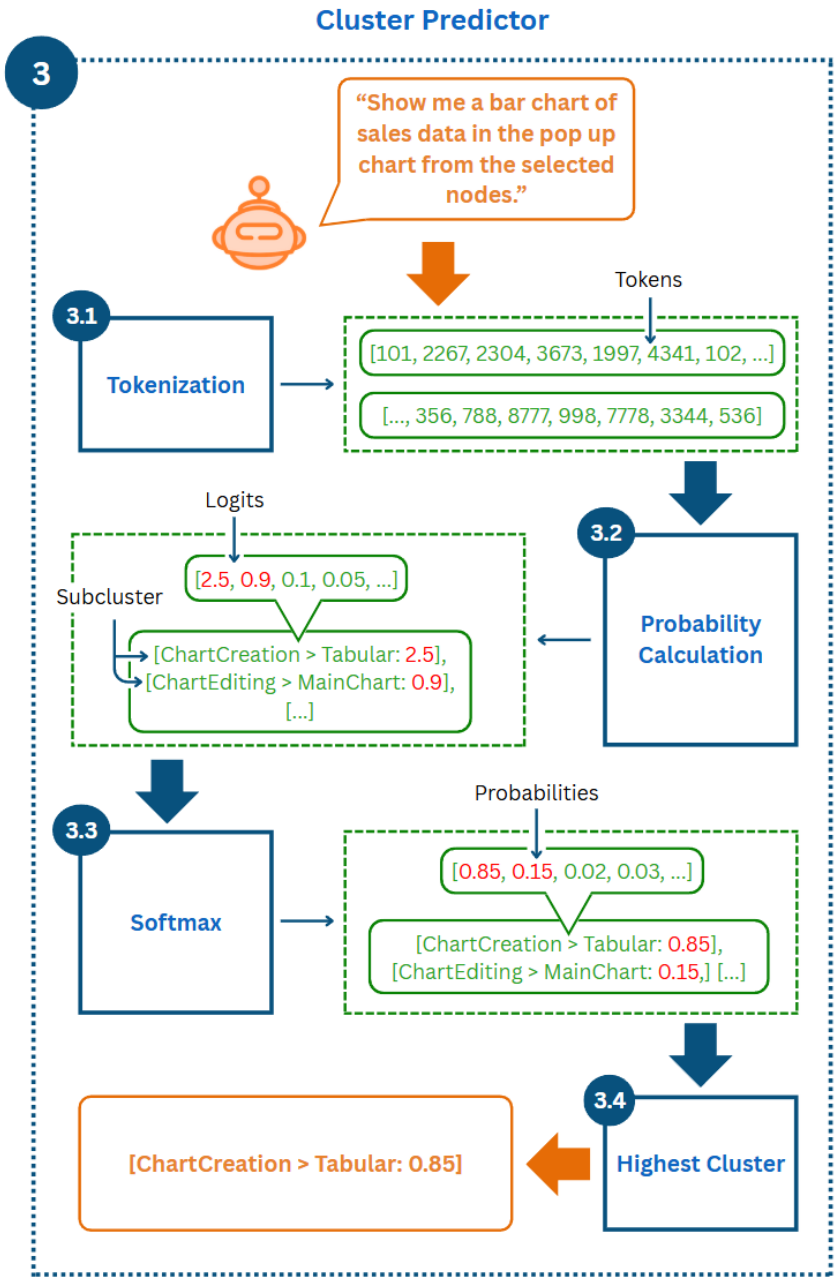


Figure 4.7: Diagram illustrating the cluster prediction process.

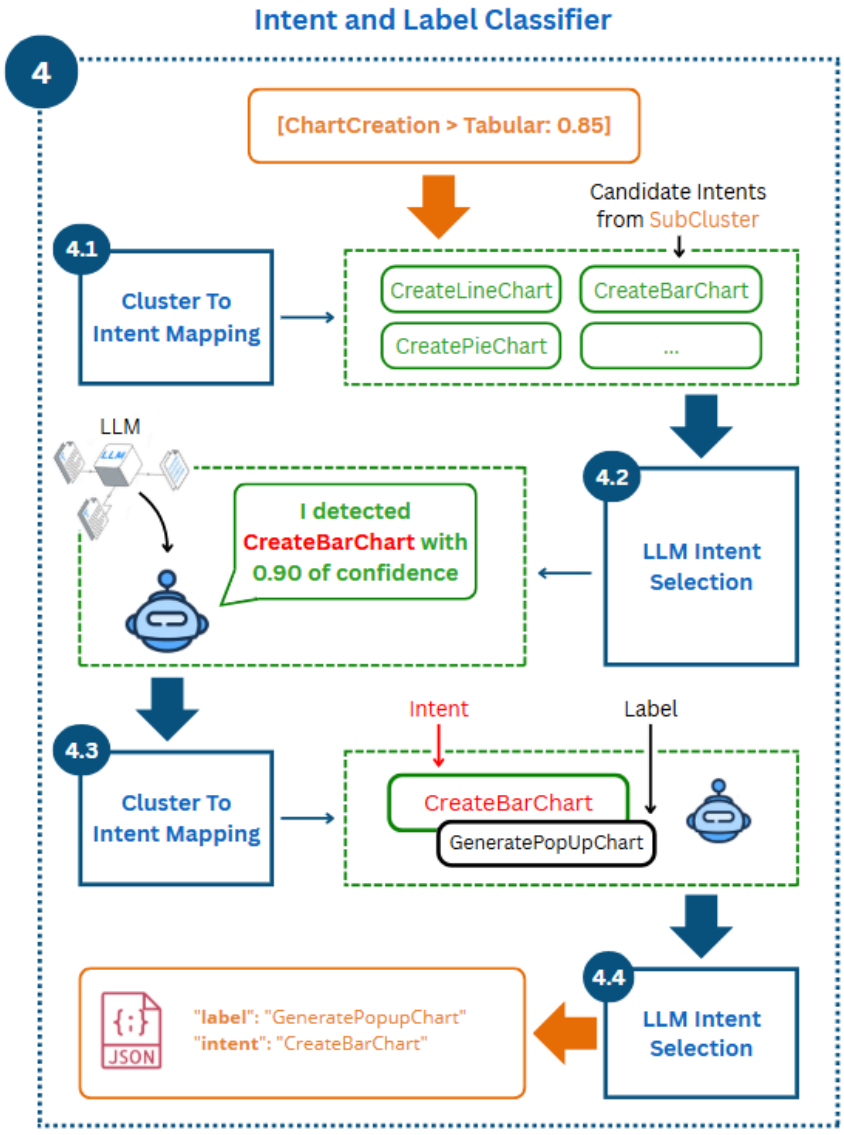


Figure 4.8: Diagram illustrating the intent and label classification process.

Figure 4.8 visually represents this process, which is broken down into four distinct steps.

The process begins with the input from the previous stage: a subcluster with its associated probability, such as "[ChartCreation > Tabular: 0.85]," indicating that the query (e.g., "Show me a bar chart of sales data in the pop up chart from the selected nodes") has been classified as a tabular chart creation task. The goal of this stage is to map this subcluster to a specific intent (the user's goal) and label (the actionable command) that the system can execute.

- **Cluster to Intent Mapping (Step 4.1)**: The first step involves mapping the identified subcluster to a set of candidate intents. For the subcluster "ChartCreation > Tabular," the system retrieves possible intents associated with tabular chart creation, such as "CreateLineChart," "CreateBarChart," and "CreatePieChart." This mapping is predefined on the basis of the system's understanding of what actions are available within each subcluster. The candidate intents represent the potential goals the user might have within the "Tabular" subcluster, and this step ensures that all relevant possibilities are considered.

- **LLM Intent Selection (Step 4.2)**: In this step, the LLM evaluates the candidate intents to select the most appropriate one based on the query and its context. The LLM is prompted with the query, the subcluster, and the list of candidate intents, along with any relevant contextual information (e.g., the query specifies a "pop up chart"). The LLM analyzes the query's language and context, determining that "CreateBarChart" is the best match for the request to create a bar chart, with a confidence score of 0.90. This high confidence indicates that the LLM is certain of its selection, ensuring that the system interprets the user's intent accurately.

- **Cluster to Intent Mapping (Labeling, Step 4.3)**: Once the intent is selected, the system maps it to a specific label that defines the actionable command. In this case, the intent "CreateBarChart" is paired with the label "GeneratePopUpChart" because the query specifies a popup chart context. This step also considers system rules, such as ensuring that only certain chart types (like tabular charts) are allowed in popups. The label "GeneratePopUpChart" indicates that the system should generate a bar chart within a popup window, aligning with the user's request. The pairing of intent and label ensures that both the goal (creating a bar chart) and the specific action (displaying it in a popup) are captured.

- **LLM Intent Selection (Final Output, Step 4.4)**: The final step consolidates the intent and label into a structured output, formatted as a JSON object for the system to process. The output includes the selected action, with "label": "GeneratePopUpChart" and "intent": "CreateBarChart," along with any additional metadata (e.g., confidence scores or context details). This JSON object is then passed to the next stage of the system (e.g., the Visual Generator or Reasoning Module) to execute the command. The structured output ensures that the system can interpret and act on the command seamlessly.

4.3 Visualization Generator

This section details how the system converts processed data and user commands into interactive visual representations, categorized into hierarchical and tabular types. Hierarchical visualizations, like circle packing or force-directed graphs, illustrate nested or interconnected data structures, while tabular visualizations, such as bar, line, or pie charts, present data in a simplified format, often derived from selected nodes in hierarchical views.

Figure 4.11 outlines the process. In step A.1, the Visual Generator receives commands like "[CreateCirclePacking, EditMainChart, ...]" for hierarchical tasks or "[GeneratePopUpChart, EditPopUpChart, ...]" for tabular ones. Step A.2 checks the visualization type, routing hierarchical commands to the Hierarchical Tool Executor (A.3), producing corresponding code. Tabular commands are sent to the Tabular Tool Executor (A.4), also generating executable code.

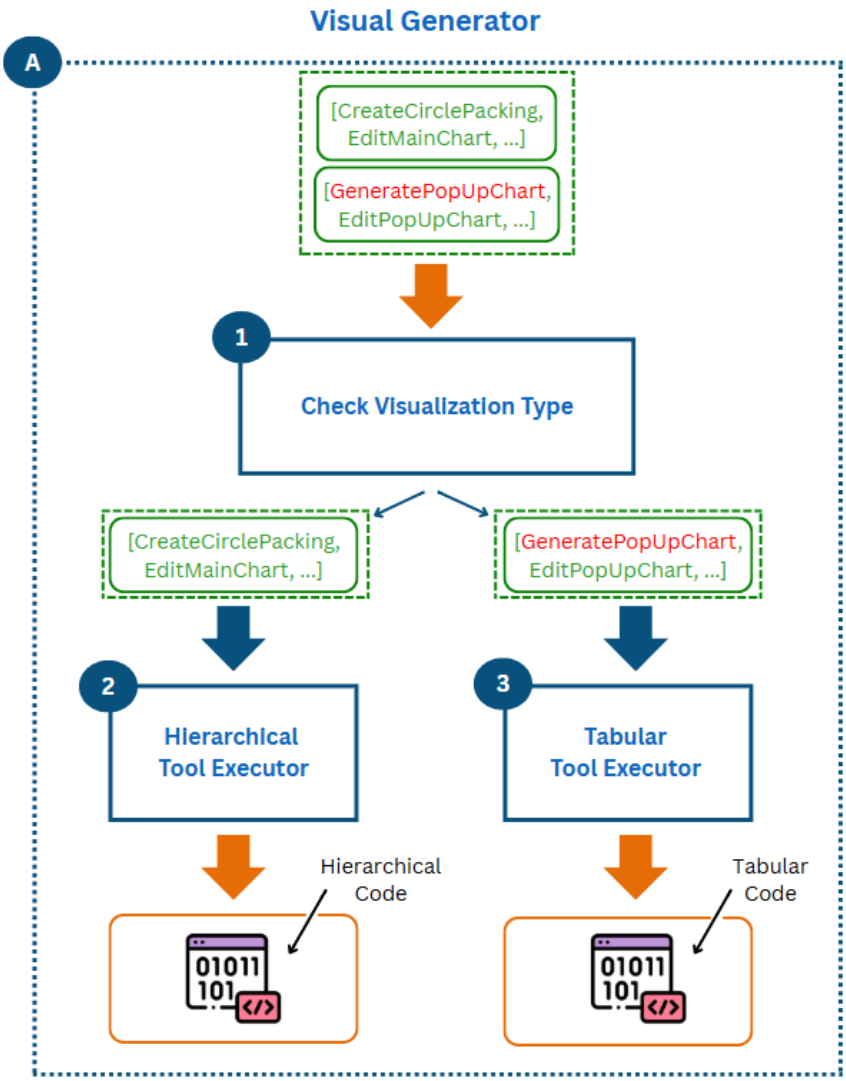


Figure 4.9: Diagram illustrating the visualization generation process.

4.3.1 Hierarchical Visualizations

Hierarchical visualizations effectively uncover the structure and connections within intricate datasets, such as those with inherent hierarchies (e.g., a family tree) or networked entities (e.g., social media interactions). These visualizations are created from a preprocessed hierarchical data structure, where nodes can represent for example entities like users or comments, and links depict relationships, such as replies between comments.

Figure 4.10 illustrates the process of generating these visualizations. In step B.1, the Hierarchical Tool Executor receives a command, such as "CreateForceChart," along with the processed data structure (e.g., nodes: [id: 1, text: "comment1", ...], links: [source: 1, target: 2, ...]). Step B.2 produces code for a hierarchical graph that visually represents the data's structure and relationships.

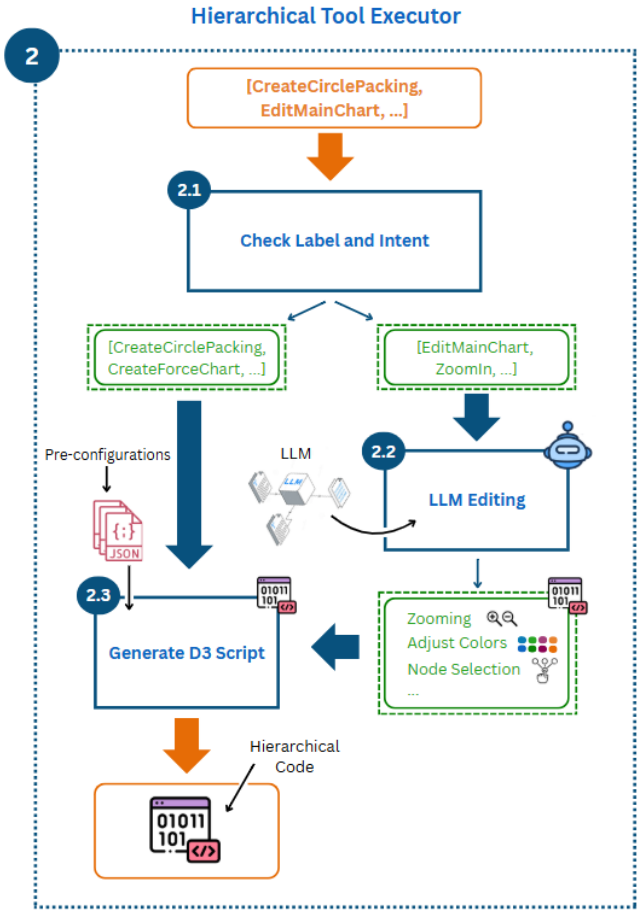


Figure 4.10: Diagram illustrating the hierarchical visualization generation process.

The system offers several types of hierarchical visualizations, including force-directed graphs and circle packing layouts. A force-directed graph shows entities as dots con-

nected by lines, where the layout adjusts dynamically to minimize overlap and clarify connections. For example, imagine a discussion thread where each comment is a dot, and lines show who replied to whom—the result is a map of the conversation's flow. Circle packing, on the other hand, nests smaller circles within larger ones to represent containment, such as comments grouped within threads. This choice depends on what the user wants to explore: connections might call for a force-directed graph, while nested structures suit circle packing.

To create these visualizations, once the system starts and reads all the data uploaded by the user, it creates the most useful hierarchical visualization for the data. Then, if the user wants to change the created graph, he can write a query, and the system analyzes it—say, "make a tree layout instead"—and matches it to the data's structure. It then uses tools that support hierarchical structures to render the layout. The design separates hierarchical visualization generation into its own component because these layouts require specialized algorithms to position elements clearly, unlike the simpler tabular charts. This separation keeps the system modular and efficient.

These visualizations are also interactive. Users can zoom in to focus on specific areas, click on nodes to reveal details, or hover over elements for more information. This interactivity makes exploring complex data engaging and intuitive. Figure 4.11 shows a force-directed graph of comments connected by replies, offering a glimpse into how these relationships come to life visually.

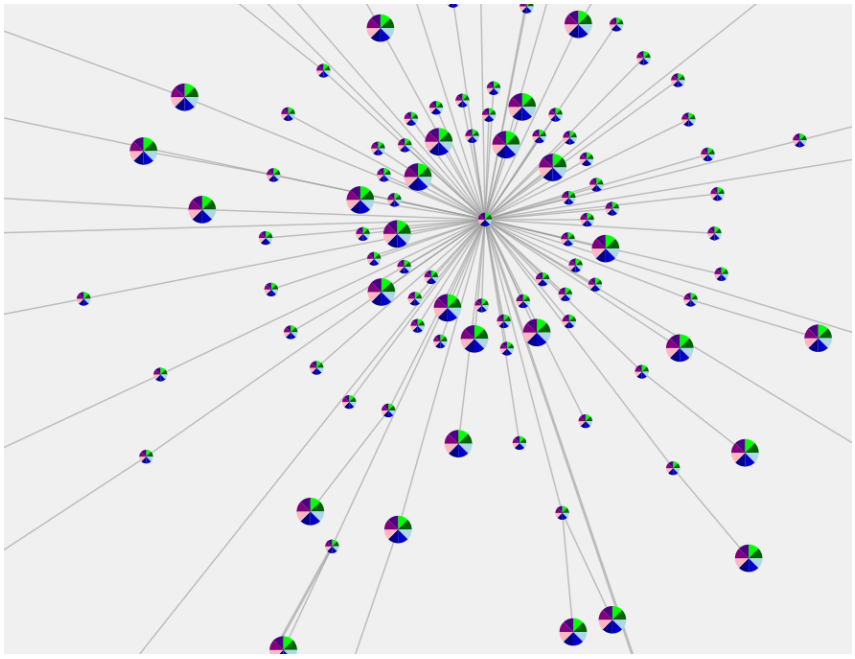


Figure 4.11: Example of a force-directed graph showing comment connections.

