Retrieval Augmented Chatbots powered by Large Language Models for Semantically Structured Data

Omijkumar Pravinbhai Mangukiya

A Thesis

in

The Department

of

Computer Science and Software Engineering(CSSE)

Presented in Partial Fulfillment of the Requirements

for the Degree of

Master of Computer Science (Computer Science) at

Concordia University

Montréal, Québec, Canada

March 2025

Concordia University

School of Graduate Studies

This is to certify that the thesis prepared

By: **Omijkumar Pravinbhai Mangukiya**

Entitled: **Retrieval Augmented Chatbots powered by Large Language Models for Semantically Structured Data**

and submitted in partial fulfillment of the requirements for the degree of

Master of Computer Science (Computer Science)

complies with the regulations of this University and meets the accepted standards with respect to originality and quality.

Signed by the Final Examining Committee:

_____ Chair
Dr. Abdelhak Bentaleb

_____ External Examiner
Dr. Abdelhak Bentaleb

_____ Examiner
Dr. Sandra Cespedes

_____ Supervisor
Dr. Essam Mansour

Approved by _____
Paquet, Joey, Chair
Department of Computer Science and Software Engineering(CSSE)

_____ 2025

_____
Mourad Debbabi, Dean
Faculty of Engineering and Computer Science

Abstract

Retrieval Augmented Chatbots powered by Large Language Models for Semantically Structured Data

Omijkumar Pravinbhai Mangukiya

Recent advancements in Large Language Models (LLMs) have transformed Natural Language Processing, yet challenges such as factual inaccuracies and inadequate reasoning over structured data persist. Retrieval-Augmented Generation (RAG) systems address these issues by grounding LLMs in external knowledge. However, conventional RAG methods typically treat knowledge sources as unstructured text, overlooking the semantic relationships vital in domains like enterprise data and healthcare. This thesis introduces a Graph-based RAG approach that leverages the structured nature of graph data to enhance both retrieval and response generation by preserving these semantic relationships. The research focuses on developing conversational question answering systems over semantically structured data, specifically targeting JIRA Issues and Knowledge Graphs through two distinct applications. The core innovation lies in maintaining the inherent data relationships during both retrieval and generation phases by employing structured queries and graph traversal techniques. This method not only allows for domain-specific optimization but also demonstrates improved performance and efficiency compared to traditional RAG methods.

Acknowledgments

I would like to express my deepest gratitude to Dr. Essam Mansour for his invaluable guidance and unwavering support throughout this research journey. His insightful feedback and expertise have been instrumental in shaping this thesis. I also extend my sincere appreciation to the members of the CODS lab for providing a stimulating and supportive research environment. A special thanks to Reham Omar, with whom I thoroughly enjoyed every discussion and collaboration on research ideas.

I am also grateful for the amazing friends I have made here in Montreal, especially Akshit, Shivam, and Deep, for their continuous motivation and unwavering belief in me throughout this journey. Lastly, I am forever indebted to my family for their unconditional love and support. Their belief in me has been my greatest source of strength.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

1.1 Overview

Recent advancements in Large Language Models (LLMs) have pushed the frontier of the field of natural language processing (NLP), especially the evolution of conversational question answering systems like OpenAI's ChatGPT(OpenAI (2022)) and Google DeepMind's Gemini(Google (2023)), which has changed the way we retrieve and explore new information. Among many impressive capabilities, these models excel at generating coherent and contextually relevant text, understanding complex and ambiguous queries, and translating between numerous languages(Minaee et al. (2024)). However, despite their remarkable performance, LLMs inherit several limitations. Among these are the factual inaccuracies, challenges of hallucinations, difficulties in handling evolving knowledge, and an inability to perform robust reasoning over complex and semantically structured domain specific information(Kandpal, Deng, Roberts, Wallace, and Raffel (2023)).

Retrieval-Augmented Generation (RAG) has emerged as a powerful approach to overcome some limitations of LLMs(Gao et al. (2024)). RAG systems enhance LLMs by leveraging external, continuously updating knowledge sources. These sources can be e.g., databases, knowledge graphs, documents. In a typical RAG setup, user queries are enriched with relevant information retrieved from the external knowledge sources, which makes the

generated responses more relevant and factually grounded, improving the accuracy and relevancy. This approach makes it more suitable for real-time applications and domain-specific tasks, where factual accuracy and up-to-date information are necessary.

Traditional RAG systems treat external knowledge sources as unstructured flattened plain text or simple document(X. Ma, Gong, He, Zhao, and Duan (2023b)), overlooking the underlying semantic relationships and inherent structures in complex datasets. This approach maybe sufficient for many simple question-answering tasks, it may lead to sub-optimal performance, especially for domains that require the preservation of underlying semantic relationships(Gao et al. (2024)). Which are crucial for effective reasoning and accurate response generation. For example, in knowledge-rich environments such as enterprise data management, customer service & technical support, and healthcare information systems, the inter-dependencies and relationships between data points are as important as the data points themselves. Ignoring such relationships can lead to document fragmentation, inaccurate lower quality generations (Xu et al. (2024)).

Knowledge Graphs (KGs) have emerged as structured representations of interlinked data, integrating it with RAG systems has created new opportunities in advancing it further (Li et al. (2023)). KGs organize the information as nodes and edges, capturing entities and their relationships. Recent research and practical implementations, such as Microsoft's advances in Graph-RAG(Edge et al. (2024)) and LinkedIn's exploration of RAG for customer service QA(Xu et al. (2024)), underscore the potential benefits of leveraging KGs in complex query scenarios. Their findings highlight that RAG systems optimized for semantically structured knowledge sources can outperform traditional flat-document based approaches, particularly while dealing with multi-layered information.

Compared to traditional RAG, where retrieval solely relies on text-based heuristics, in Graph-based RAG systems, query understanding and context retrieval must be treated differently. Graph RAG systems can utilize specialized retrieval mechanisms that consider the semantic graph structure. For example, identifying the entities and intent from query and utilizing it for retrieving paths, subgraphs, or triplets that represents the relationships between entities in the query and provides a richer and more meaningful context(Peng et al.

(2024)). In addition, managing and storing intermediate or pre-processed graph-structured knowledge efficiently which can scale easily, while maintaining semantic connections, is essential. Hybrid techniques such as using query languages like SPARQL for KGs, or embedding-based search in vector databases are being explored to optimize this process.

1.2 Contributions

This thesis presents applying RAG for conversational question answering (or chatbot) over semantically structured data from two distinct domains: Knowledge Graph Question Answering (KGQA) and Project Management using JIRA issues. To address various domain-specific challenges, two applications employing the Graph RAG framework were developed. The key contributions of this work are as follows:

(1) Preserving Semantic Structure: Our approach emphasizes the importance of maintaining the inherent relationships in underlying data during the retrieval and response generation phases. Instead of reducing the graph structured knowledge data to flat plain text documents, I explore methods that retain the semantic integrity, such as utilizing structured queries and graph traversal techniques.

(2) Domain-Specific Optimization: Our approach can be extended to other domains with similar characteristics, making it versatile and domain independent.

(3) Performance and Efficiency: Our approach is more efficient because of preservation of semantic relationships and cost effective than traditional RAG. By optimizing data retrieval and leveraging structured representations, the methods achieves faster response times and reduced computational overhead.

1.3 Outline

This thesis comprises five chapters, each exploring different aspects of two Retrieval Augmented Generation (RAG) applications I developed in distinct domains: Knowledge Graph Question Answering (KGQA) and Project Management using JIRA issues. Chapter

2 presents a comprehensive review of related literature, covering key areas such as RAG, Large Language Models (LLMs), and Conversational Question Answering (CQA). Chapter 3 details the architecture, implementation, and evaluation of the KGQA-RAG Chatbot, while Chapter 4 focuses on the system architecture and implementation of the JIRA-RAG Chatbot, including a taxonomy that compares it with existing market solutions. Chapter 5 concludes the thesis detailing the limitations and future work.

Chapter 2

Background

2.1 Large Language Models

Large Language Models (LLMs) represent a pivotal advancement in conversational AI, showcasing exceptional capabilities in understanding and generating natural language. Their evolution has roots in statistical language models, which are effective within limited contexts, lacks the robustness to handle complex language patterns. Neural language models marked a breakthrough, and the introduction of pre-trained language models (PLMs) enabled substantial improvements across diverse natural language tasks. Scaling these PLMs has led to today's LLMs (LLAMA, Gemini, GPT) models with tens to hundreds of billions of parameters, trained on vast text corpora to achieve unprecedented language understanding and generation abilities (Minaee et al. (2024)).

2.1.1 Types of LLMs

Most modern LLMs are built upon the Transformer architecture (Vaswani et al. (2017)), which introduced a self-attention mechanism, allowing models to dynamically weigh the importance of different words within a sequence. This capability is central to learning complex

and long-range dependencies in a sequence, which are essential for both language understanding and generation. The self-attention function can be mathematically represented as:

$$\text{Attention}(Q, K, V) = \text{softmax}\left(\frac{QK^T}{\sqrt{d_k}}\right)V$$

where Q, K, and V represent the query, key, and value matrices, respectively, and d_k is the dimensionality of the key vectors. This formulation enables efficient parallelization and is foundational in scaling LLMs. LLMs can be categorized into three main types, each suited for different types of tasks based on their architecture and pre-training objectives:

Model Name	**Type**	**Parameters (Size)**	**Fine-tuned Variants**	**Accessibility**
BERT	Encoder-only	Base (110M), Large (340M)	Foundation	Public
T5	Encoder-Decoder	Base (220M), Large (770M)	Foundation	Public
FLAN-T5	Encoder-Decoder	Base, Large, XL	Instruction	Public
GPT-3	Decoder-only	Up to 175B	Foundation, Instruction	OpenAI API
LLaMA 2	Decoder-only	7B, 13B, 70B	Foundation, Chat	Gated (Meta)
PaLM	Decoder-only	Up to 540B	Foundation, Instruction	Private (Google)
GPT-4	Decoder-only	Undisclosed	Chat	OpenAI API
Claude	Decoder-only	Undisclosed	Chat	Anthropic API
Gemini	Decoder-only	Undisclosed	Chat	Gemini API

Table 2.1: Overview of notable large language models categorized by architecture type, parameter size, fine-tuning approaches, and availability. Model details are sourced from their respective publications: BERT Devlin et al. (2019), T5 Raffel et al. (2020), GPT-3 Brown et al. (2020), LLaMA 2 Touvron et al. (2023), PaLM Chowdhery et al. (2022), FLAN-T5 Chung et al. (2022), GPT-4 OpenAI (2023), Claude Anthropic (2023) and Gemini Team (2024).