4.3.2 Tabular Visualizations

Tabular visualizations provide a clear and concise way of displaying data, enabling users to identify trends, patterns, and outliers with ease. These visualizations are well-suited for data structured in rows and columns, similar to a spreadsheet, and encompass common chart types like bar, line, and pie charts. The system generates these visualizations from tabular data, such as a table of toxicity levels or user counts, tailored to the user's specific request.

The system employs the Tabular Tool Executor (as depicted in Figure 4.12). The Code Generation process (Section 3.1) is designed to create code by leveraging user context, past visualizations, and a RAG (Retrieval-Augmented Generation) module for pre-configurations and additional information from extra files that the user might have that describes the dataset that is working on; second, it validates this code through a Code Validation step (Section 3.2) to ensure accuracy, marking it as "PASSED" or "FAIL" before rendering the final visualization, such as a pie chart or bar graph.

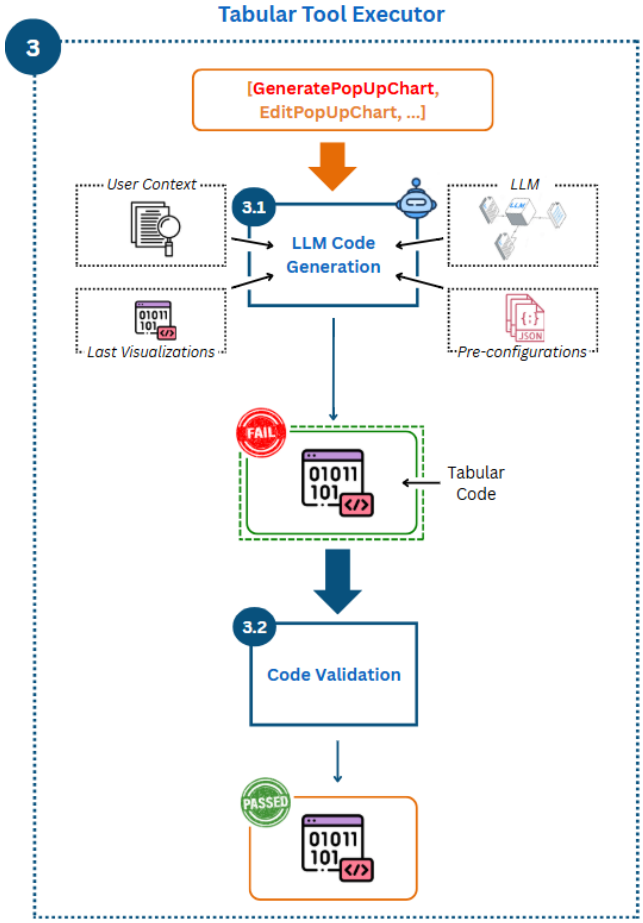


Figure 4.12: Diagram illustrating the tabular visualization generation process.

When a user submits a query, such as “show the distribution of toxicity levels,” the system interprets their intention. It might decide a pie chart works best for proportions, a bar chart for comparisons across categories, or a line chart for trends over time. This decision considers the query and the nature of the data — for example, dates might suggest a line chart. The system then selects the relevant data columns, configures the chart with labels and colors, and renders it, aiming not only to answer the query, but also to reveal additional insights.

The separation of tabular visualization generation into its own component reflects its distinct needs. Unlike hierarchical visualizations, these charts use standardized formats that do not require complex layout algorithms, allowing for quick and reliable creation. This modularity ensures that the system can efficiently handle diverse visualization tasks.

Like their hierarchical counterparts, tabular visualizations are interactive. Users can hover over bars to see exact values, click legends to filter data, or adjust views to dig deeper. Also this information is known by the internal chatbot (LLM) and questions about editing, filtering and all the things mentioned earlier can be done through the chatbot. This makes the charts more than static images—they become tools for exploration. Figure 4.13 displays a pie chart comparing toxicity levels across user groups, illustrating how these visuals bring data to life.

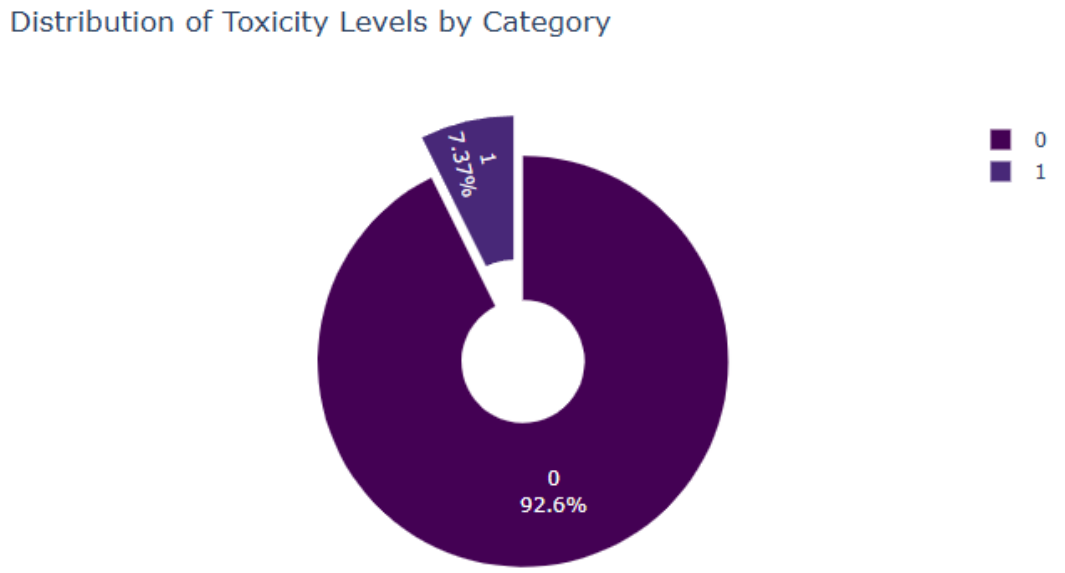


Figure 4.13: Example of a pie chart comparing toxicity levels.

4.4 Reasoning Module

The Reasoning Module improves the chatbot's ability to provide insightful responses by analyzing user interactions and generating thoughtful explanations, as depicted in the Reasoning Module scheme (Figure 4.14).

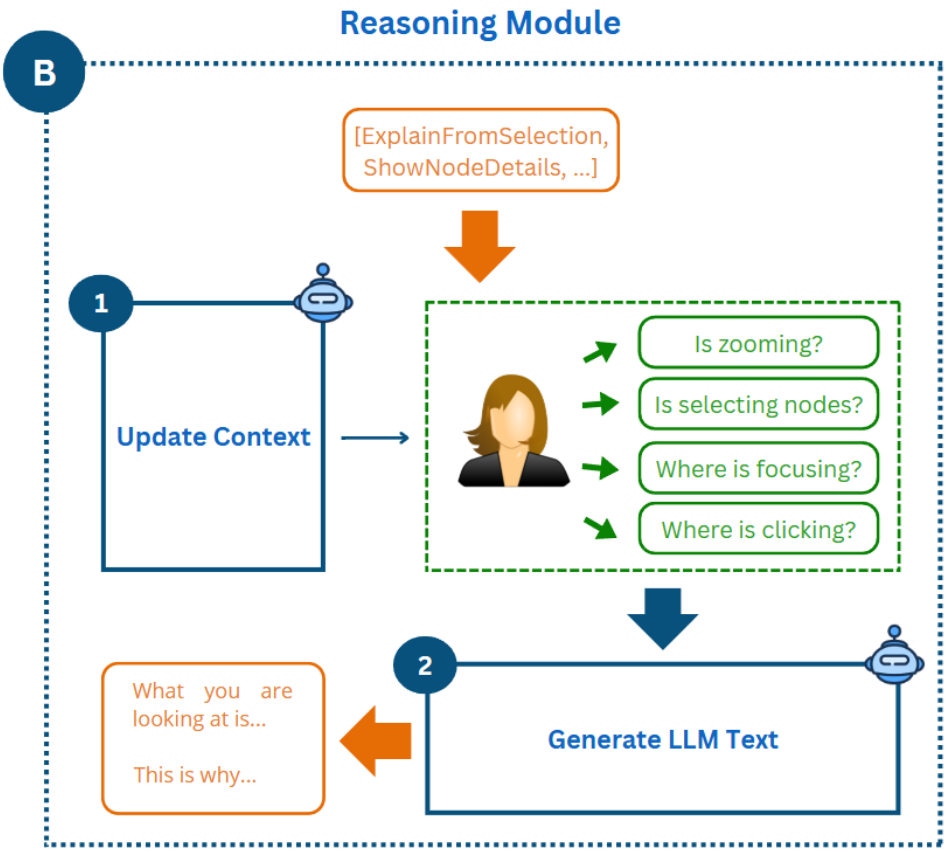


Figure 4.14: Schematic overview of the Reasoning Module's workflow.

This process starts with updating the context (Step 1), where the system gathers detailed information about what the user is doing—such as interpreting actions like "ExplainFromSelection" or "ShowNodeDetails"—and assesses the current focus, like whether the user is zooming in, selecting nodes, clicking on specific areas, or determining where their attention is directed. This context is then enriched with insights into what the user is looking at, such as "What you are looking at is..." or "This is why...", capturing the purpose or reasoning behind their actions. Next, the system generates the LLM text (Step 2), where it crafts a customized and detailed response based on the updated context, addressing questions like "Is zooming?" or "Where is focusing?" by providing clear explanations or insights that reflect the user's current interaction with the visualization, ensuring that the response is both relevant and informative.

4.5 Data Handling

Data handling ensures that raw data is brought into the system, organized, and ready for creating insightful visualizations. This process is important because it transforms messy, real-world datasets into structured formats that users can explore through interactive hierarchical diagrams. In image 4.15 we can see how datasets and mateadata are processed to create hierarchical structures. In this section, we will walk through how the system manages data, from loading it into memory to shaping it into forms suitable for both hierarchical and tabular displays.

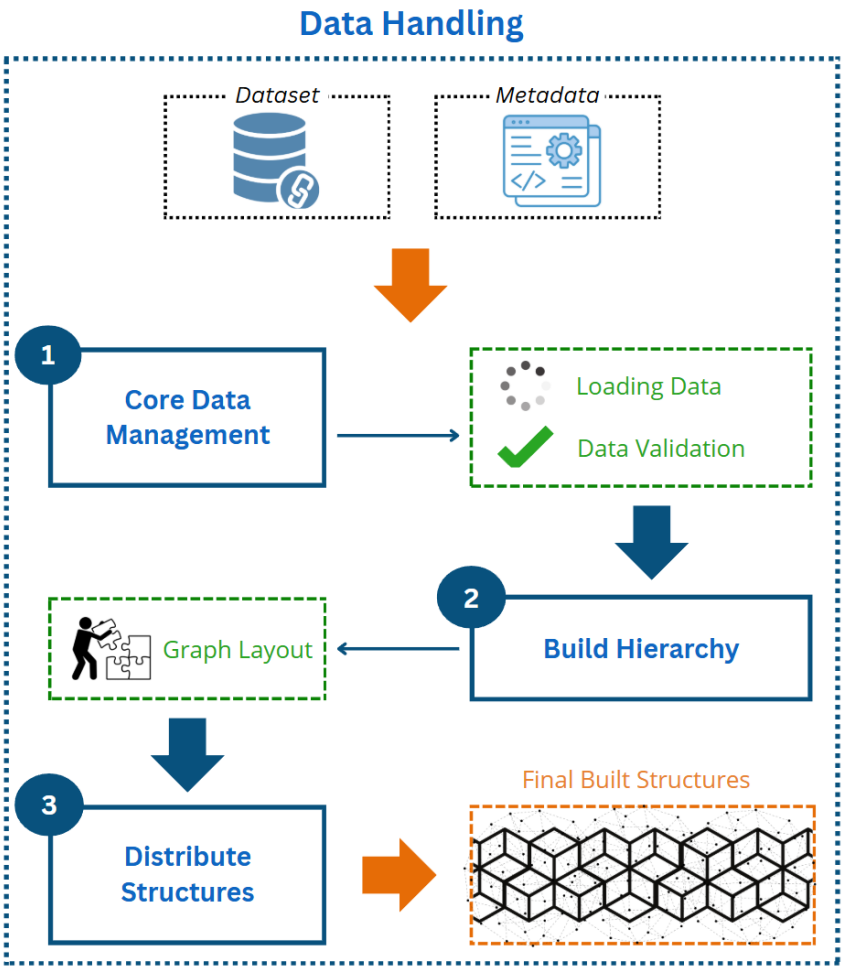


Figure 4.15: Image illustrating the data handling scheme, showing how datasets and metadata are processed to create hierarchical structures.

4.5.1 The Core of Data Management

At the center of this process is a key component responsible for overseeing how data is handled within the system. Think of it as a librarian who not only keeps track of all the books (data), but also organizes them so they're easy to find and use. This component consists of two main pieces: the dataset itself, often stored as a table (imagine a spreadsheet like a CSV file), and a separate guide, or metadata, that explains what the data means and how it is connected (typically in a structured format like JSON). When these two pieces arrive, the system springs into action.

First, it loads the dataset into a table-like structure that is efficient for sorting and searching. It checks that the table has all the columns that the metadata expects, making sure that nothing critical is missing. Next, it digs into the metadata to understand the dataset's story: What kinds of items (or entities) are in the data, what details (attributes) describe them, and how they relate to each other? For the detailed explanation of how does the metadata works, refeer to Section 5.6.1. For example, in a dataset of online comments, the metadata might tell us there are "comments" with details like "author" and "content," and that some comments reply to others, forming a hierarchy.

The system also calculates what type of information each column contains, for example, numbers, text, or categories, which helps later when deciding how to display the data visually. Then, it builds hierarchical structures (Section 5.6.2) based on the metadata, for example, a force-directed graph, which shows items as dots connected by lines, and another for a circle packing layout, where items nest inside each other like circles within circles. These structures are kept ready for other parts of the system, such as visualization tools or even a chatbot, to use whenever needed.

4.5.2 Metadata and Building the Structures

Metadata is like a map that guides the system through the dataset's landscape. It's usually a structured file that lays out the key details: what entities exist (like "comments"), how they're identified (say, by a "Comment ID"), what attributes they have (like "author" or "toxicity level"), and how they link up (for instance, one comment replying to another). It might also include extras, like which colors to use in visuals or what to show when you hover over an item.

The magic happens when the system uses this map to turn a flat table into a hierarchy ready for visualization. It starts with a "root" point—think of it as the top of a family tree—defined in the metadata. From there, it follows the relationships to build hierarchical structures. For example:

- **Force-Directed Graph Structure**: This is a web of dots and lines. Each dot (or node) represents an entity, like a comment, and each line (or link) shows a connection, such as a reply. The system begins at the root and traces the dataset to find related items, attaching their details (attributes) to each dot. For example, a comment node might carry its author's name and toxicity score, which could later color or size the

dot in the graph. Image 4.16 represents how metadata helps creating the directed force graph structure.

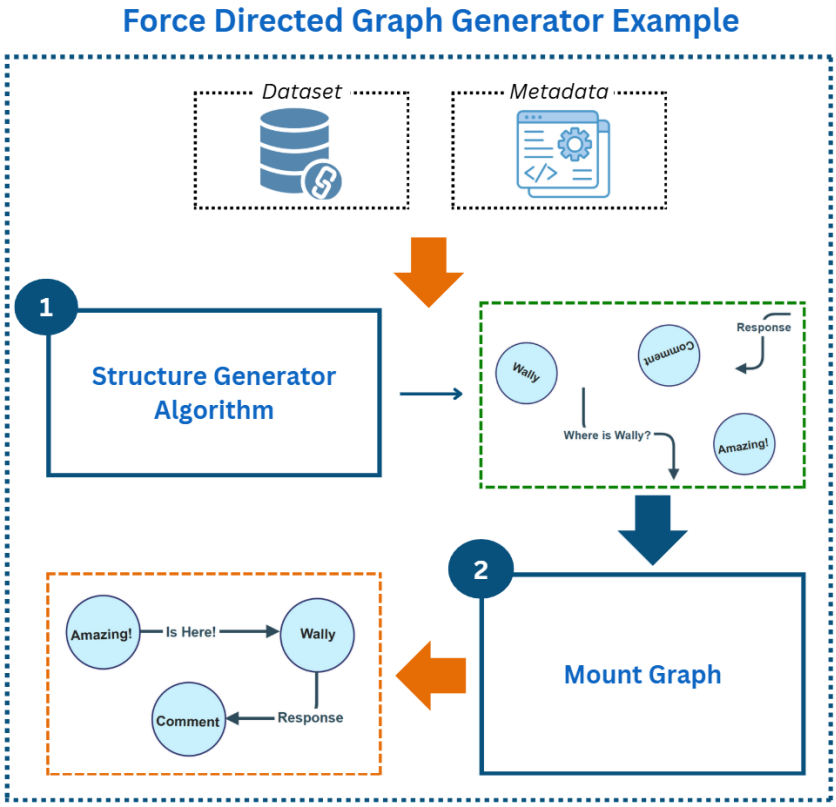


Figure 4.16: Image illustrating the metadata structure generation process, showing how metadata guides the creation of hierarchical structures.

Both processes are designed to handle imperfections gracefully. If some data is missing, like a comment without a reply, the system does not discard it. Instead, it might connect that item to the root or another part of the hierarchy, keeping as much information as possible. These structures are then stored, waiting for the visualization tools to bring them to life.

In essence, data handling is about preparing the raw material, like your dataset, into something that the system can work with creatively. Using metadata as a guide ensures that it is structured in a way that is versatile and meaningful, ready to reveal insights through interactive visuals, regardless of the dataset's quirks or complexity.

4.6 Retrieval-Augmented Generation (RAG) Module

In this project, the Retrieval-Augmented Generation (RAG) empowers the chatbot to deliver informed, contextually rich responses and enhance its visualization capabilities. RAG merges two essential processes: *retrieval*, which searches for relevant information in a collection of documents, and *generation*, which crafts answers or insights using this information. The RAG Module (as shown in Figure 4.17) facilitates this by systematically processing external documents to support chatbot responses and visualizations. This section walks through the RAG pipeline step by step, explaining how it is integrated in the system and its use.

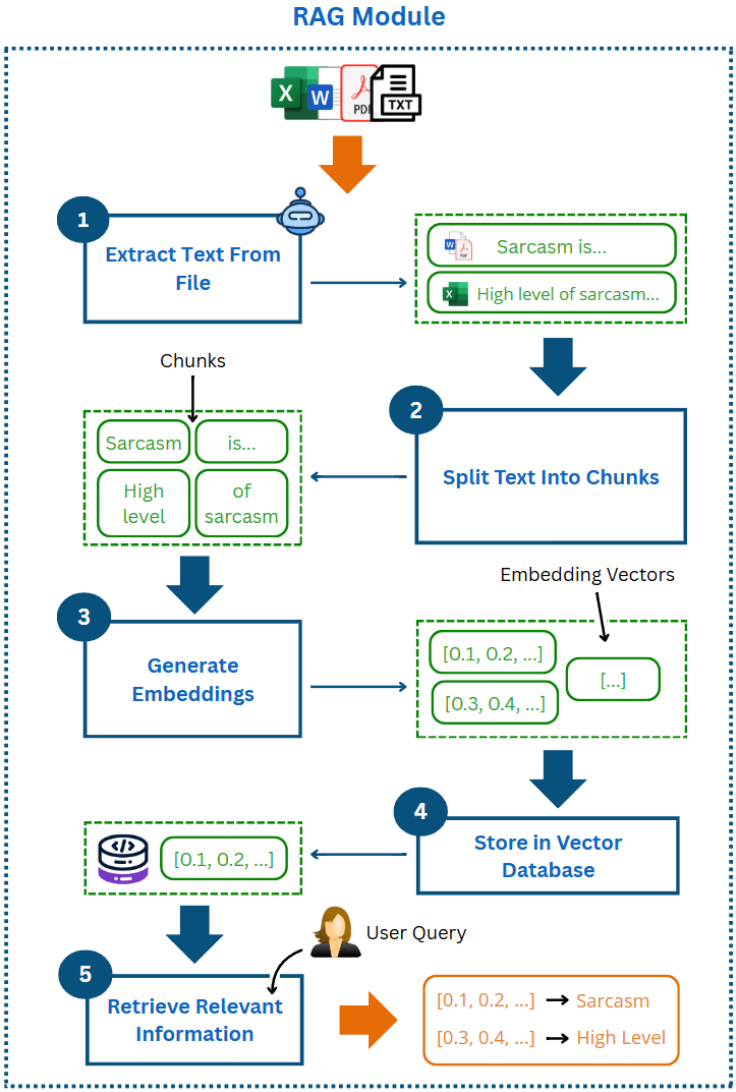


Figure 4.17: RAG Module Scheme.

The pipeline splits in five stages: first, it extracts text from files like PDFs or Word documents (Step 1); second, it divides the text into manageable chunks (Step 2); third, it generates embeddings, converting these chunks into high-dimensional vectors (Step 3); fourth, it stores these vectors in a vector database (Step 4); and fifth, it retrieves relevant information based on a user query by comparing the query's embedding to stored vectors, using similarity measures to identify matches like "Sarcasm" or "High Level" (Step 5). These steps, implemented in the `RAG System`, enable the chatbot to take advantage of external documents, enriching its understanding beyond the datasets handled in the **Data Handling** section or the queries processed by the **NLP Pipeline**. For example, when a user asks about toxicity in comments or requests a visualization of related data, the RAG Module retrieves precise details to inform both the chatbot's reply and the visual output generated in the **Visualization Generation** section.

4.6.1 Step 1: Document Processing and Text Extraction

The goal is to extract usable text, setting the stage for deeper analysis and serving as the first step in the RAG Module architecture (as depicted in Figure 4.18).

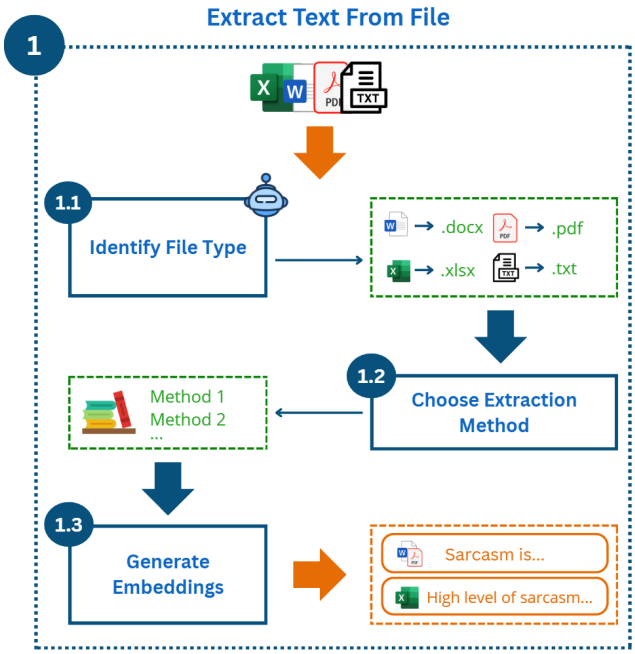


Figure 4.18: First step of the RAG module scheme.

The process starts with identifying the file type, using an automated check to recognize formats like .docx, .pdf, .xlsx, or .txt. Based on this, the system chooses the right extraction method for each file type, ensuring that the text is accurately extracted from the document. This text, which contains examples such as "Sarcasm is..." or "High level of sarcasm...," is

then prepared for the next stage of embedding generation, laying the foundation for the RAG Module's deeper analysis.

4.6.2 Step 2: Text Chunking

With the text extracted, the pipeline breaks it into smaller pieces, or *chunks*, to make it manageable for later steps, as illustrated in the **Split Text Into Chunks** scheme (Figure 4.19). Large texts overwhelm language models and embedding tools, so the process begins by initializing a text splitter (Step 2.1), which configures the splitting parameters, specifically a **chunk size of 500** characters and an **overlap of 50 characters** to preserve context across boundaries. This overlap ensures that a sentence split between chunks remains intact in at least one, avoiding loss of meaning. Next, the text is converted into a document format (Step 2.2), preparing it for the final splitting step (Step 2.3), where the **splitter divides** the text into chunks like "Sarcasm is..." and "High level of sarcasm...".

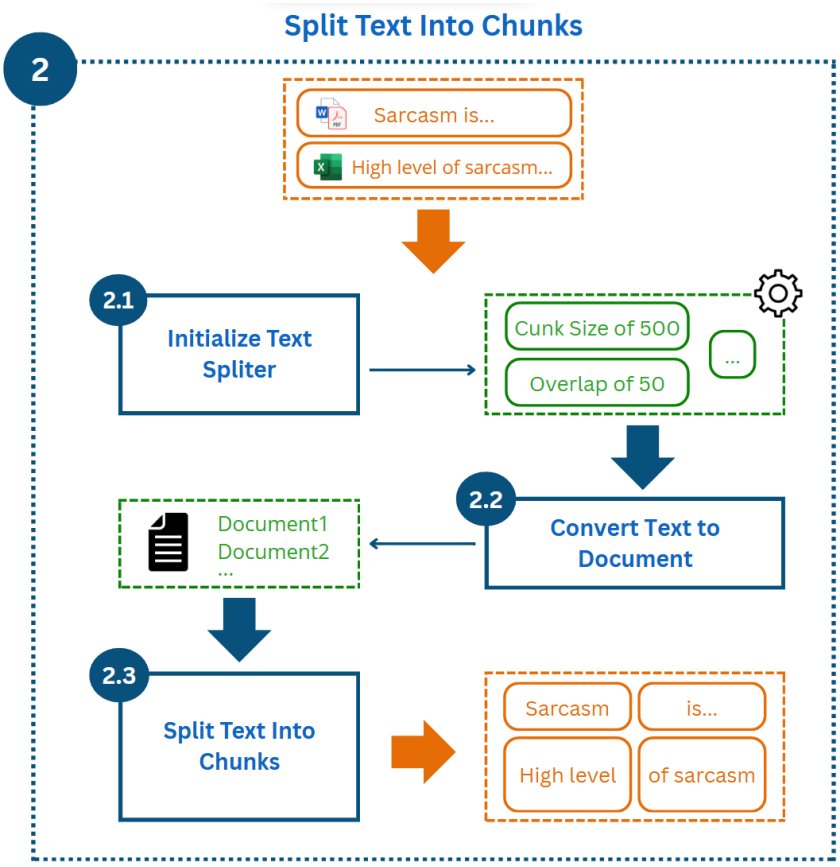


Figure 4.19: Second step of the RAG module scheme.

The result is a set of chunks, where each chunk is a manageable piece of the original text, ready for embedding generation in the next stage of the RAG Module. This step

ensures that the system can process large documents efficiently without losing critical context, setting the foundation for accurate retrieval and generation.

4.6.3 Step 3: Embedding Generation

Now, the pipeline transforms each chunk into a numerical form called an *embedded vector*, a concept central to understanding meaning in text, as shown in the Generate Embeddings scheme (Figure 4.20). Think of an embedded vector as a unique signature that captures the essence of the text: its topic, tone, or intent. The process starts by processing each chunk individually (Step 3.1), where the system uses an Embedding Model to analyze text like "Sarcasm is..." or "High level of sarcasm...". Then, it generates the embedding (Step 3.2), converting these **chunks** into vectors such as [0.1, 0.2, ...] or [0.3, 0.4, ...].

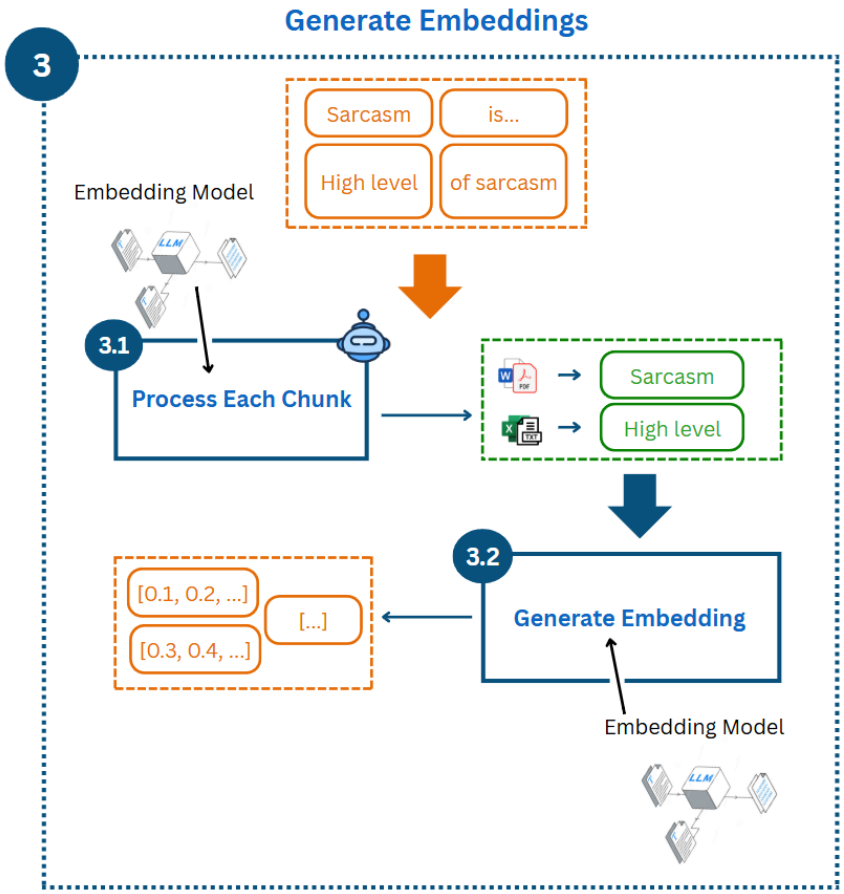


Figure 4.20: Third step of the RAG module scheme.

These coordinates act like a secret code, learned by a pre-trained model from vast amounts of text, ensuring that similar texts get similar signatures. This step connects to the **NLP Pipeline**, where understanding query intent also relies on interpreting the

meaning, although RAG applies it to document chunks rather than user inputs, ensuring that the system can later retrieve relevant information based on these meaningful vectors.

4.6.4 Step 4: Vector Storage

These embedded vectors need a home where they can be quickly accessed later, as shown in the **Store in Vector Database** scheme (Figure 4.21). The process begins by checking and initializing an **existing database**, or setting up a new one if needed (Step 4.1), using a database to store the vectors along their original **chunks** like [0.1, 0.2, ...] and [0.3, 0.4, ...]. Once initialized, the system persists the database, ensuring that the data is saved for future use. Then, it **indexes** the embeddings (Step 4.2), organizing them for efficient retrieval. The used database is optimized for high-dimensional vectors, using a smart method to build connections between vectors based on their positions, speeding up searches by avoiding the need to check every single vector.

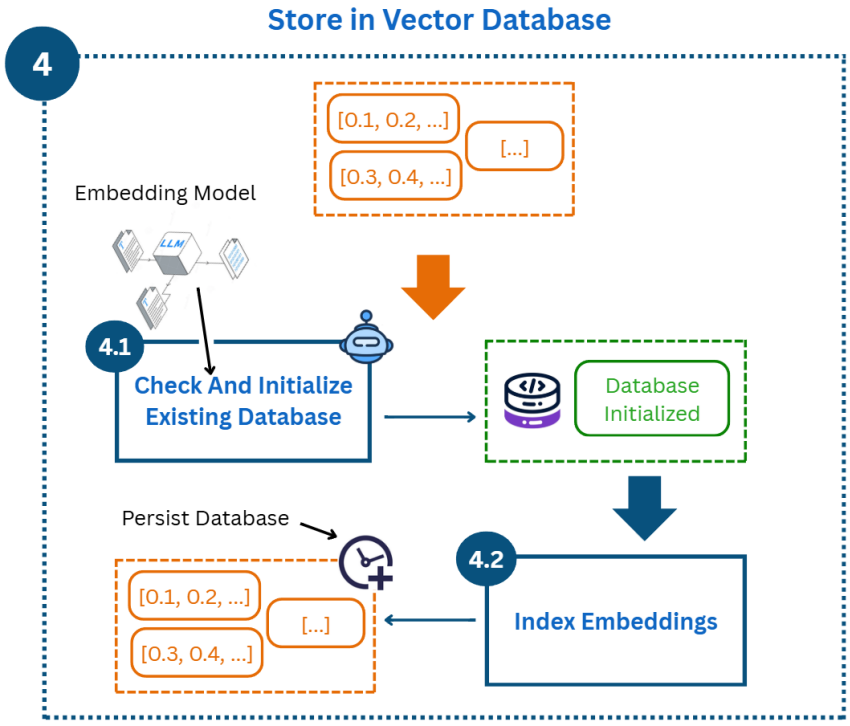


Figure 4.21: Fourth step of the RAG module scheme.

4.6.5 Step 5: Retrieval Mechanism

When a user submits a query via the **User Interaction and Chatbot** interface, the pipeline retrieves **the most relevant chunks** to inform the response, as illustrated in the **Retrieve Relevant Information** scheme (Figure 4.22). The process begins by embedding the user's

query (Step 5.1), where the Embedding Model converts the query into a vector, such as [0.5, 0.6, ...], ensuring it matches the format of stored vectors like [0.1, 0.2, ...] for "Sarcasm" or [0.3, 0.4, ...] for "High Level". Then, it performs a **similarity search** (Step 5.2), comparing the query vector with the database to identify the most closely aligned chunks, resulting in the top chunks that best fit the user's intent.

Closeness is determined by how well the vectors align in meaning, similar to how two arrows pointing in nearly the same direction would indicate they're related, regardless of their size. Using a **smart search method**, it quickly identifies the top 3 chunks with the strongest alignment, such as those labeled "Sarcasm" or "High Level". This response could also trigger a visualization, like a pie chart of factors, tying RAG to the **Visualization Generation** section, ensuring the chatbot delivers precise and contextually relevant information.

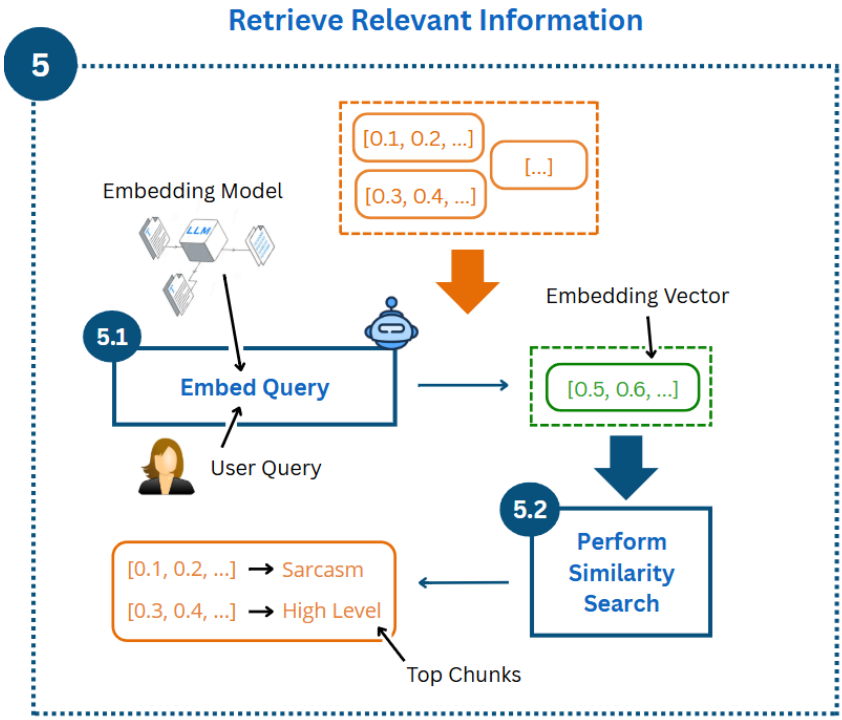


Figure 4.22: Fifth and last step of the RAG module scheme.