(1) Encoder-Only Models: Models like BERT are optimized for understanding tasks, such as text classification and question answering. These models process input sequences as a whole, generating contextualized embeddings that capture the relationships between tokens. Pre-training tasks for these models often include Masked Language Modeling (MLM), where certain tokens are masked and predicted by the model to enhance contextual understanding.

(2) Decoder-Only Models: Models in the GPT family are designed primarily for text generation tasks. They employ autoregressive generation, predicting each token sequentially based on previous tokens in the sequence. Pre-training for these models often involves tasks like Next Token Prediction, which encourages them to generate coherent and contextually appropriate continuations for a given text prompt.

(3) Encoder-Decoder Models: Also known as sequence-to-sequence models, these models combine encoder and decoder structures, making them ideal for tasks like machine translation and summarization, where an input sequence is transformed into a different output sequence. Pre-training objectives typically involve learning the relationships between source and target sequences, as seen in models like T5 and BART, which are trained on denoising tasks to improve the ability to rephrase, summarize, and translate text.

2.1.2 LLM's Capabilities

LLMs have transformed the landscape of Natural Language Processing (NLP) by offering a diverse range of capabilities that extend from fundamental NLP tasks to sophisticated reasoning abilities (Minaee et al. (2024)). In this section, we discuss the primary capabilities of LLMs:

(1) Basic NLP Tasks: These include text classification, sentiment analysis, named entity recognition, question answering, and text generation. LLMs are highly effective in these foundational NLP tasks due to their ability to capture contextual semantics from large pre-training corpora.

(2) Emergent Abilities: LLMs exhibit remarkable emergent capabilities that arise from their training scale and structure, despite not being explicitly programmed. These abilities include **in-context learning**, where the model learns new tasks from a few examples provided in the input prompt; **instruction following**, enabling it to perform tasks based on natural language instructions without requiring explicit retraining; and **multi-step reasoning**, which allows the model to break down complex

tasks into smaller steps to derive accurate answers—an essential skill for question answering and problem-solving.

(3) Augmented Capabilities: LLMs can be combined with external data sources or tools (such as knowledge bases) for enhanced capabilities, creating systems that use retrieval mechanisms to ground their responses in factual knowledge or interact with structured data to improve accuracy.

2.1.3 Prompt Engineering

Prompt engineering is an emerging field. It focuses on creating and refining prompts for LLMs. This approach boosts performance on tasks like question answering, sentiment classification, and named entity recognition. As LLMs became capable of zero-shot learning, the approach allowed downstream tasks to be reformulated using specialized prompts instead of designing task-specific training objectives. This shift has popularized the 'pre-train, prompt, and predict' model, in contrast to the older 'pre-train, fine-tune' approach.

Introduced by (Wei et al. (2022)), Chain-of-Thought (CoT) prompting enhances complex reasoning in language models by breaking down problems into intermediate reasoning steps. When combined with few-shot prompting, CoT improves performance on tasks that require structured reasoning before generating responses. However, for more complex problems requiring exploration or strategic foresight, traditional prompting techniques often fall short. To address this, (Yao et al. (2023)) proposed Tree of Thoughts (ToT), a framework that extends CoT by encouraging exploration across multiple reasoning paths, enabling more effective problem-solving. Beyond these techniques, self-consistency, introduced by (Wang et al. (2022)), further refines CoT by replacing naive greedy decoding with a strategy that samples multiple diverse reasoning paths. By selecting the most consistent answer from these sampled solutions, self-consistency significantly boosts performance in tasks involving arithmetic and commonsense reasoning.

2.2 Knowledge Graphs

A knowledge base (KB) is a structured information repository used for knowledge sharing and management purposes. Large-scale graph-structured knowledge bases, known as the Knowledge Graphs (KGs), such as Freebase (Bollacker, Evans, Paritosh, Sturge, and Taylor (2008)), DBpedia (Lehmann et al. (2015)), YAGO (Suchanek et al. (2007)) and Wikidata (Vrandečić and Krötzsch (2014)), represent real-world information in a graph format. Nodes and edges in this graph represents the real world entities and relationship between them. Entities in a knowledge graph captures objects, events, situation, or concepts. The relationship between them captures the context and the meaning of how they are connected. The underlying graph-based data model makes it particularly suited for capturing real-world knowledge (Zaib, Zhang, Sheng, Mahmood, and Zhang (2021)). KGs are applied in various domains due to its impressive capabilities. Examples include KGs about scientific publishing (e.g., Microsoft Academic Graph Sinha et al. (2015), DBLP Ley (2009)), general facts representation (e.g., YAGO, DBPedia, and Wikidata), health-science (e.g., BioKG Walsh et al. (2020)).

A Knowledge Graph can be defined as a directed, labelled graph $KG = (V, E, L)$, where V is the set of vertices (nodes), can represent entities or concepts. Each vertex $v \in V$ can also have associated properties or attributes. $E \subseteq V \times V$ is the set of edges, directed relationship between vertexes. An edge $e = (v_i, v_j) \in E$ is a direct link from v_i to v_j. L is the set of labels that annotates the edges. KGs are typically stored in RDF stores, which provides a powerful query language for querying it. SPARQL is one of the query languages, which allows for efficient inferencing and reasoning.

At the heart of the knowledge graph, it consists of a set of triples (s, p, o), where: s (subject) is the focused entity of a statement. p (predicate) is the relationship between subject and object. o (object) is the associating entity for subject through predicate. Figure 2.1 illustrates a graph-based representation of facts about "Justin Bieber" and SPARQL queries to retrieve information about his profession, nationality, and sibling. The graph includes nodes representing entities such as "Justin Bieber", "Changes", and "Canada" and

edges representing relationships like “album”, “parent” and “nationality”.

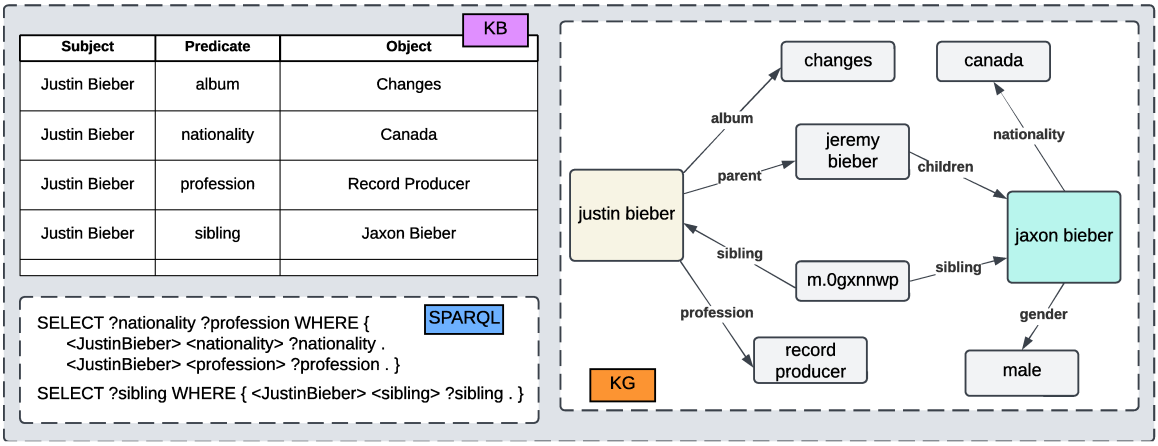

Figure 2.1: Knowledge Graph Example

Knowledge Graph	**#Entities(M)**	**#Triples(M)**	**#Predicates**	**Domain**	**Data Sources**
DBLP	18	263	167	Academic	Research Papers
YAGO	12	207	259	Cross-domain	Wikipedia, WordNet
DBpedia	14	350	60,736	Cross-domain	Wikipedia dumps
MAG	586	13,705	178	Academic	Academic publications
BioKG	10.69	30.76	NA	Biomedical	PubMed literature
ConceptNet	8	21	36	Commonsense	Crowdsourcing, Wiktionary

Table 2.2: Comparative statistics of popular knowledge graphs. Data sources: DBLP (Ley (2009)), YAGO (Suchanek et al. (2007)), DBpedia (Lehmann et al. (2015)), MAG (Sinha et al. (2015)), BioKG (Walsh et al. (2020)), ConceptNet (Speer and Lowry-Duda (2017)). Numbers reflect latest available versions (2020-2023).

The table 2.2 provides an overview of several prominent Knowledge Graphs (KGs), including DBLP, YAGO, DBpedia, and MAG. These KGs varies in size, complexity, with differing numbers of entities and triples. The diversity of data sources and predicates employed in these KGs highlights their potential for addressing a wide range of knowledge-intensive tasks in domain specific areas.

2.3 Retrieval Augmented Generation(RAG)

RAG is an emergent approach within NLP that enhances the performance of LLMs by integrating external knowledge retrieval with generative capabilities. Traditional LLMs, despite their capacity for generating coherent and contextually appropriate text, are often limited by their finite training data, leading to incomplete or outdated information especially in domain-specific or knowledge-intensive tasks, notably generating "hallucinated" content (Minaee et al. (2024)). RAG merges information retrieval with generative models to create dynamic, knowledge-rich responses and cuts down on factual errors. Its use in LLMs has driven widespread adoption, making it a key technology for chatbots and real-world applications Gao et al. (2024).

2.3.1 Different RAG Techniques

The Naive RAG (see Figure 2.2) approach follows a traditional "Retrieve-Read" (X. Ma et al. (2023b)) framework, encompassing indexing, retrieval, and generation. In the indexing phase, raw data in various formats (Markdown, PDF, HTML, Doc) is cleaned, extracted, and converted to plain text. To accommodate language model context limitations, the text is segmented into fixed-size chunks (e.g., 100, 128, 256 tokens), which are then encoded into vector embeddings and stored in a vector database. Retrieval involves identifying relevant documents based on the similarity between the query vector and document embeddings. The retrieved documents, along with the original query, are synthesized into a prompt for the frozen LLM to generate a response. The quality of the generated response depends on the specific task, model parameters, and the information contained within the retrieved documents.

However, Naive RAG faces challenges, including precision and recall issues in the retrieval phase, leading to the selection of irrelevant documents and the omission of crucial information. The generation phase is prone to hallucination, where the model produces content not supported by the retrieved context, and may also suffer from bias and toxicity in the outputs (Gao et al. (2024)).

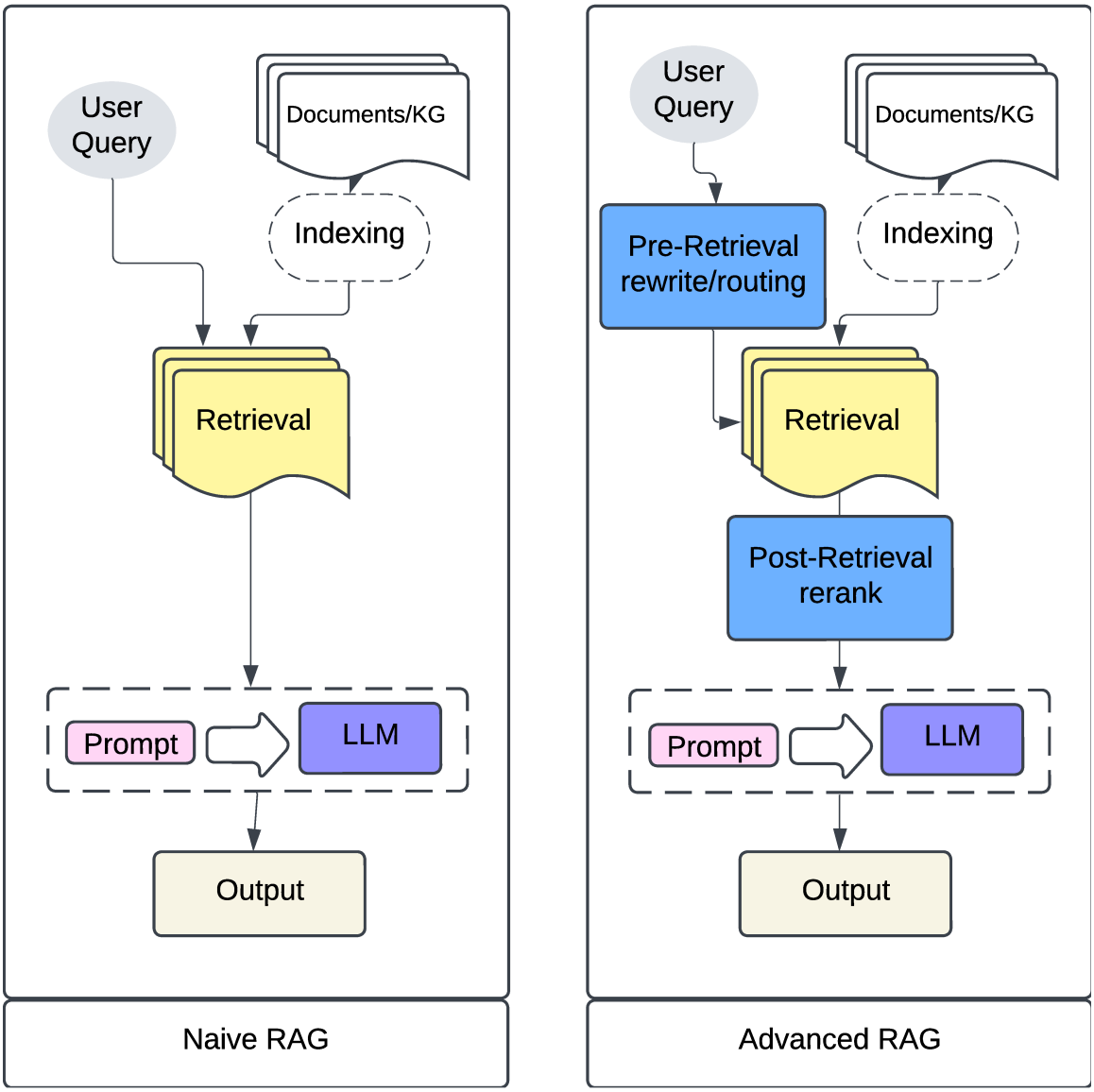

Figure 2.2: Naive vs Advance RAG

Advanced RAG (see Figure 2.2) addresses these limitations by introducing improvements in both retrieval and generation. To enhance retrieval quality, pre-retrieval strategies like query rewriting (X. Ma, Gong, He, Zhao, and Duan (2023a)) and routing, as well as post-retrieval refinements like re-ranking, filtering (Y. Ma, Cao, Hong, and Sun (2023)) and context compression, are employed. Indexing techniques are refined through the use of sliding window approaches, fine-grained segmentation, and metadata incorporation.

To further optimize RAG systems, modular frameworks have emerged. By dividing RAG

into specialized modules for distinct tasks, these frameworks offer better customization and flexibility, enabling adaptation to domain-specific requirements and various scenarios.

2.3.2 Core Components of RAG

RAG systems comprise three main components: Retrieval, Generation, and Augmentation, which work together to enhance the quality and relevance of generated content. This section reviews existing work on these core components in detail.

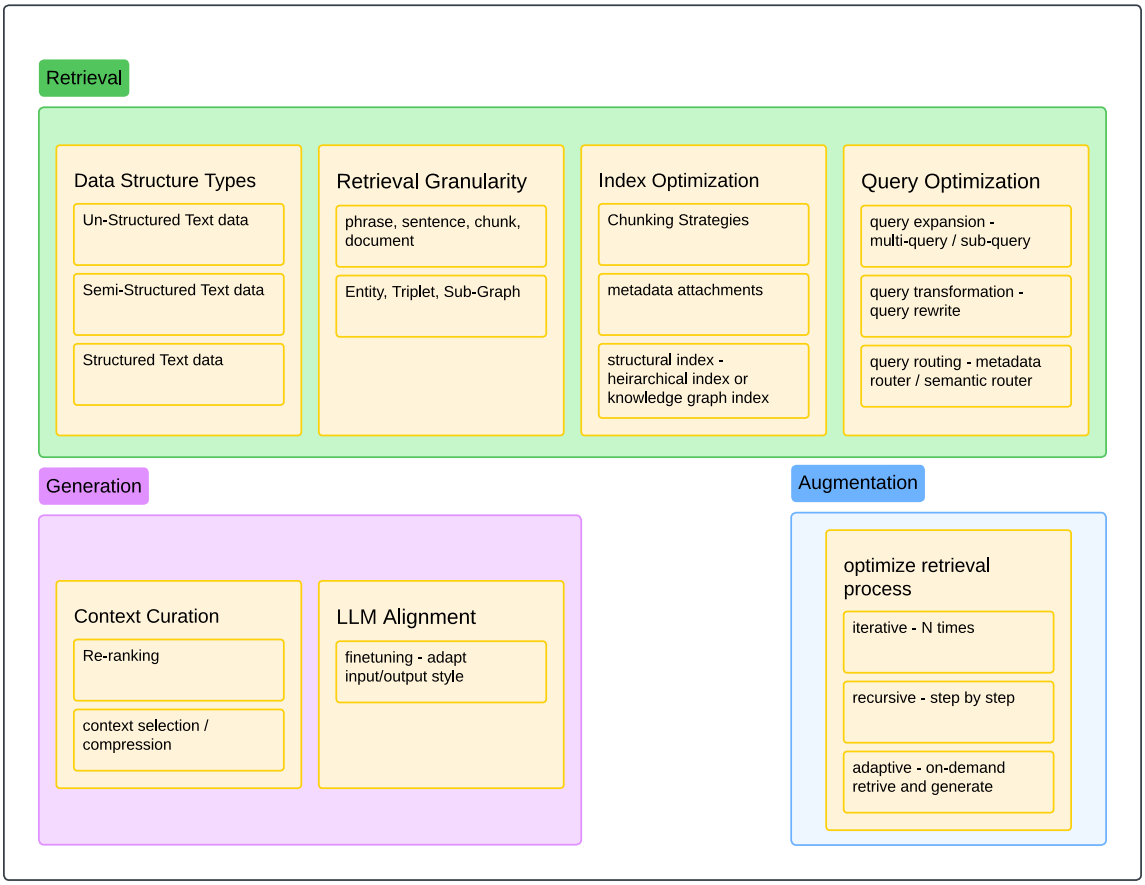

Figure 2.3: Stage-wise decomposition of Retrieval-Augmented Generation (RAG) components: (1) Retrieval with multi-granular data acquisition and query refinement, (2) Augmentation with recursive retrieval optimization processes, and (3) Generation through curated contextual knowledge integration and language model adaptation.