Chapter 5

Implementation

This chapter outlines the practical steps taken to develop the system, offering a detailed look at how the design concepts from the architecture section are put into action. The architecture section provided a blueprint of the system's structure and components, and here we expand on that by explaining the technologies, processes, and interactions used to build it.

5.1 System Overview Implementation

This section brings the architecture from the previous chapter into reality by integrating its components with specific technologies to create a working system. The architecture, illustrated in Figure 5.1, is the same as in the architecture chapter but now includes the technologies used for each part. It is split into back-end and front-end sections, and this section explains the main tools each component relies on and how they work together to process data and respond to user requests.

The back-end consists of the following components:

- **Data Handler**: Utilizes `Pandas` [14] to load and organize raw data from CSV files and JSON metadata, and `Chroma` [4] to store document information for efficient retrieval.
- **RAG Module**: Enhances responses by retrieving document context using `all-MiniLM-L6-v2` [9] for embeddings and `gpt-4o-mini`[13] to integrate query and context into informed answers.
- **NLP Parser**: Interprets natural language queries with `distilbert-base-uncased`[10], categorizing them to guide system actions.
- **Reasoning Module**: Generates explanations using `gpt-4o-mini`[13], leveraging inputs from the NLP Parser and RAG Module for clarity.

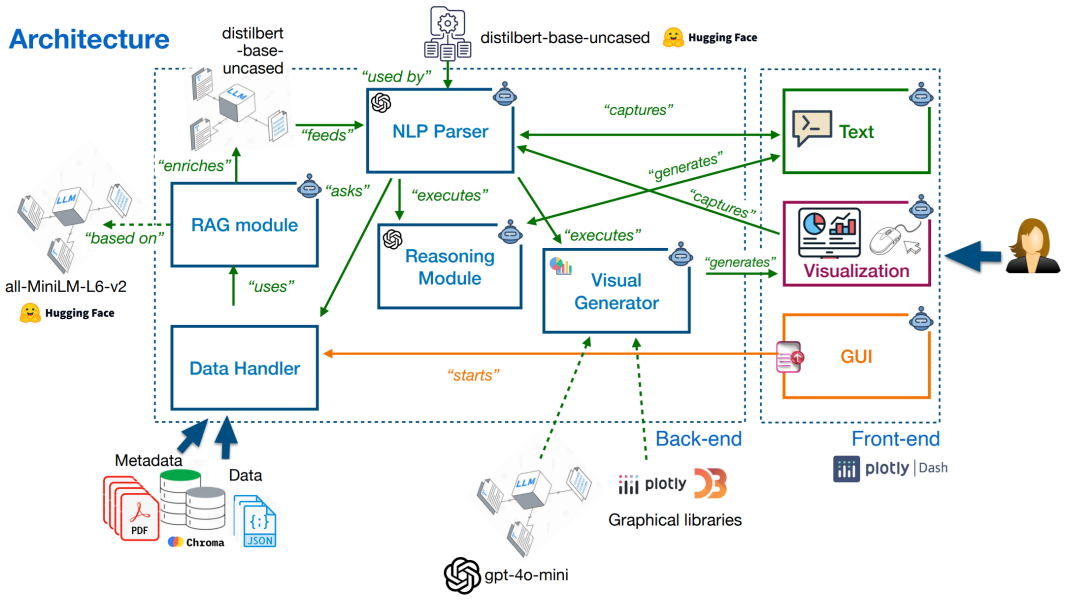


Figure 5.1: Diagram illustrating the full architecture with technologies.

- **Visual Generator**: Produces interactive visuals like charts with `Plotly` for tabular charts and `D3.js` for hierarchical graphs[16, 1], using `gpt-4o-mini`[13] to dynamically generate visualization code.

The front-end features a **GUI** built with `Plotly Dash`[16], enabling users to input queries and view results in a web-based interface. A **Controller** orchestrates the flow, routing commands from the NLP Parser to appropriate modules and displaying outputs via the GUI.

5.2 NLP Parser

The implementation of the NLP Parser transforms natural language queries into structured commands that can be interpreted and executed. This chapter explains how the parser was built, discussing the core components, the underlying technologies, and the integrated processing pipeline. Rather than treating each element in isolation, the following sections describe how these parts interact to ensure robust and accurate query interpretation.

Figure 5.2 shows the same pipeline as Figure 4.2 from the architecture explanation but with the technologies used.

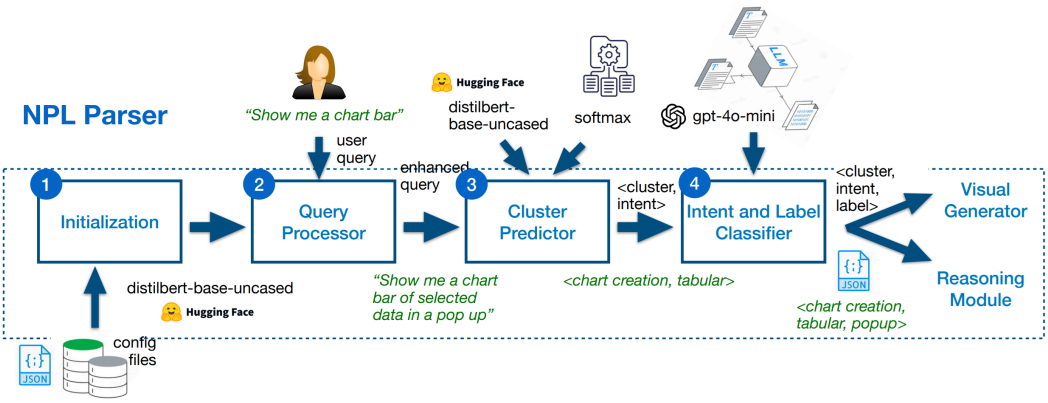


Figure 5.2: Diagram illustrating the query classification process with the technologies used.

The technologies used are the following:

- **SpaCy:** Used in the Query Processor stage for tokenization and normalization of the user input, enhancing consistency and facilitating downstream semantic processing[6].
- **Hugging Face Transformers:** The system leverages the `distilbert-base-uncased`[10] model from Hugging Face for semantic understanding. This lightweight BERT variant is used to encode user queries and assist in both the Cluster Predictor and the Intent and Label Classifier components.
- **Softmax Layer:** Applied after the transformer encoding to compute probability distributions over possible intent or cluster classes, enabling accurate classification.
- **OpenAI GPT (gpt-4o-mini[13]):** A lightweight version of OpenAI's GPT model is used in the Cluster Predictor to help generate or refine clusters and intent predictions with language generation capabilities.
- **JSON Configuration Files:** Used to initialize system parameters and store mappings or label definitions. These files guide the parsing and classification logic and are also used to store the final structured output (<cluster, intent, label>) passed to the Visual Generator and Reasoning Module.
- **Visual Generator & Reasoning Module:** These components receive the classified query representation and generate appropriate visual outputs or reasoning responses. They use the identified cluster, intent, and label to drive system behavior.

5.2.1 Cluster and Classification Transformer

This classification enables the transformation of natural language input into structured commands that the system can understand and execute. The model powering this trans-

former is `distilbert-base-uncased` [10], a lighter and faster variant of BERT that maintains strong semantic understanding while being more efficien, making it well-suited for real-time applications.

The training process begins with the collection of labeled query examples, including various synonyms and phrasings associated with specific intents. These examples are stored in a JSON file and serve as the initial training dataset. To improve the model's ability to generalize across different expressions, this dataset is augmented using paraphrasing techniques. Specifically, the T5 model [18] is used to generate multiple re-worded versions of each query, broadening the linguistic variety and making the model more robust to diverse user input.

Once the dataset is prepared, it is split into training and validation subsets. The transformer is then fine-tuned on this data using the Hugging Face Transformers library [**?**]. During fine-tuning, the model's weights are adjusted to minimize classification error on the training set, while performance on the validation set is monitored to guard against overfitting.

To keep the model up-to-date and responsive to real-world usage, a feedback mechanism is integrated into the system. When users correct or confirm the model's predictions, these interactions are logged in a feedback file. Once the number of entries exceeds a defined threshold (e.g., 10 entries), the model is retrained by incorporating this feedback into the training data. To emphasize its importance, each feedback entry is repeated several times during training. After retraining, the validated feedback is merged into the main synonyms JSON file, and the log is cleared to maintain system efficiency.

The decision to use `distilbert-base-uncased` stems from its effective balance between performance and resource usage. As a distilled version of BERT, it offers high-quality language understanding with significantly reduced computational cost, making it suitable for responsive applications. Data augmentation through paraphrasing helps overcome the limitations of small labeled datasets by introducing a broader range of linguistic variations, which improves the model's ability to handle unseen queries.

User feedback plays a key role in the system's ability to improve over time. By retraining the model on real user interactions, the system adapts to changing language patterns and ensures that its performance aligns with actual usage.

Finally, using structured JSON files for storing synonyms, intents, and clusters ensures that the system remains easy to update and scale. New intents or query variations can be added by modifying the configuration files, without the need to alter the underlying codebase.

5.2.2 Query Processing Pipeline

The system processes queries through a series of integrated steps: normalization, fuzzy matching, entity extraction, context enrichment, and query augmentation. These steps prepare the input for accurate classification and interpretation. The figure 5.3 shows a

clear scheme on how the tokenization and normalization works. In this subsection we will see how each step works and which technologies are involved.

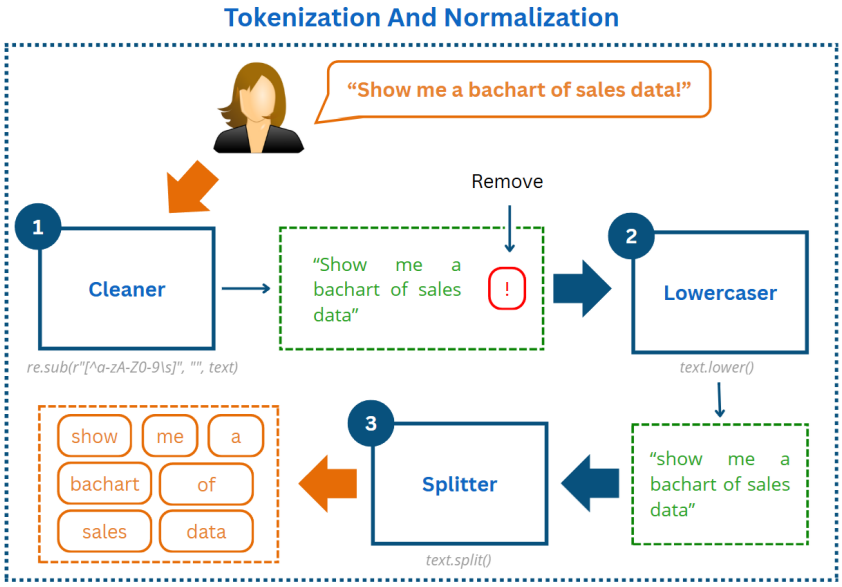


Figure 5.3: Tokenization and Normalization Implementation Scheme using SpaCy

Our system uses SpaCy[6] because it is a fast and robust open-source library for advanced Natural Language Processing (NLP) in Python, providing reliable tools for tokenization, part-of-speech tagging, and text normalization. We use it to tokenize and normalize the input query to ensure consistency and simplify downstream processing[6]. For example, the query "Show me a bar chart of sales data" is transformed into the token list ["show", "me", "a", "bar", "chart", "of", "sales", "data"]. Each token is converted to lowercase and stripped of excess whitespace. This preprocessing step is essential for improving the accuracy of intent recognition and ensuring that variations in user input do not affect the system's ability to interpret and respond appropriately.

Our system applies fuzzy matching (figure 5.4) using Levenshtein distance because it allows us to handle user input that may contain typos, misspellings, or slight variations in wording. This approach helps make the system more robust and user-friendly by not requiring exact matches between the input query and predefined keywords or phrases. The Levenshtein distance measures the minimum number of single-character edits (insertions, deletions, or substitutions) needed to transform one string into another. A similarity score is then computed based on this distance, enabling the system to still identify relevant terms even when the input is imperfect. It is calculated as:

$$\text{similarity} = 100 \times \left(1 - \frac{d(s_1, s_2)}{\max(\text{len}(s_1), \text{len}(s_2))}\right)$$

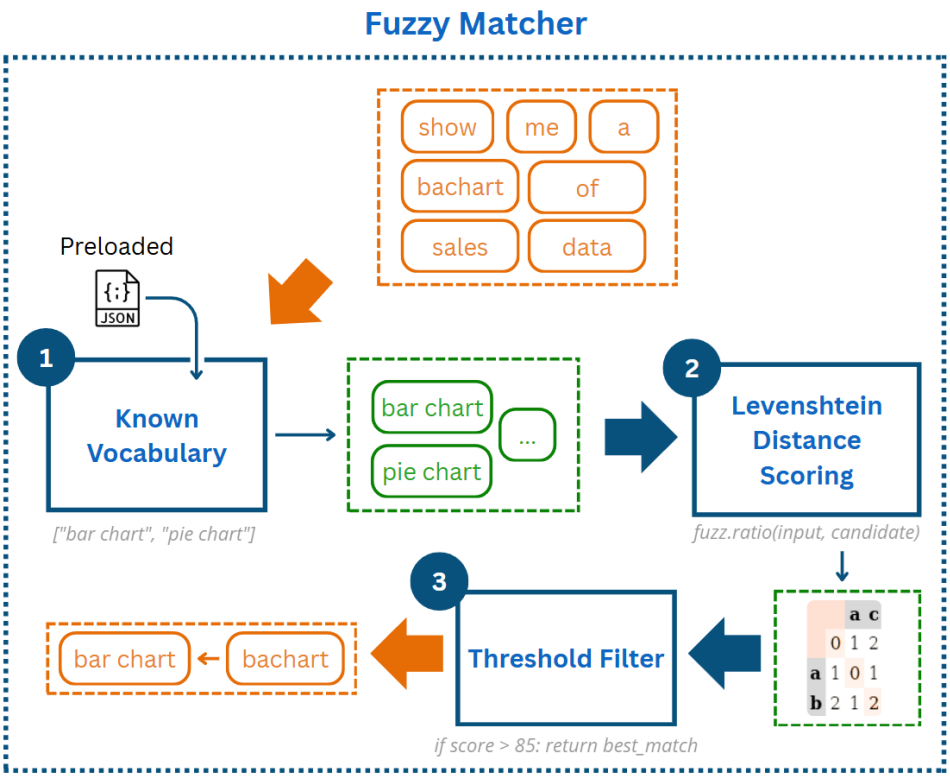


Figure 5.4: Fuzzy Matching Implementation Scheme

Tokens with scores above a threshold (e.g., 80%) are replaced with the most similar known term, such as mapping "barchart" to "bar chart".

Entity extraction (figure 5.5) follows, identifying references to dataset columns or other meaningful terms through exact or partial matching. This is important for recognizing domain-specific content like "sales" in the query.

These matched tokens are then enriched (figure 5.6) with contextual information. For example, if the query includes the word "popup", a flag is set to indicate that the chart should appear in a popup view. Additionally, conversation history may be consulted to disambiguate references like "the selected nodes".

The final result of this stage is an augmented version of the original query (figure 5.7), combining normalized tokens, matched entities, and contextual metadata. This comprehensive representation is ready for classification.