Retrieval Phase

The retrieval phase constitutes the foundational stage of Retrieval-Augmented Generation (RAG) systems, determining both the relevance and contextual adequacy of subsequent generation outputs. It can be understood by exploring various tricks and methods used in this phase, wether its because of the characteristics of the data sources, retrieval granularity, or smart and optimized querying/indexing required (Gao et al. (2024)).

Data Sources & Granularity Retrieval Source and Granularity can be attributed to three kind of data sources. The granularity of retrieved data varies based on the source data structure. For unstructured text, granularity ranges from phrases and sentences to larger chunks or entire documents. In contrast, for structured sources like knowledge graphs, the granularity can be an entity, a triple, or a subgraph represented as a set of triples.

- **Unstructured Text**: For open-domain question-answering (ODQA), techniques like embedding based retrieval are used for efficiently extracting relevant passages from vast text collections like Wikipedia Dump.
- **Semi-Structured Data**: When dealing with a combination of text and tabular data, specialized approaches are required to handle potential misalignment and ensure semantic coherence. Hybrid techniques that combine text-based search with table querying are commonly employed.
- **Structured Data**: For structured formats like knowledge graphs, sophisticated querying mechanisms are used to extract precise answers. Advanced systems, such as KnowledgeGPT (Wang et al. (2023)), leverage program-of-thought prompting to generate queries that can effectively execute on these knowledge bases.

Critical trade-offs emerge between granularity levels: smaller chunks reduce noise but risk contextual fragmentation, while larger segments preserve coherence at computational cost. Contemporary systems mitigate this through sliding window techniques and dynamic chunk merging. The indexing architecture ultimately determines retrieval precision, with hybrid

approaches (vector and graph indices) showing particular promise for multi-modal knowledge recall (Gao et al. (2024)).

Indexing Strategies & Query Optimization During indexing, source documents are processed and segmented. They are then converted into embeddings and stored in vector databases. Various chunking strategies are employed to split documents into fixed-length chunks (e.g., 100, 512 tokens) and generate embeddings for each chunk. While smaller chunks can mitigate noise, they may not fully capture the necessary context. To address this, recursive splitting and sliding window methods can be used to merge related information across multiple chunks. To enhance retrieval quality, metadata information (e.g., paragraph titles, page numbers) can be attached to chunks. This metadata can be used to implement time-aware RAG, assigning higher weights to more recent data.

Hierarchical and structural indexing can be applied to establish parent-child relationships between chunks, with summaries stored at each node. This facilitates efficient data traversal and helps the RAG system determine which chunks to extract. Knowledge graph indexing can further enhance knowledge retrieval and reasoning by linking concepts and entities, enabling the LLM to generate more coherent and contextual responses.

User-provided queries often lack clarity and precision, particularly for complex questions or specialized vocabulary. To address this, various query optimization techniques are employed. One approach involves expanding a single query into multiple queries to provide additional context. This expansion can be achieved through LLM-generated queries or by breaking down the original query into sub-questions. Additionally, query rewriting using LLMs can help improve retrieval effectiveness by transforming the query into a more optimal form. Finally, routing techniques based on query metadata can be used to filter and reduce the search scope, leading to more accurate and efficient retrieval (Gao et al. (2024)).

Generation Phase

Once the retrieval stage has yielded relevant information, the generative model processes this content in conjunction with the original user query to produce an informative and

coherent response. Directly feeding all retrieved information to the LLM can lead to the "lost in the middle" problem (Liu et al. (2023)), where the LLM prioritizes the beginning and end of long texts, neglecting the middle content. To mitigate this, the generation phase requires careful processing of the retrieved documents through content curation or LLM adaptation.

LLM Adaptation To ensure consistent output style and format, LLM fine-tuning can be employed. This involves training the LLM on specific data formats and styles to align its responses accordingly. Targeted fine-tuning, tailored to the specific scenario and data characteristics, can further enhance the quality of generated responses.

The generation phase presents several challenges that impact the reliability and quality of outputs. One major issue is **hallucination**, where the model generates factually incorrect or fabricated content, leading to misinformation. Additionally, ensuring **coherence and fluency** is crucial, as the generated text must maintain consistency, clarity, and contextual relevance to provide meaningful and accurate responses.

Augmentation Phase

The augmentation step refines the generative output by incorporating additional context or feedback. Techniques like iterative refinement and user-driven feedback loops help improve the accuracy and coherence of the final text. This stage aims to mitigate errors and enhance the overall response quality (Gao et al. (2024)).

Iterative refinement searches the knowledge base multiple times. It uses both the initial query and generated text. This process helps the LLM build a deeper understanding of the context. ITER-RETGEN (Shao et al. (2023)) leverages a synergistic approach combining "retrieval-enhanced generation" and "generation-enhanced retrieval" for tasks requiring precise information reproduction.

Recursive retrieval enhances search depth and relevance by iteratively refining queries using previous results. This technique is widely used in information retrieval and NLP. IRCoT (Trivedi, Balasubramanian, Khot, and Sabharwal (2023)) utilizes chain-of-thought reasoning to guide the retrieval process and refines the chain of thought with the retrieved

information.

Adaptive retrieval lets LLMs decide when and what to retrieve, improving efficiency and relevance. Self-RAG (Asai, Wu, Wang, Sil, and Hajishirzi (2023)) enhances quality and accuracy by combining retrieval with self-reflection. It retrieves passages on demand and generates reflective responses.

2.4 Chatbots and Conversational Question Answering

Conversational AI which focuses on developing intelligent dialogue systems that not only can interpret user prompted natural language queries and respond with factual correct and context-aware answers but also carrying out interactive conversation given a topic (e.g., "Election in the USA" to "Origins of Universe"). The advancements in this field can be broadly grouped into three research directions (see Figure 2.4): 1. task-oriented dialogue systems that perform the task given user input, for example doing flights ticket booking, 2. chat-oriented dialogue systems that carry out interactive conversation with user for example ChatGPT. 3. QA dialogue systems that respond with fact-based and context-aware answers extracted or retrieved from the different information sources such as text documents or knowledge bases.

2.4.1 Categorization of CQA Systems

The evolution of question answering systems (CQA) over the years has shifted the focus from the single-turn qa: which processes individual questions without considering the prior context; multi-turn qa: which retains the session-specific information to handle follow-up questions to conversational QA which mimics the human-like dialogues, adapting to dynamic and long-term interactions. The evolution of QA systems has been fueled by advancements in LLMs (e.g., GPT, Gemini, LLAMA) and large-scale datasets like SQuAD (Rajpurkar, Zhang, Lopyrev, and Liang (2016)) and HotpotQA (Yang et al. (2018)). Conversational Question Answering can be further categorized based on different aspects, types of data domains, types of questions and types of data sources. (Zaib et al. (2021)). Figure

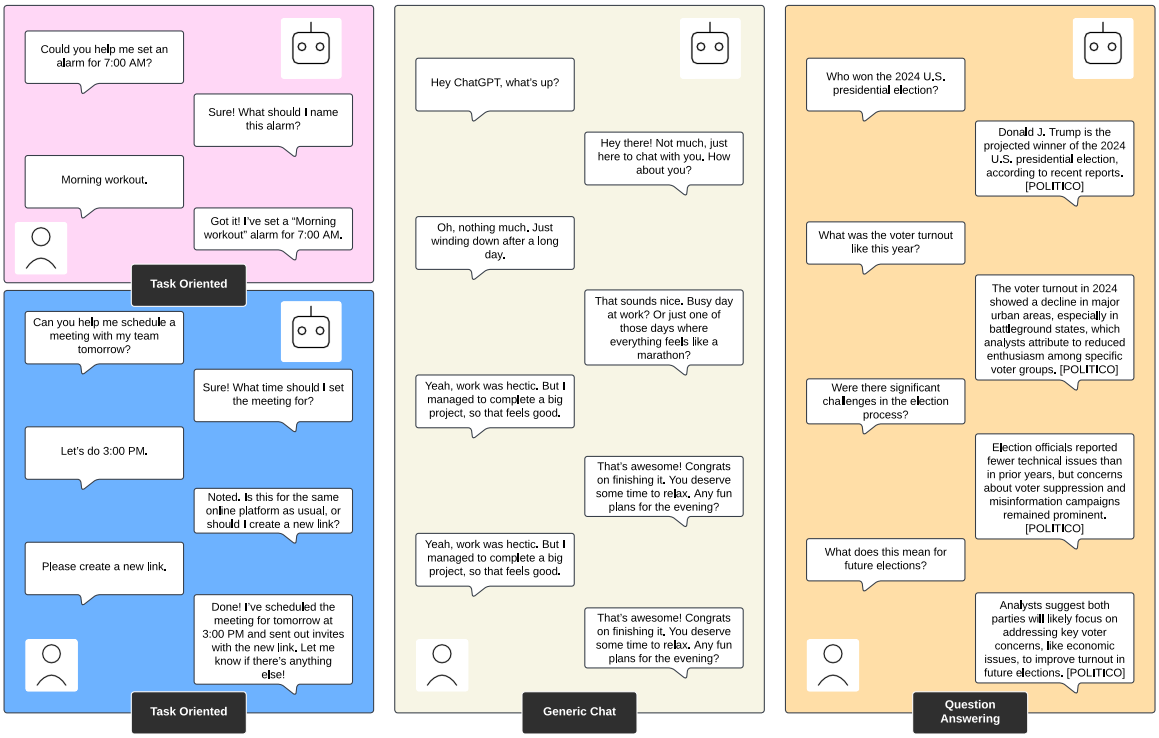

Figure 2.4: Various types of conversational systems: the leftmost represents a task-oriented system, the middle illustrates a general-purpose system, and the rightmost showcases a question-answering system with web search capabilities.

2.5 represents categorization of CQA systems. The rest of the section covers details on each category.

Types of Data Sources Based on the underlying data sources on which question answering systems operate, it can be categorized into the following types:

(1) Structured Data Sources: These sources store data in a well-defined, tabular format. Entities within these tables possess multiple attributes, which are metadata defined in a schema. Query languages are employed to access and retrieve information from these structured schemas. Examples of structured data sources include SQL databases and RDF graphs. Datasets like QALD, LC-QuAD, and KGQAn utilize RDF graphs as their data source.

(2) Semi-Structured Data Sources: While semi-structured data sources lack a rigid schema

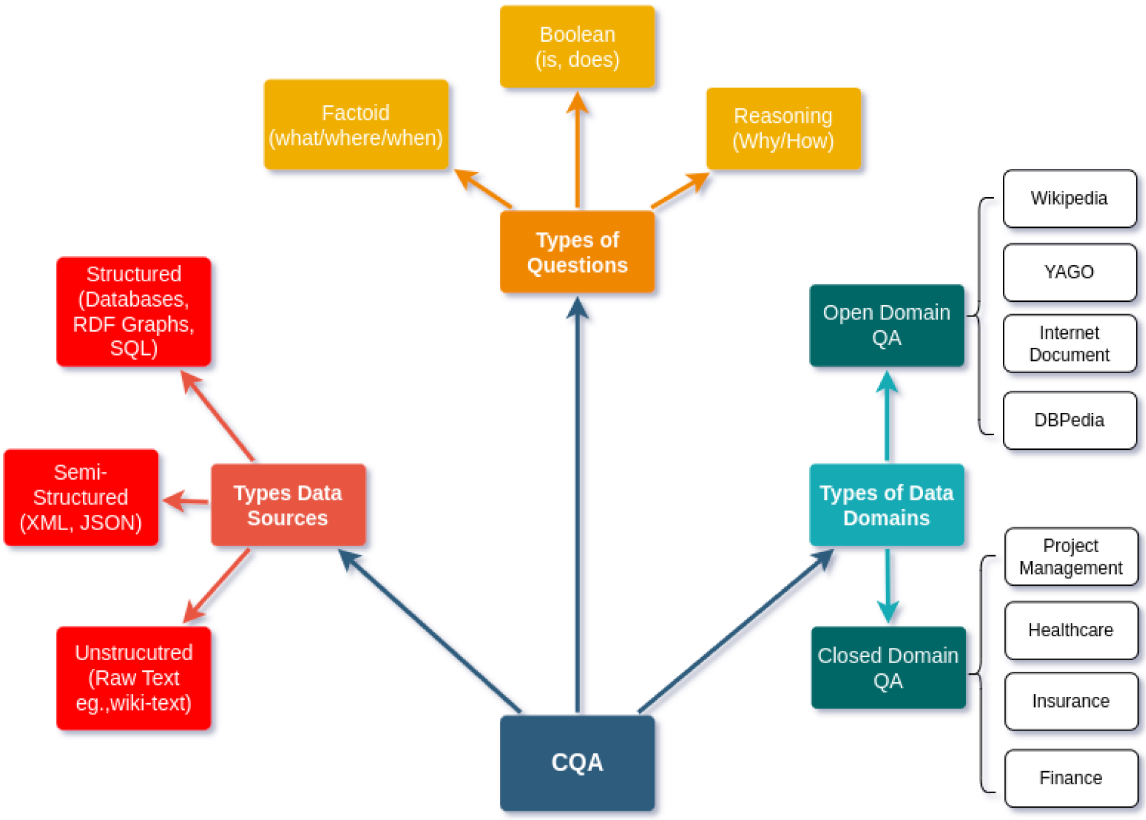

Figure 2.5: Categorization of Conversational Question Answering Systems.

like structured sources, they still exhibit some level of structure. XML and JSON documents are common examples, where entities can have a variable number of attributes.

(3) Unstructured Data Sources: Unstructured data sources lack predefined rules for data storage. They typically consist of unstructured text documents, requiring natural language processing and information retrieval techniques to extract relevant information. Examples include SQuAD, CNN/Daily Mail, and QuAC.

Types of Data Domains Based on the Question Answering application's data domains, user-asked questions can be categorized into two primary types:

(1) Open Domain: Questions that are not restricted to a specific domain. Examples of such domains include general knowledge bases like Wikipedia, YAGO, and DBpedia.

(2) Closed Domain: Questions that are limited to a particular domain, such as healthcare

or finance. Normally, it contains fewer questions due to its narrowed focus on the specific domain compared to Open Domain.

Types of Questions In Question Answering systems, questions are categorized based on the nature of the question, its answer, and the techniques used to answer different types of questions.

(1) Factoid Questions: All WH-questions whose answers result in a short fact phrase or sentence are factoid questions. e.g, "Who is the current President of the United States?". To answer such questions, systems first identify their latent meaning and then look for the answer using deep neural networks or the sentence structure from the text passage. The process of finding an answer to a simple question consists of three basic steps: i) question analysis; ii) relevant documents/knowledge graphs retrieval; and iii) answer extraction.

(2) Boolean Questions: Questions whose answer is in a binary format, i.e., Yes/No or True/False, e.g., "Is the Earth flat?" Since these types of questions do not follow the text span extraction for the answer, they require strong inference from the underlying system.

(3) Reasoning Questions: Questions that begin with "why" or "how" require detailed explanations and are not straightforward. The step to get answers to such questions involves a multi-level understanding of the semantic meaning from the questions, e.g., "Why is the sky blue?"

(4) Listing Questions: Listing questions require a list of facts as an answer, e.g., "List the names of all the former presidents of the United States." Question answering systems consider such questions as a list of factoid questions asked iteratively.

Chapter 3

KGQA RAG Chatbot

3.1 System Overview

3.1.1 Introduction

Knowledge Graph Question Answering (KGQA) systems enable users to explore and retrieve information from knowledge graphs by providing answers to natural language queries (Omar, Dhall, Kalnis, and Mansour (2023)). These systems have seen substantial advancements in recent years, significantly improving their ability to handle a variety of stand-alone or self-contained questions. Such questions are independent of any prior conversational context and are typically free of pronouns or references to earlier entities in the conversation (e.g., "Who is the author of the Harry Potter series?").

However, traditional KGQA systems lack the necessary mechanisms to support conversational interactions. This limitation hinders user engagement and restricts the exploration capabilities of knowledge graphs in scenarios where context-dependent or conversational queries arise. Addressing these challenges requires extending the capabilities of current KGQA systems to facilitate a more dynamic and engaging user experience Omar, Mangukiya, Kalnis, and Mansour (2023).

3.1.2 Motivation

Despite the progress in KGQA systems, their inability to effectively manage conversational question answering (CQA) presents a significant gap. Non-stand-alone or context-dependent questions often include pronouns or references to entities mentioned in previous conversational turns. Understanding and resolving such references are critical for providing accurate answers in conversational settings. Existing systems are not well-equipped to handle this complexity, which limits their applicability in real-world scenarios.

Moreover, current KGQA systems (e.g, KGQAn Omar, Dhall, et al. (2023), EDGQA Hu, Shu, Huang, and Qu (2021)) primarily interact with the knowledge graph by generating a list of top-k SPARQL queries that may contain answers to stand-alone questions. While this approach works for retrieving structured information, it lacks the capability to produce human-friendly answers directly. Conversational systems must address this limitation by incorporating post-processing mechanisms to transform retrieved answers into coherent and natural language responses.

To overcome these challenges, we propose a Conversational KGQA RAG Chatbot system for knowledge graphs. This system extends existing KGQA architectures by integrating Large Language Model(LLM) powered pre-processing and post-processing modules in RAG based KGQA pipeline. The pre-processing module reformulates ambiguous or incomplete questions to ensure clarity and contextual relevance, while the post-processing module generates natural and human-friendly responses from the retrieved answers. Additionally, a conversational history or context memory module is introduced to maintain the dialogue context. This module updates dynamically after each conversational turn and provides contextual references for question reformulation.

By addressing these limitations, our proposed system enhances the user experience and extends the applicability of KGQA systems to conversational and dialogue-driven scenarios, thereby bridging the gap between traditional QA and conversational AI for knowledge graphs.

3.1.3 System Architecture

The proposed system architecture consists of several key components designed to enable conversational question answering over knowledge graphs. These components work together to classify, reformulate, retrieve, and generate human-friendly answers. The overall architecture and its interactions are illustrated in Figure 3.1, providing a visual representation of the system's workflow and data flow. Below, we describe each component in detail.

Question Reformulator

The *Question Reformulator* is responsible for transforming the input question Q_t into a self-contained format. As illustrated in Figure 3.1, the *Question Reformulator* is positioned at the top left of the system architecture diagram, serving as the entry point to the system. It is represented by a yellow box, highlighting its role in preprocessing user queries before passing them to subsequent components. This process is divided into two steps:

(1) **Classification:** The input question Q_t is classified as either a self-contained question or a non-self-contained question. Self-contained questions are complete and do not rely on any prior conversational context, whereas non-self-contained questions may include pronouns, co-references, or ellipsis to previous conversational history.

(2) **Reformulation:** If the question is classified as non-self-contained, the reformulator resolves pronouns and anaphora using the conversational history. This step transforms Q_t into a self-contained question Q'_t, ensuring that it includes all necessary contextual information to retrieve an accurate answer from the system.

The system assumes that the first question in the conversation is always self-contained. This assumption is critical, as without an initial reference entity, the conversation would lack meaningful context and coherence.