5.2.3 Cluster Prediction and Classification

With the augmented query prepared, the system employs DistilBERT[10], a lightweight version of BERT designed to retain most of its performance while reducing computational

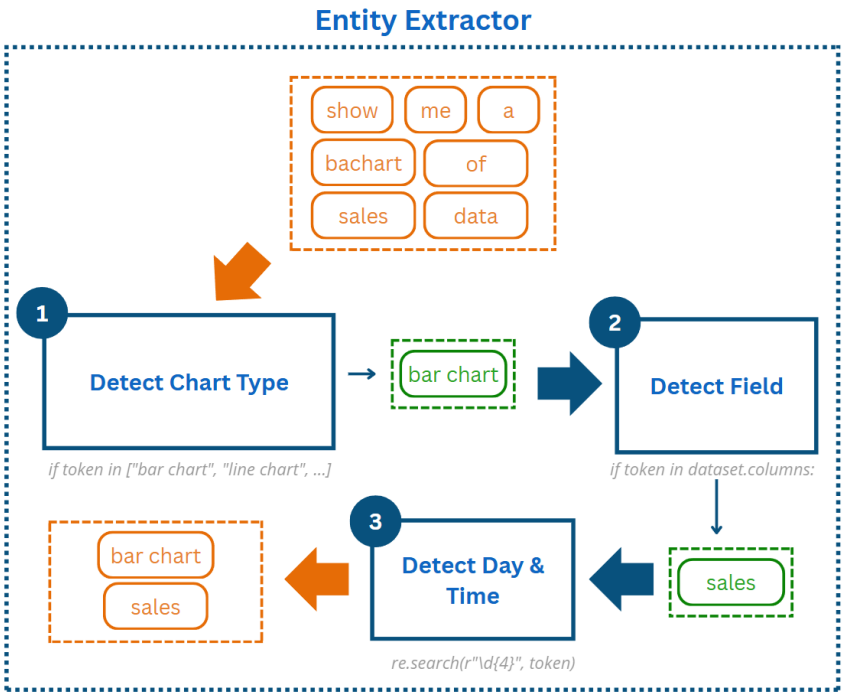


Figure 5.5: Entity Extraction Implementation Scheme

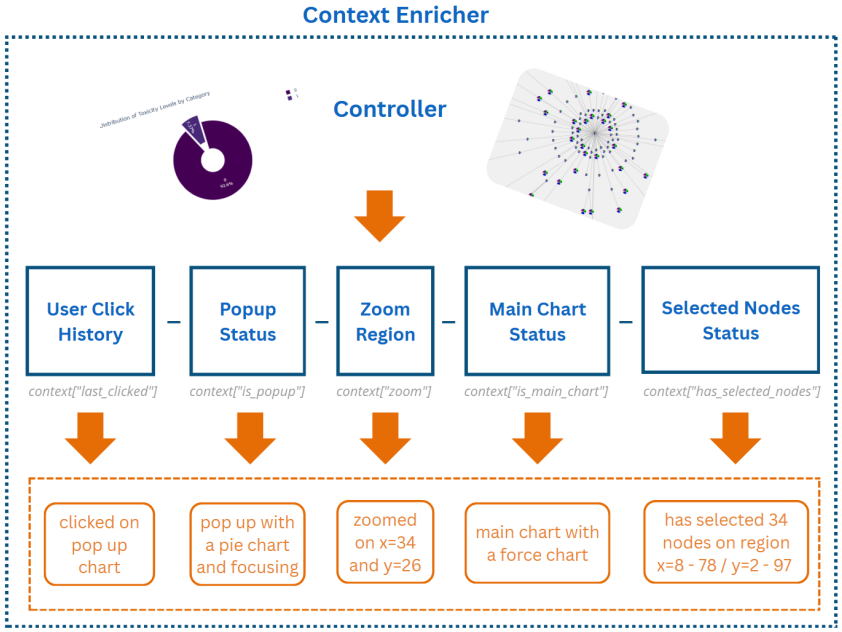


Figure 5.6: Context Enricher Implementation Scheme

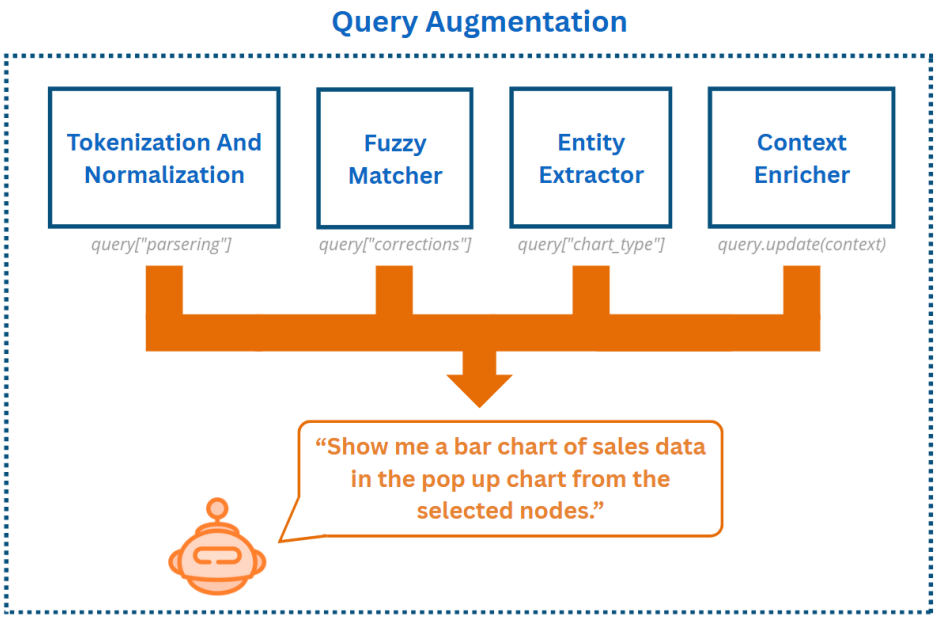


Figure 5.7: Query Augmentation Implementation Scheme

cost. It is particularly well-suited for real-time inference tasks like ours, where efficient classification of user intents is essential. The query is first tokenized (figure 5.8) using the DistilBERT tokenizer[10], which inserts special tokens such as [CLS]—used to aggregate the overall meaning of the sentence for classification—and [SEP], which separates different parts of the input when needed.

This structured input is then passed through the model to generate raw output scores (logits), which are subsequently transformed into class probabilities using the softmax function:

$$p_i = \frac{e^{x_i}}{\sum_j e^{x_j}}$$

The cluster with the highest probability is selected. For example, if the result is [0.85, 0.10, 0.05] for clusters ["ChartCreation > Tabular", "ChartEditing > MainChart", "DataInteraction > Explanation"], the system selects "ChartCreation > Tabular".

5.2.4 Intent and Label Resolution via LLM

After predicting the cluster, the system leverages **GPT-4o-mini**[13] to enhance the interpretation of the query with greater contextual awareness (figure 5.9). A carefully crafted prompt is provided to the model, containing detailed instructions designed to help it reason effectively about the predicted cluster and the semantics behind each possible intent.

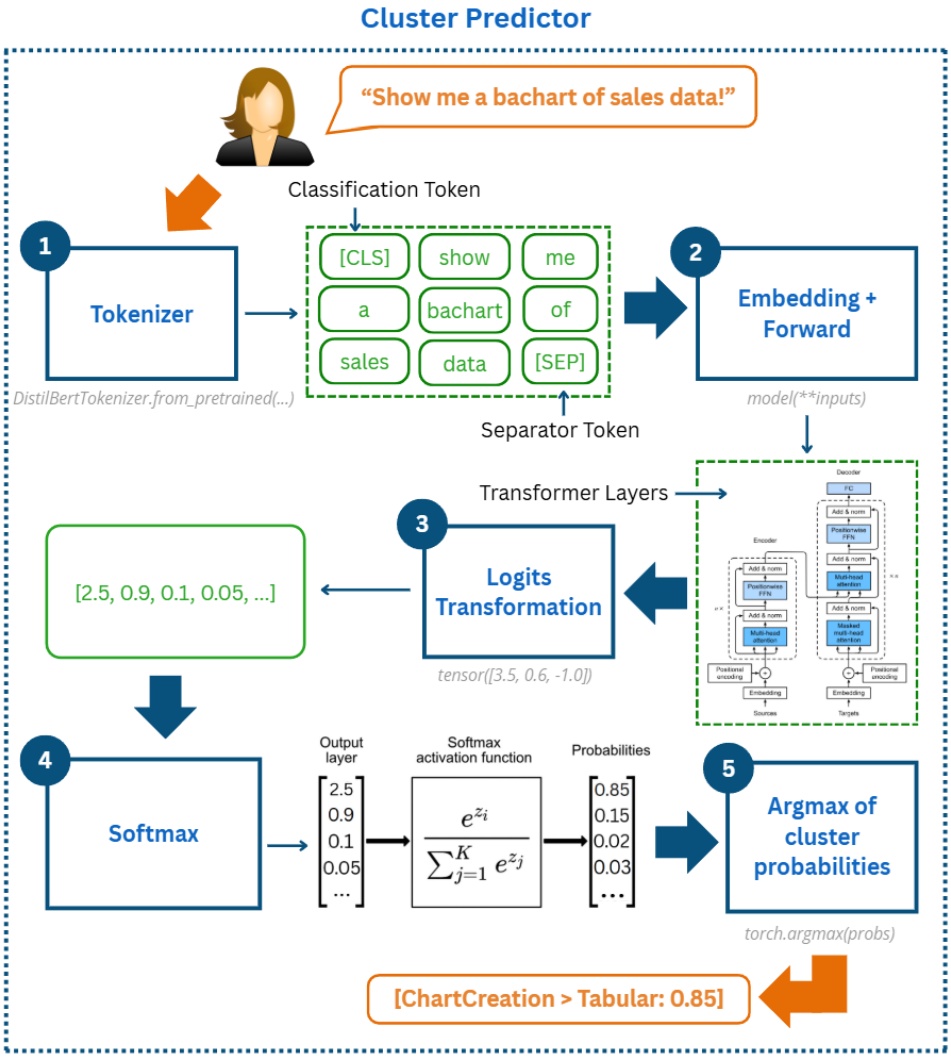


Figure 5.8: Cluster Prediction Implementation Scheme

For example, a cluster such as *"ChartCreation > Tabular"* is mapped to a predefined list of candidate intents like `CreateLineChart`, `CreateBarChart`, or `CreatePieChart`.

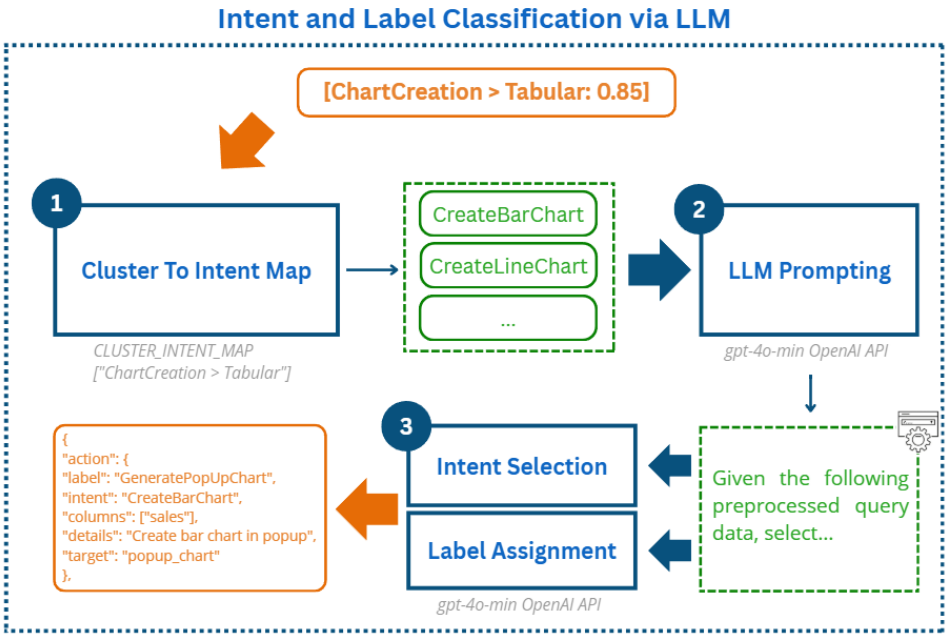


Figure 5.9: Intent and Label Classification Implementation Scheme

The model is prompted with the original user query, the identified cluster, and these candidate intents. Using the full prompt and its internal reasoning capabilities, the model selects the most appropriate intent based on the context. For example, a query like *"Show me a bar chart of sales data in the popup chart"* would lead the model to choose the intent `CreateBarChart`.

Once the intent is selected, the system assigns a corresponding label that defines the complete action logic. If the query context implies that the result should appear in a popup, the final label might be `GeneratePopUpChart`, encapsulating both the type of visualization and its display behavior.

If a more detailed version of the output of the JSON returned by the GPT-4o-mini model, please refer to Appendix A.

5.3 Controller Implementation

The Controller serves as the central hub for managing and directing user requests, ensuring that the system processes them efficiently and routes them to the appropriate modules. It interprets incoming JSON-formatted actions, such as those labeled `"GeneratePopUpChart"` or `"CreateBarChart"`, by first performing a *Check Actions* step (Step 1), where it evaluates the intent behind the user's request. Based on this evaluation, it routes the action to one of

two pathways: either to the Visual Generator (Path A) for actions involving the creation or editing of visualizations (e.g. `"GeneratePopUpChart"`, `"CreateCirclePacking"`), or to the Reasoning Module (Path B) for deeper analyses and explanations (e.g. `"ExplainFromSelection"`, `"ShowNodeDetails"`). Figure 5.10 illustrates this decision flow.

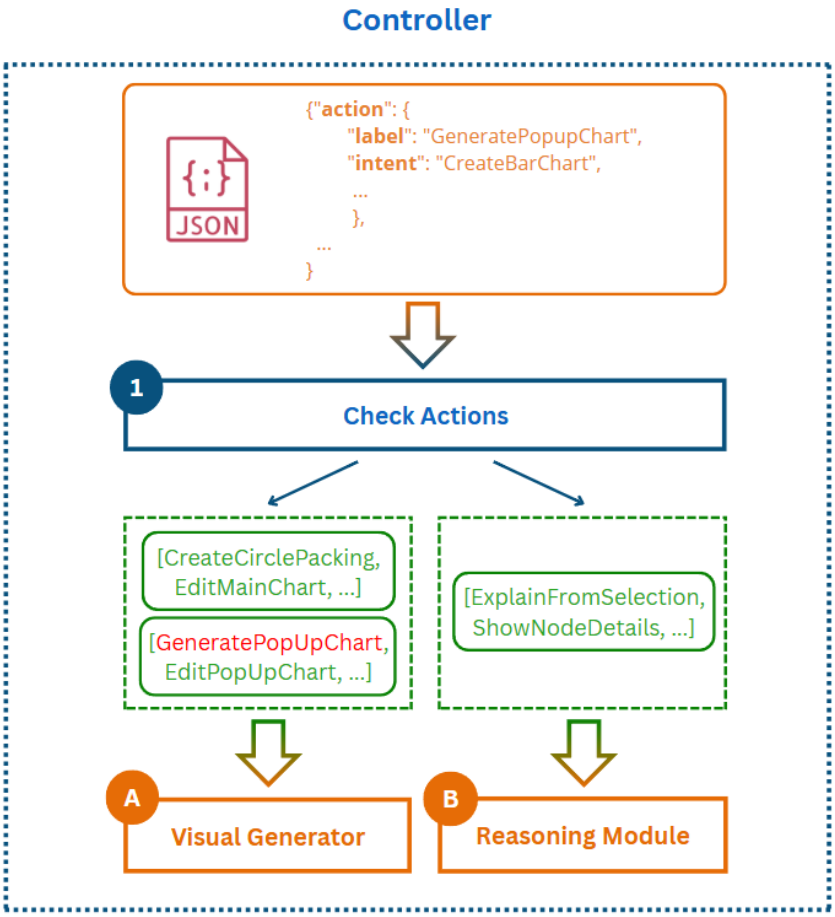


Figure 5.10: Diagram illustrating how each JSON action label is routed to either the Visual Generator (Path A) or the Reasoning Module (Path B).

5.3.1 Key Components and Technologies

The Controller uses the following components and libraries:

- **QueryClassifier**: A custom class for classifying user queries into action labels using natural language processing (NLP) techniques.
- **Data Handler**: Manages the dataset, providing access to columns, types, and metadata.

- **Visualizer**: Generates interactive visualizations using Plotly and D3.js.
- **Action Executor**: Executes specific actions, such as chart creation or editing.
- **Pandas**: Facilitates data manipulation and analysis.
- **Plotly**: Renders interactive visualizations.
- **JSON**: Manages configuration and data exchange.
- **Dotenv**: Handles environment variables, such as API keys.

These components are integrated within the `Chatbot` class, which maintains state through attributes like `current_context`, `conversation_history`, and `current_figure_type`.

5.3.2 Context Management

The Controller maintains a context dictionary, `current_context`, updated by the `update_context` method. This tracks states such as:

- `interacting_with`: The focus element (e.g., `"main_chart"`).
- `has_popup_chart`: A boolean indicating popup chart presence.

Context updates are time-sensitive, using a threshold (e.g. 30 seconds) to determine focus validity:

$$\text{if } (t_{\text{now}} - t_{\text{last}}) \leq 30, \text{ then update focus}$$

This ensures queries are interpreted in the correct context, enhancing action accuracy.

5.3.3 Configuration Updates

The `handle_main_plot_edit` method updates the visualization configuration of hierarchical graphs by generating targeted changes through a carefully crafted prompt sent to the OpenAI API (using the `gpt-4o-mini` model[13]). This prompt-based approach was chosen to enable natural language-driven updates that allow users to modify the visualization interactively and intuitively, without needing to manually edit configuration files or code.

When a user submits a query q (for example, "Make nodes bigger") along with a configuration section S (such as node radius settings), the method constructs a prompt combining these inputs and sends it to the language model. The model then returns an updated configuration section S' that reflects the user's request. This design decision, to send only the relevant configuration subset along with the user query, minimizes unnecessary data transmission and focuses the model on precise, context-aware edits.

To integrate the update, if the configuration section is dictionary-based, the returned S' is merged into the existing section S, overwriting any conflicting keys:

$$S \leftarrow S \cup S' \quad \text{(with } S' \text{ overwriting keys in } S\text{)}$$

For simpler, non-dictionary sections, S is fully replaced by S'. This merging strategy was chosen to preserve existing settings while efficiently applying only the requested modifications.

For example, given:

$$S = \{\texttt{"nodeRadius": \{"default": 10\}}\}, \quad q = \texttt{"Make nodes bigger"}$$

the API might return:

$$S' = \{\texttt{"nodeRadius": \{"default": 12\},"message": "Increased node size."}\}$$

which is merged into S, updating the node radius and triggering the visualization to reflect the change immediately.

Overall, this prompt-driven update mechanism balances flexibility and user-friendliness, enabling efficient, real-time configuration changes using natural language input while maintaining control over the exact parameters modified. It simplifies interaction and reduces friction in customizing complex visualizations.

5.4 Visualization Generator Implementation

The Visualization Generator transforms processed data and user commands into interactive visual representations, categorized into hierarchical and tabular types. Hierarchical visualizations, such as circle packing or force-directed graphs, are designed to depict nested or interconnected data structures, while tabular visualizations, including bar, line, or pie charts, present data in a simplified, often aggregated format. This section details the implementation of these visualization types, focusing on the underlying code and the technologies used.

5.4.1 Hierarchical Visualizations

Hierarchical visualizations leverage **D3.js**[1], a JavaScript library renowned for creating dynamic, interactive data visualizations in web browsers using SVG, HTML, and CSS. D3.js excels at handling hierarchical data due to its support for layouts like tree, pack, and force-directed graphs, making it ideal for this task.