Raw Answer Retrieval

Once the reformulated question Q'_t is generated, it is used to retrieve raw answers from the underlying KGQA system. As depicted in the blue box on the top right in Figure 3.1,

the KGQA system is a component responsible for interpreting and answering queries. Most KGQA systems, including KGQAn, accept self-contained questions and return answers in the form of SPARQL queries and extracted triples. These systems can be accessed through an API interface with a well-defined input format, which includes the reformulated question along with configurable parameters such as the maximum number of answers and the target knowledge graph. The KGQA system typically processes the input through multiple stages, including query understanding, entity-relation linking, and filtering. After these steps, the system produces either a ranked list of SPARQL queries or a structured set of nodes and edges representing the extracted knowledge.

LLM Answerer

The Answerer component, depicted in the bottom green box of Figure 3.1, refines the raw KGQA output into human-readable responses suitable for conversational interactions. While the extracted triples from SPARQL queries provide valuable structured information, they must be processed into coherent, context-aware answers. This transformation involves the following steps:

(1) **Input Handling**: The Answerer receives the reformulated question Q'_t and the retrieved triples from the KGQA system as input.

(2) **Prompt Construction**: It constructs a structured prompt incorporating the question, extracted triple labels, and predefined answer guidelines for answer generation.

(3) **Response Generation**: Using this prompt, the Answerer formulates a natural language response while minimizing hallucination by strictly adhering to retrieved knowledge.

(4) **Context Maintenance**: Updates the conversational history with the current question-answer pair and relevant metadata, maintaining context for future interactions

Conversational History

The conversational history, depicted as the center component in Figure 3.1 serves as a dynamic context memory, capturing the flow of dialogue. It plays a crucial role in enabling context-aware question answering by providing essential references for reformulating non-self-contained questions. This component is actively utilized by the Question Reformulator 3.1.3 to reconstruct queries with missing context and updated by the LLM Answerer 3.1.3 after each turn. By continuously evolving with each interaction, it ensures seamless and coherent multi-turn conversations.

3.2 Implementation Details

3.2.1 Question Reformulator

The Question Reformulator operates in two steps: classification and reformulation, both of which are powered by a prompt-based interaction with a LLM. We adopted a prompt template design pattern for better generalization and adaptability to various LLM capabilities. This approach allows for the flexible construction of dynamic prompts by filling predefined template variables at runtime. Furthermore, our design is extensible to advanced prompting techniques such as Chain-of-Thought (CoT) or Self-Consistency, ensuring future scalability. The reformulator processes an input question Q_t in following two steps:

(1) **Classification:** Q_t is classified as either self-contained or non-self-contained using a Question Classification Prompt Template. The classification determines whether the question can be answered without additional context.

(2) **Reformulation:** For non-self-contained questions, the reformulator resolves pronouns, anaphora, and co-references by utilizing the conversation history H_t and constructs a self-contained question Q'_t using a Reformulation Prompt Template.

Algorithm 1 describes the process of question reformulation. Given an input question Q_t, the algorithm attempts to reformulate it into a self-contained question Q'_t by verifying its classification up to three times.

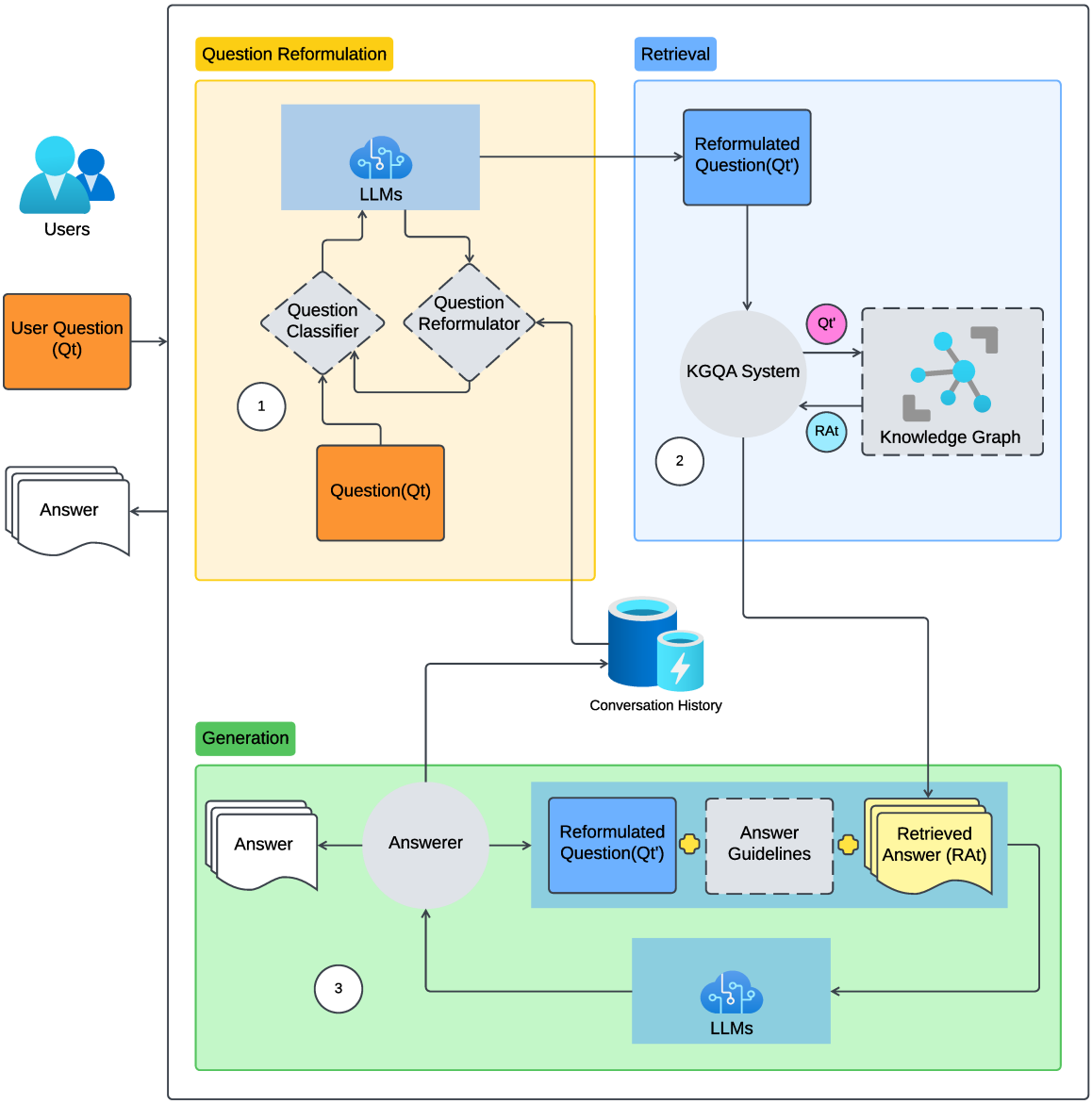

Figure 3.1: KGQA RAG Chatbot System Architecture

First, the algorithm checks whether Q_t is already self-contained by sending it to a language model (LLM) using a classification prompt. If the classification result confirms that the question is self-contained, it is returned as Q'_t without modification.

If the input question is not self-contained, the algorithm proceeds with a reformulation process. It initializes a retry counter and iterates up to three times to generate a reformulated version of the question. During each iteration, the algorithm constructs a reformulation prompt using the input question Q_t and the conversational history H_t. The reformulated question Q'_t is then sent for verification using the same classification prompt.

If any reformulated version of the question is classified as self-contained, the algorithm returns it immediately. Otherwise, it continues the reformulation attempts until the retry limit is reached. If none of the reformulated versions satisfy the verification criteria after three attempts, the algorithm returns a fallback message indicating ambiguity and requesting further clarification from the user. This approach ensures that user queries are reformulated in a way that makes them independent of prior conversation context while minimizing unnecessary retries.

3.2.2 Raw Answer Retrieval

The KGQA system is responsible for retrieving the answers to the reformulated, self-contained question Q'_t provided by the reformulator component. KGQA systems are typically designed to interpret natural language queries and translate them into SPARQL queries to extract information from the target knowledge graph. In our implementation, we integrate the KGQAn system (see Section 3.3.3), which provides an HTTP API endpoint for answering questions. The system processes input questions and retrieves SPARQL queries ranked by their likelihood of answering the given question accurately. The details of the request structure and configuration are described below.

API Request Details

To interact with the KGQAn system, the reformulated self-contained question Q'_t is sent via an HTTP POST request to the KGQAn system's API endpoint. The required fields in

Algorithm 1: Question Reformulation Process with Verification and Retries

Input: Q_t: Input question, H_t: Conversational history
Output: Q'_t: Reformulated self-contained question or fallback message

Initialize `classification_prompt` using Classification Prompt Template;
Fill `classification_prompt` with Q_t;
Send `classification_prompt` to LLM and get classification result;
if *`classification result` is "self-contained"* **then**
    Set $Q'_t \leftarrow Q_t$;
    return Q'_t;
Set `retry_count` $\leftarrow 0$;
while *`retry_count`* < 3 **do**
    Increment `retry_count`;
    Initialize `reformulation_prompt` using Reformulation Prompt Template;
    Fill `reformulation_prompt` with Q_t and H_t;
    Send `reformulation_prompt` to LLM and get reformulated question Q'_t;
    Initialize `verification_prompt` using Classification Prompt Template;
    Fill `verification_prompt` with Q'_t;
    Send `verification_prompt` to LLM and get verification result;
    if *`verification result` is "self-contained"* **then**
        return Q'_t;
return *"Too much ambiguity, can you clarify more?"*;

the request body are as follows:

- **question:** The reformulated, self-contained question Q'_t in natural language.
- **knowledge_graph:** The specific knowledge graph on which the query is intended to operate. This allows the system to select the relevant dataset.
- **max_answers:** The maximum number of SPARQL queries to return, ordered by descending probability of relevance.

The request is constructed as a JSON Body in the following format:

```
{
    "question": "What is the capital of France?",
    "knowledge_graph": "yago",
    "max_answers": 3
}
```

API Request Workflow

The reformulated question Q'_t is packaged into the request body along with other required parameters and sent to the API endpoint. The server processes this request and generates a list of SPARQL queries corresponding to the input question. The HTTP server expects the following:

- A valid and meaningful self-contained natural language question (Q'_t).
- A specified knowledge graph to be used by KGQAn for the query scope.
- A numerical limit (`max_answers`) for the number of queries returned.

The response from the KGQAn system contains the retrieved SPARQL queries and/or extracted triples, which are passed to the next component for further processing.

3.2.3 LLM Answerer

The final step in the pipeline is constructing a human-readable answer from the retrieved list of SPARQL queries. This process is crucial because the raw representation of the answer in the form of SPARQL queries or triples is not easily interpretable by end-users. The answer builder addresses this by performing two key operations: extracting and labeling triples from SPARQL queries, and generating a natural language response.

Step 1: Resolving Labels from SPARQL Queries

The Knowledge Graph Question Answering (KGQAn) system provides a list of SPARQL queries as its output, each representing a possible answer. However, these SPARQL queries often include URIs (Uniform Resource Identifiers) or triples containing predicates and objects in URL form, which are not user-friendly. To resolve this, the following sub-steps are performed:

- **Extract Triples from SPARQL Queries:** Each SPARQL query is executed against the knowledge graph's SPARQL endpoint to retrieve a list of triples that represent the answer.

- **Label Metadata Resolution:** For each URI in the retrieved triples, a label metadata resolution query is executed. This involves querying the knowledge graph for English language labels or annotations associated with the URIs. The resulting labels provide a more interpretable representation of the entities and predicates.

By the end of this step, the system has a set of labeled triples in English that are ready to be used for generating a human-friendly answer.

Step 2: Generating the Human-Friendly Answer

Once the triples are labeled, the system utilizes a prompt-based approach to generate the final answer in natural language. A Large Language Model (LLM) is leveraged for this purpose. The labelled triples and the original user query are provided as inputs to the LLM using a predefined prompt template. The template ensures the model focuses on the relevant information and produces coherent answers. The prompt template used is as follows:

Prompt Template : Answer Generation Prompt

You are an expert assistant that converts structured KGQA results into human-readable answers.

Given the question and the raw answer list, follow these steps:

1. Extract values from the ”value” fields in the answer list.
2. Format values appropriately (e.g., dates, URIs).
3. Synthesize a concise, grammatically correct answer.
4. **Do not add information not present in the answer list**.

Question: {question}

Answer List: {answers}

Response:

The placeholders in the template are filled dynamically:

- {`question`}: The original question asked by the user.

- {`answers`}: The list of triples with their labels resolved in English.

3.2.4 Conversation History/Memory

The system maintains a conversation history stored as a temporary list in memory. This history plays a crucial role in managing the context for conversational question answering, especially when resolving non-self-contained questions.

Structure of Conversation History

The conversation history is implemented as a list of tuples, with each tuple representing a single turn in the conversational flow. Each tuple contains the following fields:

- **Original Query:** The exact question asked by the user in natural language.
- **Reformulated Query:** The transformed self-contained question generated by the reformulator. If the original question is already self-contained, this field will match the original query.
- **Final Answer:** The human-readable answer formed by the system, utilizing the LLM and the labeled triples from the KGQA system.
- **Label Triples:** The intermediate list of labeled triples retrieved from the KGQA system, which were used to construct the final answer.

The conversation history serves as a dynamic context memory, capturing the flow of dialogue and ensuring continuity throughout interactions. At the end of each conversational turn, before returning the final answer to the user, the system appends a new tuple for the current turn, keeping the history up-to-date with the latest interactions. This updated history is then utilized by the *Question Reformulator*, particularly when processing non-self-contained questions. By leveraging previously stored context, the reformulator resolves pronouns, anaphora, or co-references to construct a well-formed query. The process of maintaining and utilizing conversation history is formalized in Algorithm 1, ensuring robust context-aware question answering.

3.3 Related Work

3.3.1 CONVEX

CONVEX (Christmann, Saha Roy, Abujabal, Singh, and Weikum (2019)) is an unsupervised method for answering incomplete questions over a knowledge graph (KG) by maintaining conversational context through previously seen entities and predicates. It automatically infers missing or ambiguous information in follow-up questions using a graph exploration algorithm that expands a frontier to identify candidate answers. To evaluate CONVEX, the authors introduced ConvQuestions, a crowdsourced benchmark with 11,200 distinct conversations spanning five domains.

A key system limitation of this approach arises from its reliance on an expanding context subgraph, which can lead to a combinatorial explosion of candidate nodes as the conversation context broadens. Although CONVEX mitigates this critical issue by judiciously controlling subgraph expansion through a combination of look-ahead, weighting, and pruning techniques, these measures only partially address the inherent challenge. As a result, the method still faces theoretical and computational constraints when dealing with ambiguous or incomplete queries that span multiple topics, impacting both efficiency and precision.

3.3.2 CONVINSE

CONVINSE (Christmann, Saha Roy, and Weikum (2022)) is an end-to-end conversational question answering pipeline over heterogeneous sources. It operates in three stages: (i) learning a structured representation of the incoming question and its conversational context, (ii) leveraging this representation to retrieve relevant evidence from knowledge bases, text, and tables, and (iii) using a fusion-in-decoder model to generate the final answer. It is limited to adapting a new domain or dataset, which requires the training of the model.

3.3.3 KGQAn

KGQAn (Omar, Dhall, et al. (2023)) is a universal question-answering system that does not require customization for each target KG. Instead of relying on curated rules, it

frames question understanding as a text generation problem, converting questions into an intermediate abstract representation via a neural sequence-to-sequence model. At query time, a just-in-time linker maps the abstract representation to a SPARQL query, retrieving answers from the KG using only publicly available APIs and RDF store indices, without any pre-processing.

3.3.4 Chatty-Gen

Chatty-Gen (Omar, Mangukiya, and Mansour (2025)) is a retrieval-augmented generation (RAG) platform for automatically generating high-quality dialogue benchmarks tailored to specific domains using KGs. It follows a four-stage pipeline: (i) extracting and summarizing relevant subgraphs, (ii) generating questions linked to specific KG triples, (iii) using zero-shot learning to generate SPARQL queries for retrieving answers, and (iv) constructing coherent dialogues. The final dataset consists of dialogue sequences, independent questions, and corresponding SPARQL queries.

3.4 Evaluation and Experiments

3.4.1 Dialogue Generation and Curation Process

To assess our KGQA RAG Chatbot pipeline, We conducted experiments at multiple levels, including individual module evaluations, prompt effectiveness, and an end-to-end system evaluation. This required an evaluation dialogue benchmark aligned with our system, for which we considered two options:

- **ConvQuestions** 3.3.1: A crowdsourced dialogue benchmark containing 11,200 distinct conversations from five domains, where all questions are formed using Wikidata.
- **Chatty-Gen** 3.3.4: A novel multi-stage retrieval-augmented generation platform that generates high-quality dialogue benchmarks tailored to specific domains and knowledge graphs (KGs).

We chose the Chatty-Gen platform over ConvQuestions for the following reasons:

(1) **Alignment with our KGQA system**: Unlike ConvQuestions, which is tied to Wikidata, Chatty-Gen supports multiple KGs, including YAGO, DBLP, DBpedia, and MAG.

(2) **Comparable question quality**: Chatty-Gen's LLM-generated dialogues are indistinguishable from human-annotated ones in 70% of cases, demonstrating its effectiveness as a benchmark generator (Omar et al. (2025)).

Chatty-Gen (see 3.3.4) is a flexible framework that allows users to generate customized benchmarks using various parameters. Chatty-Gen can create dialogue benchmarks from seed node URLs, supporting two modes of entity selection: *automatic sampling* and *manual entity selection*. In the latter mode, users can supply a text file containing seed entity URLs from the KG for generation. The *automatic sampling* method leverages the weighted distribution of node types in the KG.

Here are the key set of configurable parameters for Chatty-Gen:

- Knowledge graph source and SPARQL endpoint.
- Number of dialogues and questions per dialogue.
- Language model endpoint.
- Generation approach (*single-step* or *multi-step*).
- Seed Nodes (Optional)

configuration for benchmark generation

```
{
    "dbpedia_endpoint": "<add_endpoint_version>",
    "dialogue_size": 5,
    "pipeline_type": "multi-step",
    "approach": "subgraph-summarized",
    "language_model": "gpt-4o",
    "seed_nodes_file": "sample_seed.txt"
}
```

Below is the generated dialogue benchmark's json file, it contains the seed, dialogue questions and original questions along with sparql answer queries for each dialogues.

```
// JSON Scema of Generated Dialogue
{
    "data": [
        {
            "seed_entity": "<url>",
            "seed_label": "<entity_label>",
            "dialogue": ["transformed_questions"],
            "original": ["original_questions"],
            "queries": ["sparql_queries"],
            "triples": [("subject", "predicate", "object")]
        },
        ...
    ]
}

// Example Dialogue
{
    "seed_entity": "http://dbpedia.org/resource/Friends",
    "seed_label": "Friends",
```

```
    "dialogue": [
        "How many episodes does Friends have?",
        "Who created it?",
        ...
    ],
    "original": [
        "How many episodes does Friends have?",
        "Who created Friends?",
        ...
    ],
    "queries": [
        "SELECT ?count WHERE { <http://dbpedia.org/resource/Friends>
           <http://dbpedia.org/ontology/numberOfEpisodes> ?count }"
           ,
        "SELECT ?creator WHERE { <http://dbpedia.org/resource/
           Friends> <http://dbpedia.org/ontology/creator> ?creator.
           }",
        ...
    ],
    "triples": [
        [
            [
                "http://dbpedia.org/resource/Friends",
                "http://dbpedia.org/ontology/numberOfEpisodes",
                ""
            ]
        ], ...
    ]
}
```

3.4.2 Seed Selection Process for Comparison

To compare our KGQA-RAG system with the baseline system CONVINSE across two knowledge graphs (DBpedia and YAGO), we randomly selected 20 seed entities from the ConvQuestions 3.3.1 benchmark. This manual seed selection was used instead of automatic to ensure we evaluate both systems fairly, as baseline system CONVINSE only works Wikidata, whereas our system works with KGs (YAGO, DBPedia, DBLP, MAG). Each chosen seed was required to meet the following criteria:

(1) The seed must have associated URL predicates linking it to both DBpedia and YAGO.