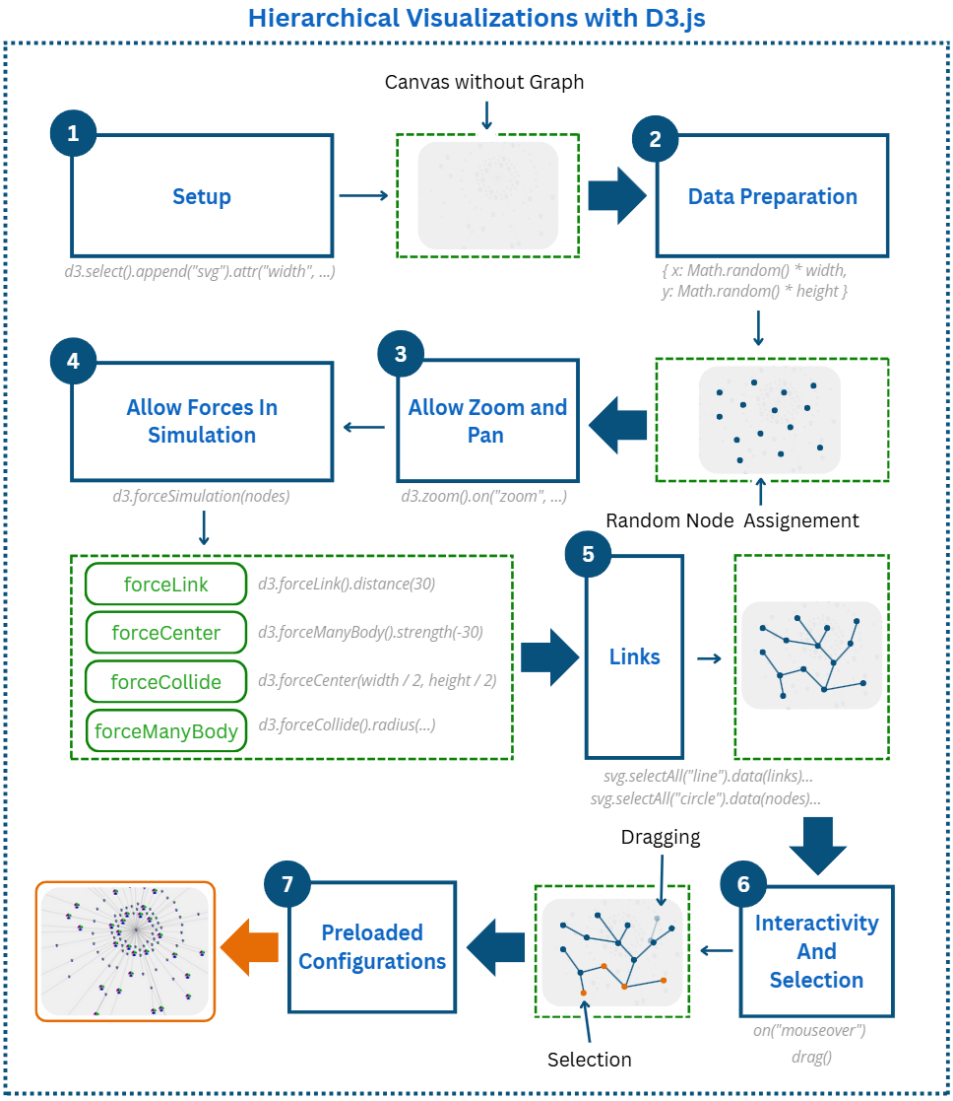


Figure 5.11: Hierarchical Tool Implementation Scheme

The implementation centers on the `HierarchicalPlotExecutor` class, which orchestrates the generation of hierarchical visualizations. Key methods of hierarchical graphs can be:

- `generate_force_graph`: Produces force-directed graphs.
- `generate_circle_packing`: Creates circle packing layouts.

All of them have almost the same structure, so for the explanation, we will put as an example the `generate_force_graph` method as it is the most similar of the other supported

structures. It processes a graph structure with nodes and links, generating JavaScript code that utilizes D3.js to render an interactive force-directed graph.

The process involves (figure 5.11):

1. **Setup**: Initializes an SVG element with dimensions specified by `width` and `height`.
2. **Data Preparation**: Assigns initial random positions to nodes within the SVG bounds if not predefined.
3. **Zoom and Pan**: Implements zoom and pan functionality via D3's `d3.zoom()` behavior, enabling users to navigate the graph.
4. **Simulation**: Employs a force simulation using D3's `d3.forceSimulation`, incorporating forces such as:
   - `forceLink`: Maintains link distances, modeled as springs with a distance parameter (e.g., 30).
   - `forceManyBody`: Applies electrostatic repulsion with a negative strength (e.g., -30).
   - `forceCenter`: Centers the graph at the SVG midpoint.
   - `forceCollide`: Prevents node overlap with a radius based on node size plus a factor (e.g., 5).

   The simulation solves a system of differential equations approximating a physical system where nodes reach an equilibrium state minimizing energy, as described by:

$$F_i = \sum_{j \neq i} (k \cdot d_{ij} - l) \cdot \hat{r}_{ij} - q \cdot \sum_{j \neq i} \frac{1}{d_{ij}^2} \cdot \hat{r}_{ij}$$

   where F_i is the net force on node i, k is the spring constant, d_{ij} is the distance between nodes i and j, l is the ideal link length, q is the charge strength, and $\hat{r}_{ij}$ is the unit vector from i to j.
5. **Links**: Renders SVG lines with attributes (color, thickness) derived from data, fading in via transitions.
6. **Nodes**: Creates circles or shapes with properties (color, size, opacity) based on node attributes, optionally using pie charts for multi-attribute visualization.
7. **Interactivity**: Adds event listeners for:
   - Mouseover: Displays tooltips with node details.
   - Click: Selects nodes, highlighting them (e.g., in orange).
   - Drag: Allows node repositioning, adjusting the simulation.
   - Shift + Drag: Enables area selection, highlighting multiple nodes.
8. **Selection**: Highlights nodes based on a condition (e.g., attribute-based filtering), updating visuals dynamically.

The circle packing and tree diagram implementations follow similar principles, using D3's[1] `d3.pack()` and `d3.tree()` layouts, respectively.

5.4.2 Tabular Visualizations

Tabular visualizations use **Plotly**[16], a powerful graphing library chosen for its ability to create interactive, high-quality charts across Python, JavaScript, and R. Plotly was selected because it supports a wide variety of chart types—from simple bar and line charts to more complex scatter plots and heatmaps—making it highly adaptable to different tabular data needs. Additionally, its built-in interactivity features, such as zooming, hover tooltips, filtering, and dynamic updates, significantly enhance user engagement and data understanding, allowing users to explore their data more intuitively compared to static charts.

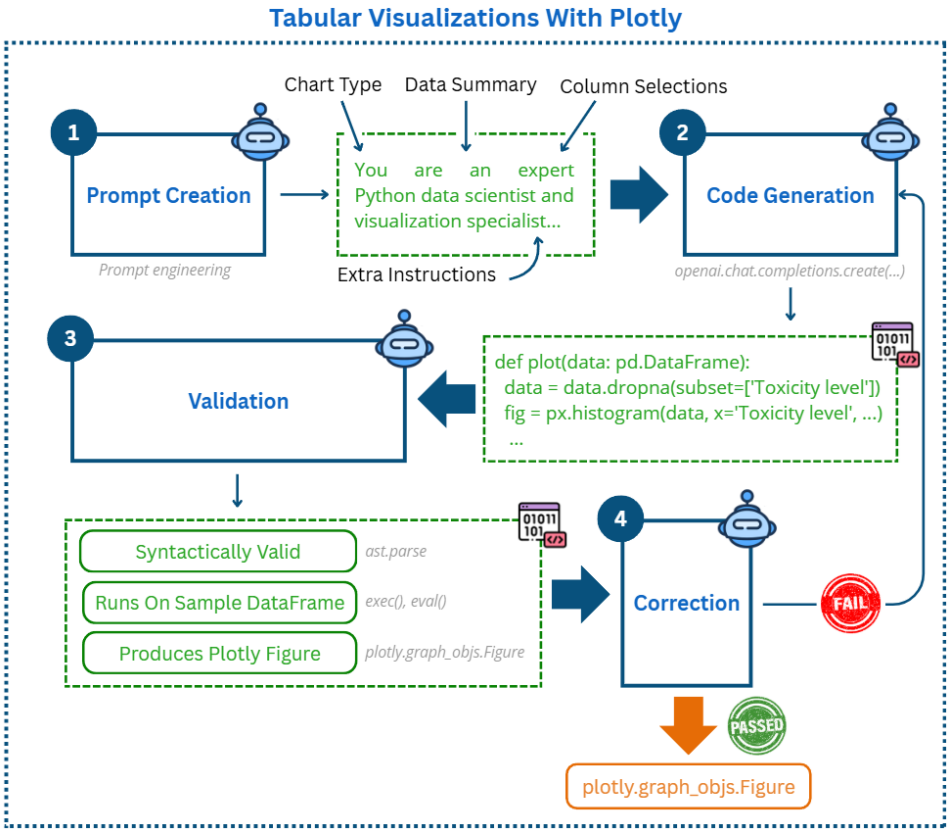


Figure 5.12: Tabular Tool Implementation Scheme

The `TabularPlotExecutor` class manages these visualizations by leveraging Plotly's flexibility to produce rich, dynamic charts (figure 5.12). A key innovation of this system is its integration with the **OpenAI API**[13], which dynamically generates Python plotting code based on natural language user inputs. This approach was chosen to maximize us-

ability and adaptability: rather than requiring users to write code or manually configure charts, the system translates simple queries into precise visualization instructions. This real-time code generation enables users to customize and interact with their visualizations instantly, making the data exploration process smoother and accessible even to those without programming experience.

The generation process is as follows:

1. **Prompt Creation and Code Generation**: The system constructs a detailed prompt incorporating chart type, selected columns, dataframe summary, and chart-specific instructions (e.g., aggregation for bar charts, trend lines for line charts). This prompt is sent to the OpenAI API to `generate_plot_code`, which returns Python code using Plotly (e.g., `plotly.express`)[13] to create the desired chart.
2. **Validation**: The `validate_code` function ensures the generated code is correct by checking syntax with `ast.parse`, verifying essential structure (such as the presence of a `plot` function), and executing the code on a sample dataframe to confirm it produces a valid Plotly figure.
3. **Correction**: If validation fails, the `correct_code` function attempts to fix errors by making further API calls that incorporate error feedback to generate corrected code.
4. **Fallback**: Should correction fail after multiple attempts, the system falls back to a default chart (e.g., a histogram of a numeric column) to guarantee a valid visualization.

The interactivity of our system extends beyond the basic chart features to deliver a richer user experience. Users can hover over data points to view precise values, facilitating a deeper understanding of the information without overcrowding the visualization. Clicking on legend items enables dynamic filtering of the displayed data, allowing users to effortlessly focus on specific subsets. For more complex analyses involving multiple columns, the system automatically generates subplots using Plotly's `make_subplots` function whenever the user query suggests comparisons or multidimensional views. This layered interactivity empowers users to explore their data from different perspectives, making insights more accessible and the overall experience highly engaging.

To enhance the quality and relevance of the generated plotting code, the system uses the Retrieval-Augmented Generation (RAG) approach, which can be seen in Section 5.7. Before generating the code, the user's natural language query is employed to retrieve relevant information from uploaded documents or existing knowledge through the RAG pipeline. This retrieved context provides the model with up-to-date, domain-specific, or dataset-related insights that may not be explicitly included in the prompt or dataset.

By integrating this additional context, the OpenAI API can produce more accurate and customized Python code for the plot, addressing not only the user's immediate request but also the broader relevant information. This ensures that the generated code aligns better with the data structure, visualization standards, and specific analytical goals, resulting in visualizations that are more meaningful and effective.

5.5 Reasoning Module

The Reasoning Module serves as the core intelligence layer of the system, enabling it to interpret user intent, generate visualization goals, and provide coherent, context-aware explanations. At its core, the module interacts with the OpenAI API—specifically the `gpt-4o-mini` model—to process user queries in natural language and respond with meaningful insights. It operates on selected portions of data and metadata, combining them with carefully designed prompts to generate output that is not only linguistically coherent but also semantically aligned with the structure and content of the data.

5.5.1 Explanation and Reasoning

To provide explanations and reasoning (figure 5.13), the system utilizes the OpenAI API's GPT model (gpt-4o-mini), a powerful tool for natural language generation. Given a user query q and selected data D, the system constructs a prompt $P(q, D)$ that integrates both elements. This prompt is then processed by the GPT model, which generates a response by sampling the most probable sequence of words conditioned on the input.

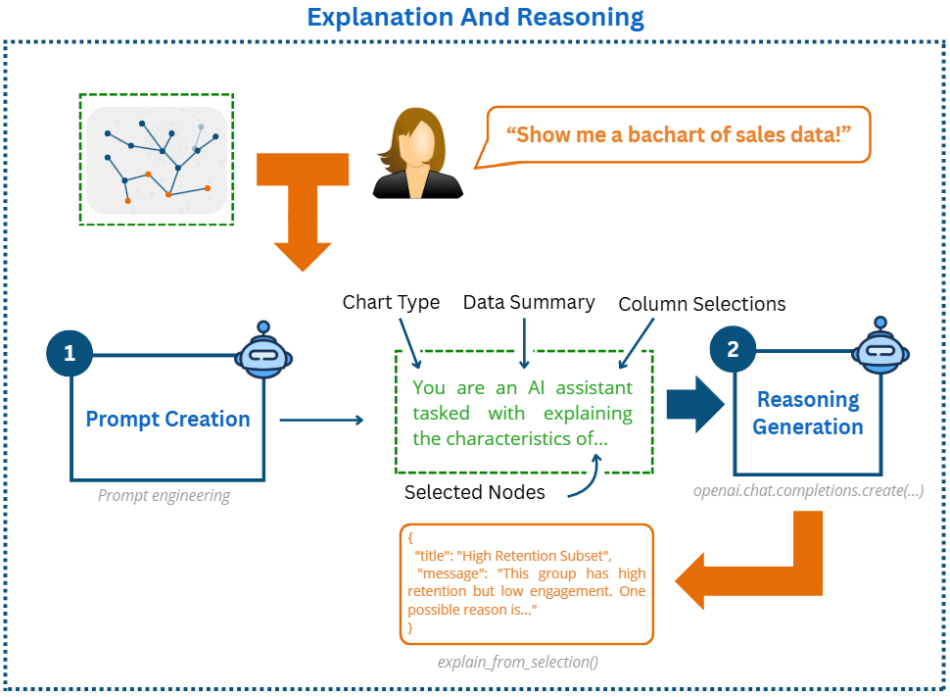


Figure 5.13: Explanation and Reasoning Implementation Scheme

The generation process can be expressed as:

$$P(w_t \mid w_1, \ldots, w_{t-1}, P(q, D))$$

where w_t represents each subsequent word in the response, predicted based on the preceding words $w_1, \ldots, w_{t-1}$ and the prompt $P(q, D)$. The output is an explanation, formatted as a JSON object with a title and a message, produced through methods such as `explain_from_selection`. This structured format ensures seamless integration into the user interface, delivering clear and contextually relevant insights to the user.

5.5.2 Goal Generation

While **LIDA** [15], a widely recognized Microsoft library, offers robust capabilities for goal generation, it was not employed in this system, but we mention it because we studied its implementation due to its potential. The primary reason is that LiDA does not natively support hierarchical goal generation within the specific structural framework required by our system. Hierarchical goal generation is essential here because our visualizations involve nested or multilevel data relationships (e.g., parent-child structures in `circle_packing` or `force_graph` layouts), which demand a method that can explicitly handle and represent such complexity (figure 5.14).

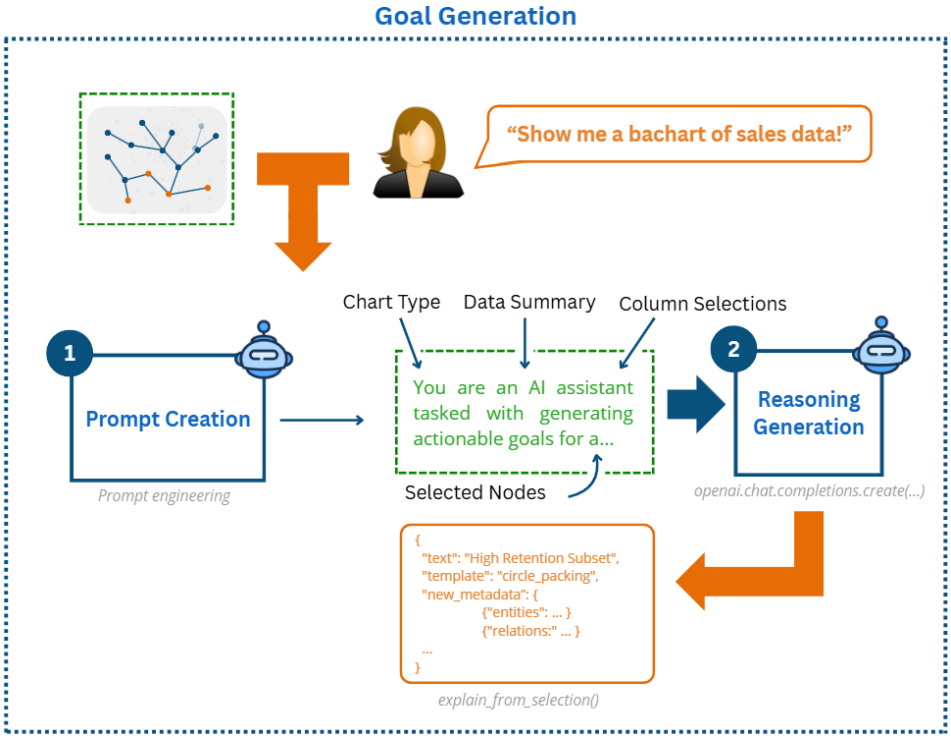


Figure 5.14: Goal Implementation Scheme

This limitation led us to develop custom goal generation methods that were tailored to our needs. Our `_generate_goals` method produces n actionable visualization suggestions, each expressed as a JSON object, by leveraging the OpenAI API (`gpt-4o-mini`). It uses a sample of the dataset (first 10 rows), metadata, available columns, and their types

to propose hierarchical visualization goals that align with our system's structure and interactivity requirements.

Formally, the process can be seen as a mapping:

$$G : D \to \{g_1, g_2, \ldots, g_n\}, \quad g_i = \{\text{text}, \text{template}, \text{new_metadata}, [\text{columns}, \text{condition}]\}$$

where D represents the dataset context, and each goal g_i consists of:

- `text`: A descriptive suggestion (e.g., "Visualize comment threads").
- `template`: Visualization type, either `circle_packing` or `force_graph`.
- `new_metadata`: Definitions of entities (e.g., {"Comment": {"id": "Comment ID", "attributes": {"author": "User ID"}}}) and relations (e.g., {"name": "replies_to", "source_entity": "Comment ID", "target_entity": "Thread"}).
- Optional `columns` and `condition` for filtering data subsets (e.g., "d.attributes['Toxicity level'] > 0").

A prompt including these details is sent to the API to generate n goals. The responses are then validated to ensure structural correctness and appropriate column references. If any generated goal is invalid, it is replaced with a predefined fallback goal:

$$g_{\text{fallback}} = \{\text{"Visualize hierarchy"}, \text{"force_graph"}, \text{default metadata}\}$$

5.6 Data Handling

The data handling component is responsible for taking raw datasets—often in CSV format—and transforming them into structured, hierarchical representations that can be visualized as force-directed graphs or circle-packing diagrams. In this chapter, we will:

- Describe how raw CSV data is ingested and how metadata (in JSON) drives its interpretation.
- Explain how the JSON metadata is defined, parsed, and used to guide the creation of a general hierarchical structure.
- Detail the overall algorithmic approach (in Python) for constructing a generic hierarchical graph from flat data.
- Show how the `DataHandler` class ties everything together, allowing for dynamic updates when either the data or the metadata changes.

By the end of this chapter, you should have a clear, step-by-step guide to reproducing and extending this data-handling pipeline for any dataset that can be organized hierarchically.

5.6.1 Data Input and Metadata Management

Before any hierarchy can be generated, we must load two things:

1. **Raw Data (CSV)**: The actual dataset—e.g., a table of comments, each with an ID, author, content, and various feature columns.
2. **Metadata (JSON)**: A blueprint that explains how the flat table should be interpreted as a hierarchy—what the root node is, what each entity's fields are, and how entities link to one another.

In practice, Python's `pandas`[14] library reads CSV files into a `DataFrame`, which provides fast filtering, iteration, and missing-value handling. The metadata is stored in JSON format and is loaded using Python's native `json` module. If a full detailed explanation of how metadata works refer to Section B.

Once the JSON metadata file (e.g., `hier_metadata.json`) and the CSV data file (e.g., `hier_data.csv`) are in place, the Python code does the following:

1. Load the JSON into a Python dictionary. This dictionary is checked at runtime to ensure it contains at least "root," "entities," and "relations" sections, each with the required fields.
2. Read the CSV into a `pandas.DataFrame`, which now holds columns such as "Comment ID," "User ID," "Comment," "Date," "Time," and all feature columns.
3. Instantiate the `DataHandler` class with this metadata dictionary and DataFrame. Its constructor immediately validates inputs, infers column types, and invokes the hierarchical graph generator.

No rows are dropped during this process; every comment is preserved, even if its parent is missing or circular. Warnings are printed if the designated root is not found (in which case the code defaults to the first non-null row) or if isolated nodes remain (comments that neither reply to anyone nor are replied to).

5.6.2 General Algorithm for Building a Hierarchical Graph

Regardless of whether the final layout is force-directed or circle-packed, the underlying process of turning flat rows into a generic hierarchical graph follows the Algorithm 1. If more explanation of each step is needed, see Section C.

In essence, this generic algorithm ensures that every row in the DataFrame becomes a node (unless explicitly excluded), that every parent–child relationship is captured, and that orphans/self-referentials attach to the root. The similar node and edge result serves as input to any graph layout, whether force-directed, circle-packed, or otherwise, without further structural changes. So once this is finished, the structure will be inputted to whatever hierarchical graph the user asks for.

Algorithm 1 Build Hierarchical Graph from Flat Data

Require: DataFrame D, Metadata M
Ensure: Node list N, Edge list E

Validate Metadata:
if root object, entities, or relations missing in M **then**
    raise error
end if
Identify Root:
Locate row r where $r[\texttt{root.field}] == \texttt{root.value}$
if not found **then**
    pick first non-null row as fallback
end if
Create root node from r's attributes, flattening nested fields
Initialize Nodes:
$N \leftarrow \{\text{root node}\}$, $E \leftarrow \emptyset$
$A \leftarrow \emptyset$ ▷ Set of added node IDs
Handle Orphans and Self-referentials:
for all row $r \in D$ **do**
    $srcID \leftarrow r[\texttt{source_entity}]$
    $childID \leftarrow r[\texttt{link_key_child}]$
    if $srcID == childID$ or missing parent/child key **then**
        if $srcID \notin A$ **then**
            add $srcID$ as node to N
        end if
        add edge (root $\rightarrow$ $srcID$) to E
        $A \leftarrow A \cup \{srcID\}$
    end if
end for
Build Parent–Child Relations:
for all row $r \in D$ not marked orphan **do**
    $parentID \leftarrow r[\texttt{link_key_parent}]$
    $childID \leftarrow r[\texttt{link_key_child}]$
    if $parentID$ is null or equals root ID **then**
        continue
    end if
    if $parentID \notin A$ **then** add to N, A
    end if
    if $childID \notin A$ **then** add to N, A
    end if
    add edge ($parentID \rightarrow childID$) to E
end for
Warn About Isolated Nodes:
$I \leftarrow$ nodes in N not in any edge of E
if $I \neq \emptyset$ **then**
    issue warning with IDs in I
end if
return (N, E)

5.7 RAG Module

The Retrieval-Augmented Generation (RAG) module is a core component in the system, responsible for combining information retrieval techniques with generative AI to produce grounded and contextually relevant responses.

The general structure of this module aligns with the one previously presented in the **Architecture** section 4.6. However, Figure 5.15 expands on that view by highlighting the main technologies used at each stage of the RAG pipeline. These include document parsing libraries (`PyMuPDF`, `python-docx`, `pandas`), text splitting tools (`LangChain`), vector embedding models (`all-MiniLM-L6-v2`), vector databases (`ChromaDB`), and language generation models (`GPT-4o-mini`)[17, 2, 14, 12, 9, 4, 13]. Each of these technologies plays a critical role in enabling efficient retrieval and accurate context-aware response generation.

5.7.1 Step 1: Document Processing and Text Extraction

The RAG pipeline starts with raw documents—PDFs, Word files, text files, or Excel spreadsheets—uploaded by users, much like the datasets managed in the **Data Handling** section 5.6. The goal is to extract usable text, setting the stage for deeper analysis. Each document type requires a tailored tool:

- **PDFs**: PyMuPDF (`fitz`)[17] extracts text while preserving structure, like headings and paragraphs.
- **Word Documents**: `python-docx`[2] pulls out paragraphs in sequence.
- **Text Files**: Read directly as plain text, needing no special handling.
- **Excel Spreadsheets**: `pandas`[14] converts tables into text strings, linking rows or columns logically.

This process can be expressed as a function T, where a document d and its format f yield extracted text s:

$$s = T(d, f)$$

5.7.2 Step 2: Text Chunking

With the text extracted, the pipeline breaks it into smaller pieces, or *chunks*, to make it manageable for later steps. Large texts overwhelm language models and embedding tools, so we use `RecursiveCharacterTextSplitter` from LangChain[12] to divide s into chunks of 500 characters, with a 50-character overlap to preserve context across boundaries. This overlap ensures that a sentence split between chunks remains intact in at least one, avoiding loss of meaning.

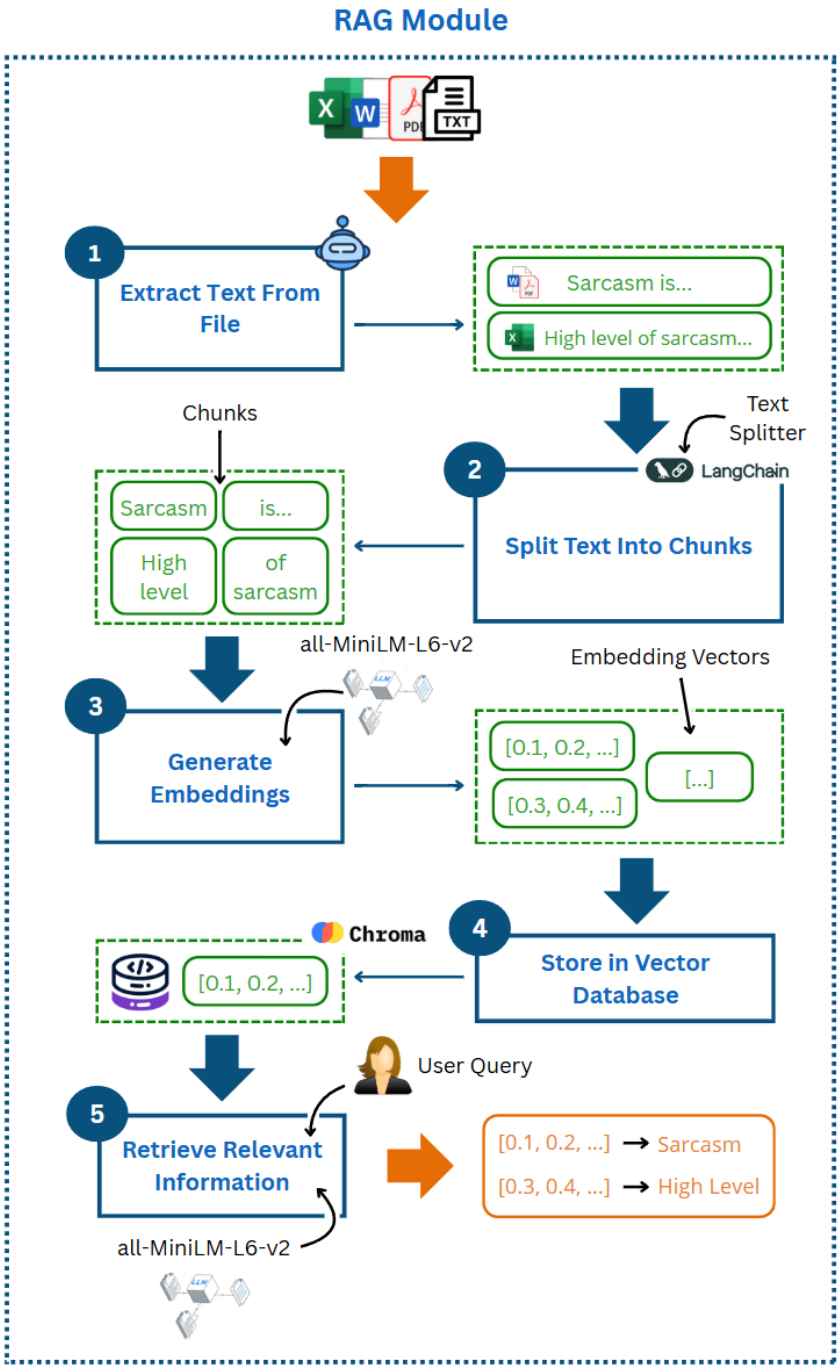


Figure 5.15: RAG Module Scheme with technologies used.

The result is a set of chunks $C = \{c_1, c_2, \ldots, c_n\}$, where the number of chunks n approximates:

$$n \approx \frac{\text{length of } s}{500 - 50} + 1$$

5.7.3 Step 3: Embedding Generation

Now, the pipeline transforms each chunk into a numerical form called an *embedded vector*, a concept central to understanding meaning in text. Think of an embedded vector as a point in a vast imaginary space where each coordinate captures some aspect of the meaning of the text: its topic, tone, or intent. In this project, we used a 384-dimensional space, which means that each vector has 384 coordinates[9]. But what does that mean?

Imagine a 3D space with x, y, and z axes, where the position of a point describes something simple, like an object's location. Now, scale that up to 384 axes (each representing a feature of the text) like 'is it positive?' or 'does it mention toxicity?' These features are not labeled individually; instead, a pre-trained model, `all-MiniLM-L6-v2`, learns them from massive datasets, assigning values so that similar texts end up close together in this space. For a chunk c_i, the model produces a vector $\mathbf{v}_i \in \mathbb{R}^{384}$:

$$\mathbf{v}_i = E(c_i)$$

Here, E is the embedding function and $\mathbb{R}^{384}$ is the real 384-dimensional number space. Inside this vector, the numbers (coordinates) are like a secret code for the meaning of the chunk: small shifts in values reflect subtle differences in content.

5.7.4 Step 4: Vector Storage

These embedded vectors need a home where they can be quickly accessed later. We store them in a Chroma database[4] as pairs of vectors and their original chunks:

$$V = \{(\mathbf{v}_1, c_1), (\mathbf{v}_2, c_2), \ldots, (\mathbf{v}_n, c_n)\}$$

Chroma is optimized for high-dimensional vectors, using an approximate nearest neighbor (ANN) search method called HNSW (Hierarchical Navigable Small World). HNSW builds a network of connections between vectors based on their positions in the 384-dimensional space, speeding up searches by avoiding exhaustive comparisons.

5.7.5 Step 5: Retrieval Mechanism

When a user submits a query q, the pipeline retrieves the most relevant chunks to inform the response. First, the query is embedded into the same 384-dimensional space:

$$\mathbf{v}_q = E(q)$$

Next, we find **chunks** whose vectors are **closest** to $\mathbf{v}_q$. Closeness here is measured by *cosine similarity*, a mathematical tool that compares the direction of two vectors, not their length. Picture two arrows in space: if they point nearly the same way, they're similar, regardless of how long they are. The formula is:

$$\text{sim}(\mathbf{v}_q, \mathbf{v}_i) = \frac{\mathbf{v}_q \cdot \mathbf{v}_i}{\|\mathbf{v}_q\| \|\mathbf{v}_i\|}$$

Here, $\mathbf{v}_q \cdot \mathbf{v}_i$ is the dot product (a sum of multiplied coordinates), and $\|\mathbf{v}_q\|$ and $\|\mathbf{v}_i\|$ are the magnitudes (lengths) of the vectors. The result ranges from -1 (opposite) to 1 (identical), with higher values indicating greater semantic similarity. Using HNSW, the `retrieve_relevant_info` method fetches the top 3 chunks with the highest similarity.[4]

5.7.6 Step 6: Response Generation

The retrieved chunks are combined into a context string t, which the language model (GPT-4o-mini) uses alongside the query q to generate a response:

$$r = G(q, t)$$

For visualizations, context guides the **Visualization Generation** module, ensuring that charts or graphs reflect document insights, for example, highlighting toxicity factors in a bar chart. By blending retrieval with generation, RAG elevates the system beyond static data analysis.

Chapter 6

Evaluation

The goal of this chapter is to present a preliminary usability and performance assessment of the hierarchical data exploration platform. To support this evaluation, a video demonstration explaining the system's functionality and showcasing the evaluation process is available at the following link:

https://drive.google.com/file/d/1ie4juVb19kPhVxH-yoE9YEGjtrxT1i1m/view?usp=sharing.

Also a shorter version explaining the gist of the system proposal is available at the following link:

https://drive.google.com/file/d/1djuaQubFR1J-G5BsI0IYTGkDXHgT17jY/view?usp=sharing.

Six participants representing three expertise levels performed three structured tasks reflecting core workflows: (1) data upload and visualization generation, (2) interacting with the visualization and the chatbot, and (3) editing the visualization and providing feedback.

6.1 Methodology

Sessions were moderated and conducted one-on-one in a controlled environment. Each participant first completed a brief demographic questionnaire (gathering age range, years of experience with visualization tools, and related background). All participants performed the same three tasks in the same order (within-subject design). Before testing with actual participants, a preliminary session with a familiar user refined instructions, timing procedures, and observation protocol. During each task, objective metrics (completion, time, clicks, errors, chatbot queries, and negative facial expressions as a proxy for frustration) were recorded by the moderator. After each task, participants responded to a brief

post-task question on perceived success or satisfaction. At the end of all tasks, participants completed the standard SUS questionnaire (10 items, Likert 1–5).

6.2 Participants

Six primary participants were recruited and grouped by domain expertise:

- **Novice (2 participants):** No prior exposure to hierarchical visualization systems.
- **Intermediate (2 participants):** Students with coursework in data visualization concepts.
- **Expert (2 participants):** Professionals actively working in information visualization.

Demographic details beyond expertise level (e.g., exact age, gender) were collected via questionnaire but are not reported individually here, focusing instead on expertise-related patterns.

6.3 Procedure and Tasks

Each session began with a brief demographic questionnaire, then testing was moderated one-on-one in a controlled environment. A preliminary run with a familiar user verified clarity of instructions and refined timing/observation procedures; its data are not included in the reported tables.

Participants then proceeded through three tasks in fixed order. During each task, the moderator recorded objective metrics: completion (yes/no), time on task (minutes), number of clicks, errors (e.g., wrong file uploads, mis-selections, command syntax issues), chatbot queries, and observed negative facial expressions as a proxy for frustration. After each task, participants answered a brief post-task question about perceived success or satisfaction. Once all tasks were done, they completed the standard System Usability Scale (SUS) questionnaire (10 Likert items) plus three supplementary questions on navigation clarity and feature expectations.

They are described here in sequence:

1. **Upload Data and Generate Visualization:** Participants uploaded a CSV file containing hierarchical data and a JSON metadata file, then waited for the automatic rendering of a force-directed graph. Metrics recorded: completion status; time to render; clicks; errors (e.g., selecting wrong file, needing to restart); any chatbot queries; and observed negative expressions.
2. **Interact with Visualization and Use Chatbot:** With the force-directed graph displayed, participants selected nodes (e.g., via drag-select), opened the chatbot sidebar, submitted a query, and followed a suggestion (for example, generating a popup

chart or requesting an explanation). Metrics: completion; time; clicks; errors (e.g., mis-selections); number of chatbot queries; and observed negative expressions.

3. **Edit Visualization and Provide Feedback:** Participants issued an editing command through the chatbot (e.g., recolor nodes), observed the visualization update, then provided feedback on the chatbot's suggestion using the feedback control. Metrics: completion; time; clicks; errors (e.g., command phrasing issues); chatbot queries; and observed negative expressions.

All participants performed these tasks in the same order (within-subject design). After tasks, SUS and supplementary feedback were collected.

6.4 Results

For each task, Tables 6.1–6.3 show per-user data and summary statistics (mean ± SD). Detailed individual times are presented, but narrative focuses on group-level patterns rather than describing each user in turn.

6.4.1 Task 1 Results

Table 6.1: Task 1 (Upload Data and Generate Visualization): Metrics for Primary Participants

User	Completed	Time (decimal min)	Clicks	Errors	Queries	Neg. Expr.
User1	Yes	5.40	15	3	2	1
User2	Yes	3.53	8	0	1	0
User3	Yes	3.75	12	2	0	0
User4	Yes	2.37	18	4	1	2
User5	Yes	2.40	10	0	1	0
User6	Yes	3.80	14	1	2	1
Mean		3.54±1.12	12.8±3.60	1.67±1.63	1.17±0.75	0.67±0.82

Participants all completed the task. On average, time was 3.54min (SD 1.12), with 12.8 clicks (SD 3.60), 1.67 errors (SD 1.63), 1.17 chatbot queries (SD 0.75), and 0.67 negative facial expressions (SD 0.82). Novices showed higher time and error rates than experts; intermediates fell in between.

6.4.2 Task 2 Results

All participants completed the interaction task. Mean time was 4.74min (SD 1.18), mean errors 1.00 (SD 0.89), mean queries 1.17 (SD 0.75), and mean negative expressions 0.50

Table 6.2: Task 2 (Interact with Visualization and Use Chatbot): Metrics for Primary Participants

User	Completed	Time (decimal min)	Clicks	Errors	Queries	Neg. Expr.
User1	Yes	6.40	20	2	2	1
User2	Yes	3.53	12	0	1	0
User3	Yes	4.37	15	1	1	0
User4	Yes	5.50	18	2	1	1
User5	Yes	3.40	10	0	0	0
User6	Yes	5.25	16	1	2	1
Mean		4.74±1.18	15.2±3.71	1.00±0.89	1.17±0.75	0.50±0.55

(SD 0.55). Novices tended to take longer and issue more queries; experts were faster with fewer errors or queries.

6.4.3 Task 3 Results

Table 6.3: Task 3 (Edit Visualization and Provide Feedback): Metrics for Primary Participants

User	Completed	Time (decimal min)	Clicks	Errors	Queries	Neg. Expr.
User1	Yes	7.02	25	3	2	2
User2	Yes	4.92	15	0	2	0
User3	Yes	5.63	18	1	1	1
User4	Yes	6.28	22	2	1	1
User5	Yes	3.92	12	0	1	0
User6	Yes	5.73	20	1	2	1
Mean		5.58±1.08	18.7±4.72	1.17±1.17	1.50±0.55	0.83±0.75

All participants completed the editing and feedback task. Average time was 5.58min (SD 1.08), with average 1.17 errors (SD 1.17), 1.50 queries (SD 0.55), and 0.83 negative expressions (SD 0.75). Novices experienced more errors and frustration; experts completed more quickly with minimal issues.

6.4.4 SUS Results

Mean SUS score was 64.17 (SD 24.78) (figure 6.4), below the common industry benchmark of 68. Experts scored higher (e.g., 90 and 95), while novices ranged from 30 to 65. Supplementary questions indicated that experts found navigation clear and features met expectations, whereas novices occasionally reported uncertainty about context and feature discoverability.

Table 6.4: Individual SUS Scores for Primary Participants

User	SUS Score
User1	50
User2	95
User3	65
User4	30
User5	90
User6	55
Mean	64.17±24.78
Range	30–95

6.5 Discussion

The preliminary evaluation shows that all core workflows (data upload, interaction, editing, feedback) can be completed by participants across expertise levels. Expert users completed tasks more quickly, with fewer errors and queries, indicating that the interface aligns with domain-expert mental models. Novice users experienced friction at multiple points, suggesting directions for improvement:

- **Onboarding and Labels:** Novices struggled to locate upload controls and understand required formats. Adding clearer labels, introductory tooltips, or a brief guided walkthrough could reduce initial confusion.
- **Selection Mechanisms:** Mis-selections during node selection indicate need for refinements such as improved drag-select feedback, zoom controls, or an alternate selection tool (e.g., lasso).
- **Chatbot Command Guidance:** Errors in phrasing editing commands suggest enhancing natural-language parsing or providing example commands/prompts, perhaps via a help menu or inline suggestions.
- **Feedback Control Visibility:** Some participants had difficulty finding the feedback button; making this control more prominent or placing it in a consistent, easily discoverable location would help.
- **Context Awareness:** Novices sometimes felt uncertain about their current context in the workflow. Persistent headers, breadcrumbs, or progress indicators could reinforce orientation.

Although the overall mean SUS score (64.17) is below 68, the high scores from expert participants (mean 90) suggest that the platform meets expert needs. Since the target users are trained visualization professionals, expert satisfaction carries more weight; nonetheless, improving the onboarding experience would facilitate adoption by less-experienced collaborators or new team members.

6.6 Limitations

This evaluation is preliminary and has limitations:

- **Sample Size and Composition:** Only six participants were included, spanning novices to experts even though the target audience is experts. This limits statistical generalizability.
- **Controlled Environment:** Testing in a lab setting does not capture real-world variability (network conditions, hardware diversity, different screen sizes).
- **Observer Bias in Frustration Metrics:** Negative facial expressions were recorded by observers, introducing subjectivity.
- **Early Release Context:** Testing was performed on an early release without planned refinements (e.g., onboarding guidance), so results should be interpreted in the context of an evolving platform.

Future work includes larger-scale testing with expert users in realistic environments, automated analytics for behavioral metrics, and A/B tests of suggested improvements before a broad rollout.

Chapter 7

Conclusions and Future Work

This project aimed to create a system that combines natural language processing (NLP) with interactive visualizations, making it easier for users to explore hierarchical, multivariate datasets. The main goal—bringing together a conversational chatbot with dynamic visualization tools—has been achieved. Users can now issue commands like "Color by department" or "Generate a pie chart of toxicity levels," and the system updates the visuals quickly and accurately.

Several key objectives were met during development. The NLP component, based on a fine-tuned DistilBERT model, reached strong accuracy in understanding user intents, improving the overall interaction. Visualization features were implemented using D3.js and Plotly, supporting force-directed and circle-packing graphs, as well as interactive actions such as zooming, panning, node selection, dynamic styling, and creating additional charts like bar, pie, and line graphs based on the hierarhical graph. The Retrieval-Augmented Generation (RAG) module helped the chatbot provide relevant answers using uploaded documents, and the platform demonstrated good performance with large datasets while handling invalid inputs gracefully.

That said, some goals remain partially complete. Multi-criteria commands—such as combining node size and color based on different attributes—are only supported to a limited extent. We would like to implement an undo functionality to go to th previous chart or maybe have some state saving. Also, the chatbot's ability to interpret more complex or nuanced queries has room for growth.

Looking ahead, future work will focus on expanding the system's capabilities. This includes adding new visualization types like treemaps, Sankey diagrams, and chord diagrams to offer more ways to analyze data. Enhancing conversational control—especially for commands involving multiple criteria and richer interactions—will make the platform more powerful and flexible for expert users. Create a undo system to allow multiple steps of rollback, as well as integrating more advanced language models (such as GPT-4 or newer), could further boost usability and accuracy. Finally, ongoing development should be guided by actual user feedback and usage data to ensure continuous improvement.

In summary, this project successfully combines natural language interaction with interactive visualization, fulfilling its core goals and establishing a solid base for future enhancements.

Appendix A

Technical Manual

This technical manual provides guidance for accessing and utilizing the HierAI project. The source code and supporting materials are hosted in a GitHub repository, enabling authorized readers, such as project advisors and evaluators, to test the project locally. This chapter describes the repository's contents, access requirements, ensuring users can explore HierAI's functionality while adhering to its copyright restrictions.

A.0.1 Repository Access

The repository is located at:

https://github.com/Xz9RubenSG/HierAI

As the repository is private, access requires authorization. Project readers and advisors must request access by contacting the author at [magicruben2003@gmail.com]. Upon approval, users will be granted read-only access, allowing them to clone the repository but prohibiting any modifications or contributions to the main branch. The repository is configured with branch protection rules, ensuring that only the author can merge changes or push to the `main` branch.

Appendix A

Intent And Label Recognition Ouput By LLM


```
{
  "action": {
    "label": "GeneratePopUpChart",
    "intent": "CreateBarChart",
    "columns": ["sales"],
    "details": "Create bar chart in popup",
    "target": "popup_chart"
  },
  "secondary_action": {
    "label": null,
    "intent": null,
    "columns": [],
    "details": "",
    "target": "none"
  },
  "use_current_figure": false,
  "validation": {
    "is_valid": true,
    "errors": []
  },
  "confidence": {
    "label_confidence": 0.95,
    "intent_confidence": 0.90
  },
  "reasoning": "Query requests bar chart, context indicates popup, so selected GeneratePopupChar
  "fallback": ""
}
```

Appendix B

Metadata Full Explanation

The metadata must specify:

- **Root Identification**: Which column and value denote the hierarchy's starting node (for instance, indicating that "Comment ID = 0" is the root).
- **Entities**: A list of entity types—each with:
  - An identifier field (e.g., "Comment ID"),
  - A set of attributes (which may include simple columns like `author` or nested structures such as a timestamp with separate date and time fields),
  - Optionally, a list of "feature" attributes (e.g., ''`Comment level`'', ''`Argumentation`'', etc.).
- **Relations**: Definitions of how one row relates to another, including:
  - The `source_entity` (the column that holds the child's ID).
  - The `target_entity` (the column that holds the parent's key).
  - `link_key_parent` and `link_key_child` (which columns link a child row to its parent row).
  - The type of relation, usually "hierarchical" for a parent–child relationship, but it can also be "direct" if non-hierarchical linking is required.
- **Rules (Optional)**: Additional traversal guidelines—such as explicitly stating which keys to use when walking from child to parent—useful when multiple relations exist.
- **Color Mappings (Optional)**: Mapping certain attribute values (for example, "Toxicity level" = 0, 1, 2) to specific colors (green, yellow, red), to be used later in the visualization layer.
- **Attribute Colors (Optional)**: A fixed color assigned to each feature column, which can be used when rendering a pie chart of features inside each node.

- **Tooltip and Pie Attributes**: Lists of columns that should appear in a hover-tooltip or be used to slice an in-node pie chart.

To exemplify the metadata for further understanding, we tested in our project a metadata to work with a "comments" dataset. Its main points are:

- The root is defined by the pair `{"field":"Comment ID", "value":0}`, meaning we locate the row where `Comment ID == 0` and treat that as the root comment.
- We have a single entity type called `Comment`. Its identifier field is `"Comment ID"`, and its attributes include:
  - `"author"` (the user ID),
  - `"content"` (the text of the comment),
  - `"timestamp"` (split into "Date" and "Time"),
  - A list of feature columns: `"Comment level"`, `"Argumentation"`, `"Constructiveness"`, `"Toxicity level"`, `"Thread"`, `"Sarcasm"`, ... up to `"Toxicity"`.
- We define one hierarchical relation named `"replies_to"`, where:
  - `source_entity` = `"Comment ID"`,
  - `target_entity` = `"Thread"`,
  - `link_key_parent` = `"Comment ID"`,
  - `link_key_child` = `"Thread"`,
  - `type` = `"hierarchical"`.

  In other words, if a comment row has `Thread = 5`, it is a reply to the comment whose `Comment ID = 5`.
- We also include a small `rules` block (e.g., `"hierarchical_traversal": {"root":"Comment", "parent_key":"Thread", "child_key":"Comment ID"}`) to explicitly state how to locate parents and children, though the code can usually infer this.
- Two optional sections specify color settings:
  - **color_mappings** for "Toxicity level," mapping values 0→green, 1→yellow, 2→red.
  - **attribute_colors** for each feature column (e.g., `"Comment level"`→red, `"Argumentation"`→dark green, etc.).
- Finally, `tooltip_attributes` lists all columns to show when hovering over a node, and `pie_attributes` lists those used to build an in-node pie chart (for instance, `"Constructiveness", "Argumentation", "Sarcasm", ...`).

In narrative form, the metadata says:

"Find the comment whose `Comment ID` is 0. That is the root. Treat each row's `Thread` value as pointing to its parent comment's `Comment ID`. Extract all specified features from each row (including nested timestamp fields), and apply the given color mappings so that when we visualize the graph, 'Toxicity level = 2' will appear red, etc. We will also display a tooltip containing the comment text, author, and all listed feature values, and draw a small pie chart inside each node using the features from `pie_attributes`."

Appendix C

Full Hierarchical Algorithm Explanation

1. **Validate Metadata**: Confirm that the JSON includes:
    - A `root` object with both `field` and `value`.
    - An `entities` block where each entity has an `id` field and an `attributes` definition.
    - A `relations` list where each relation includes `name`, `source_entity`, `target_entity`, `link_key_parent`, `link_key_child`, and `type`.
2. **Identify the Root Row**:
    - Look up the row in the DataFrame where `<root field> == <root value>`.
    - If none is found, issue a warning and select the first non-null row in that column as a fallback.
    - Extract all defined attributes for that row—flattening any nested structures (for instance, separate date and time fields)—to form the root node.
3. **Initialize Node Collection**:
    - Create an empty list (or dictionary) to hold every node that will appear in the graph.
    - Insert the root node first, marked with its ID and type.
    - Maintain a set (or similar) of "added IDs" to avoid duplicates.
4. **Classify Orphans and Self-Referentials**:
    - For each row in the DataFrame:
        - Let `srcID` = the value in `source_entity` column.
        - Let `childID` = the value in `link_key_child` column.

  - If `srcID == childID` (self-reference) or if either `link_key_parent` or `link_key_child` is missing, that row is considered "orphaned."
- For each orphan row (excluding the root), create a node if needed and connect it directly to the root by adding an edge `(rootID → srcID)`. Attach any edge attributes (e.g., "Toxicity level," "Comment level").
- Mark each such `srcID` as "added" to avoid duplication later.

5. **Build Parent–Child Relationships**:
   - Iterate through the DataFrame again for every non-orphan row:
     - Extract `parentID` from the `link_key_parent` column.
     - If `parentID` is null or equals the root ID, skip (already handled).
     - Search for rows in which `link_key_child == parentID`, obtaining all `childID`s.
   - For each discovered `(parentID, childID)` pair:
     - Ensure a node exists for `parentID` (create it if not added).
     - Ensure a node exists for `childID` (create it if not added).
     - Add an edge `(parentID → childID)`, including any specified edge attributes.
   - Continue until every row has been processed for parent–child linkage.
6. **Detect and Warn About Isolated Nodes**:
   - After building all edges, collect all node IDs that never appear as either source or target in any edge.
   - Issue a warning listing these isolated nodes (they have neither parents nor children).
7. **Finalize Data Structures**:
   - At this point, we have:
     - A complete collection of `nodes`, each with its ID, type, and attribute map.
     - A complete collection of `edges`, each with source ID, target ID, relation name, and edge attributes.
   - These can be serialized directly into JSON or passed to visualization routines.

Bibliography

[1] Michael Bostock, *D3.js*, https://d3js.org/.

[2] Steve C. Buehler, *python-docx documentation*, https://python-docx.readthedocs.io/en/latest/.

[3] Joseph Chee Chang, Yongsung Kim, Victor Miller, Michael Xieyang Liu, Brad A Myers, and Aniket Kittur, *Tabs.do: Task-centric browser tab management*, Proceedings of the 34th Annual ACM Symposium on User Interface Software and Technology, ACM, 2021, pp. 1–14.

[4] Chroma Team, *Chromadb documentation*, https://docs.trychroma.com/.

[5] Deepset, *Haystack: An nlp framework for question answering, summarization, and more*, https://github.com/deepset-ai/haystack, 2021.

[6] Explosion AI, *Spacy usage documentation*, https://spacy.io/usage.

[7] Enamul Hoque, Vid UK Setlur, Melanie Tory, and Isaac Dykeman, *Applying pragmatics principles for interaction with visual analytics*, IEEE Transactions on Visualization and Computer Graphics **24** (2017), no. 1, 309–318.

[8] Jianbin Huang, Yiwen Xi, Jiangtao Hu, and Jianhua Tao, *Flownl: Asking the flow data in natural languages*, IEEE Transactions on Visualization and Computer Graphics **29** (2022), no. 1, 1200–1210.

[9] *all-minilm-l6-v2 model*, https://huggingface.co/sentence-transformers/all-MiniLM-L6-v2.

[10] *Distilbert base uncased model*, https://huggingface.co/distilbert/distilbert-base-uncased.

[11] Ecem Kavaz, Anna Puig, and Inmaculada Rodríguez, *Chatbot based natural language interfaces for data visualisation: A scoping review*, Applied Sciences **13** (2023), no. 12, 7025.

[12] LangChain Team, *Langchain documentation*, https://python.langchain.com/docs/.

[13] OpenAI, *Openai api documentation*, https://platform.openai.com/docs/.

[14] Pandas Development Team, *Pandas documentation*, https://pandas.pydata.org/docs/.

[15] Rohan Patel, Sayak Das, Yifan Guo, Aadit Joshi, Harshita Ramaswamy, Eric Horvitz, Ece Kamar, and Aman Shukla, *Lida: Language-interface for data analysis*, https://github.com/microsoft/lida, 2024.

[16] Plotly, Inc., *Plotly javascript*, https://plotly.com/javascript/.

[17] PyMuPDF Team, *Pymupdf documentation*, https://pymupdf.readthedocs.io/en/latest/.

[18] Colin Raffel, Noam Shazeer, Adam Roberts, Katherine Lee, Sharan Narang, Michael Matena, Yanqi Zhou, Wei Li, and Peter J. Liu, *Exploring the limits of transfer learning with a unified text-to-text transformer (t5)*, https://github.com/google-research/text-to-text-transfer-transformer, 2020.

[19] Arjun Srinivasan and Vidya Setlur, *Snowy: Recommending utterances for conversational visual analysis*, Proceedings of the 34th Annual ACM Symposium on User Interface Software and Technology, ACM, 2021, pp. 864–880.

[20] Arjun Srinivasan, John Stasko, and Steven M. Drucker, *Nl4dv: Natural language generation for data visualization*, https://github.com/nl4dv/nl4dv, 2020.

[21] Yifan Wang, Shixia Liu, Hongyu Guo, and Huamin Qu, *Interactive visual analysis of multivariate time series data using natural language queries*, IEEE Transactions on Visualization and Computer Graphics **28** (2022), no. 1, 1–11.