(2) The set of facts (i.e., properties and relations) for the seed in Wikidata must also be present in DBpedia and YAGO.

To perform this selection, we followed a two-step process:

Step 1: Mapping via SPARQL Queries. For each seed, we first identified its corresponding entries in DBpedia and YAGO using SPARQL queries. For example, the mapping for DBpedia was obtained with the following query:

DBpedia Mapping SPARQL Query

```
PREFIX owl: <http://www.w3.org/2002/07/owl#>
SELECT ?dbpedia
WHERE {
    ?dbpedia owl:sameAs <http://www.wikidata.org/entity/Q937>.
}
```

Similarly, the mapping for YAGO was identified using this SPARQL query:

YAGO Mapping SPARQL Query

```
PREFIX owl: <http://www.w3.org/2002/07/owl#>
SELECT ?yago
WHERE {
    ?yago owl:sameAs <http://www.wikidata.org/entity/Q937>.
}
```

Step 2: Fact Verification. After obtaining the mappings, we manually verified that all facts associated with each seed in Wikidata were also present in the corresponding DBpedia and YAGO entries. This ensured consistency across the knowledge graphs for each seed entity. Using the verified seed nodes, we then generated an evaluation set of dialogues by employing GPT-4o with a dialogue size of 5 turns.

3.4.3 Question Reformulator Experiments

The question reformulation process in the KGQA-RAG pipeline involves a two-step, prompt-based approach that transforms user queries into formats that are optimally processed by the underlying KGQA system. To evaluate the robustness of prompt wording and its impact on the later stages of the pipeline, we conducted several experiments focused on prompt engineering.

User Question Classification Classifying user questions as *independent/self-contained* or *dependent/non-self-contained* is critical to system accuracy. Misclassification can lead to errors, as dependent questions require reformulation for accurate answers.

Prompt Template : Classification Prompt 1

Classify a given question as either '**independent**' or '**dependent**.' In this context, 'independent' questions are those that can be understood and answered without needing additional context or information, while 'dependent' questions require additional context or information to be answered. It's crucial for the model to identify ambiguity and resolve unknowns to determine the classification correctly.

Question: {question}

Classification:

Question: When was Granada District founded?

Classification: **dependent** // True Label : independent

Prompt Template : Classification Prompt 2

Classify a given question as either '**self-contained**' or '**non-self-contained**' while considering the context of resolving pronouns and references.

In this task, 'self-contained' questions are those that can be understood and answered without needing additional context or information, even when they refer to entities or concepts mentioned in prior conversation. 'Non-self-contained' questions are those that, due to the presence of pronouns or references to prior conversation, require additional context or information to be answered correctly.

Question: {question}

Classification:

Question: When was Granada District founded?

Classification: **self-contained** // True Label : self-contained

To evaluate prompt variations, we used the set of 40 questions from the evaluation benchmark 3.4.1 (20 original and 20 dialogue-based), each with ground truth labels. As shown in Table 3.1, Classification Prompt 1—using the terms "independent" vs. "dependent"—achieved an accuracy of 60%, while Classification Prompt 2—using "self-contained" vs. "non-self-contained"—attained a 95% accuracy. These results suggest that the choice of terminology in the prompt templates can significantly improve classification performance and reduce hallucinations.

Prompt Template	**Accuracy**	**Correct**	**Total**	**Hallucinations**
Classification Prompt 1	60%	24	40	0
Classification Prompt 2	95%	38	40	1

Table 3.1: Evaluation results for user question classification using "independent vs. dependent" (Prompt 1) and "self-contained vs. non-self-contained" (Prompt 2) terminologies.

Question Reformulation A core challenge in question reformulation for non-self-contained queries is accurately resolving pronouns and coreferences by leveraging the relevant conversational context. Our approach addresses this by integrating the active focal entity. To validate the quality of our LLM-driven reformulation method, we employed BLEU and ROUGE metrics, assessing both the similarity to human-annotated reformulations and the accuracy of entity integration. We chose two established question rewriting datasets, ConvQuestions 3.3.1 and QReCC Anantha et al. (2021), known for their diversity in dialogue structure. ConvQuestions represents dialogues with short question sets, whereas QReCC provides longer, multi-turn conversations (up to 10+ turns). This diversity allowed us to investigate how conversation length impacts coreference resolution. Our evaluation set included 20 dialogues from ConvQuestions (100 questions) and 20 from QReCC (188 questions). By comparing different variations of context history during reformulation within this set, we aimed to specifically measure the effect of complex and extended conversation history on the accuracy of coreference resolution in reformulated questions.

Here are used two variations of representing Context History:

(1) **Immediate Context Approach:** Using only the previous self-contained question-answer pair to reformulate the next non-self-contained question.

(2) **Full History Approach:** Incorporating all previous question-answer pairs as context for the reformulation.

Table 3.2 summarizes the evaluation results for both prompt templates.

Results indicate that the Full History Approach (represented by Question Reformulation Prompt 2) consistently outperformed the Immediate Context Approach (Prompt 1). The lower performance of the first approach is attributed to extra stop words introduced by the LLM during reformulation, likely due to bias or error propagation in long-depth conversations. For example, in the mlqrcc dataset:

- **Depth 4:** The actual question "Are boer goats good for meat?" was reformulated as "Can you provide more information about Boer goats?"

- **Depth 6:** The actual question "What is the best goat for fiber production?" was reformulated as "What breed of goat is generally considered the best for fiber production?"

Prompt Template : Reformulation Prompt 1

Rephrase the follow-up question to be self-contained using ONLY given context items.

1. Previous Reformulated Question: Previous turn self-contained question

2. Previous Answer: Answer to previous turn self-contained question

Follow these rules:

- Resolve pronouns/anaphora (it, they, he, she, them) using the previous QA pair

- Preserve the original question's intent and focus completely

- Never introduce new information or assumptions

- Maintain the focal entity from previous context unless explicitly changed

- Keep questions concise but fully unambiguous

Previous Reformulated Question: {previous_reformulated}

Previous Answer: {previous_answer}

Follow-up Question: {current_question}

Rephrased Self-Contained Question:

Prompt Template : Reformulation Prompt 2

Rephrase the given follow-up question to make it self-contained by resolving any pronouns, anaphora, or coreferences using the provided context. Do not add, modify, or extend the meaning of the original question. The rephrased question must strictly retain the original intent and focus.

Conversation Context:

{chat_history}

Follow-up Question: {question}

Rephrased Self-Contained Question:

Prompt Template	Dataset	BLEU	ROUGE-1	ROUGE-2	ROUGE-L
Question Reformulation Prompt 1	convex	0.302	0.619	0.456	0.598
	mlqrcc	0.302	0.669	0.497	0.643
Question Reformulation Prompt 2	convex	0.384	0.714	0.551	0.678
	mlqrcc	0.499	0.822	0.700	0.809

Table 3.2: Evaluation results for question reformulation prompts. Metrics (BLEU, ROUGE-1, ROUGE-2, ROUGE-L) for two datasets (convex and mlqrcc) indicate that Prompt 2 yields higher similarity to the ground truth.

3.4.4 Overall System Evaluation

Experimental Setup and Results To assess the performance of our developed KGQA-RAG system, we conducted experiments using a custom evaluation dialogue benchmark (see Section 3.4.1 for details on dialogue generation). In this benchmark, each dialogue consists of a SPARQL query that returns a list of bindings for the unknown variable in the question. For every query, the underlying KGQA system (KGQAn) produces multiple ranked answers. We compute performance using the metrics Precision at 1 (P@1), Mean Reciprocal Rank (MRR), and Hit@5.

The evaluation involves comparing three systems:

(1) **KGQA-RAG:** Our developed system, which integrates conversational context while leveraging the KGQA component.

(2) **KGQAn:** The baseline KGQA system, representing the theoretical performance limit.

(3) **CONVINSE:** A second baseline conversational question answering system over heterogeneous sources. Due to resource constraints and reproducibility issues with local dependencies, we used the hosted version available at https://convinse.mpi-inf.mpg.de/. This version is accessed via an API call with a request body structured as follows:

```
body = {
```

```
        "question": question,
        "history_questions": history_questions_list or [],
        "history_answers": history_answers_list or []
    }
```

To calculate the evaluation metrics, we used the implementation provided by the CONVINSE framework. This implementation compares the list of answer label values produced by each system against the list of ground truth labels generated for each dialogue in the benchmark. The post-processing of each retrieved binding involves normalizing literal-typed values and retrieving the `rdf:label` (in English) for URI types via an additional SPARQL request.

System	DBPedia			YAGO		
	P@1	MRR	Hit@5	P@1	MRR	Hit@5
KGQAn	0.2765	0.2787	0.2872	0.5813	0.5918	0.6046
KGQA RAG Chatbot	0.2447	0.2589	0.2766	0.5262	0.5201	0.5279
CONVINSE	0.2447	0.2447	0.2447	0.3372	0.3372	0.3372

Table 3.3: Overall performance comparison of KGQA system variants and CONVINSE on DBpedia and YAGO benchmarks.

Table 3.3 summarizes the experimental results on two benchmarks, DBpedia and YAGO. As seen in the table, while the KGQA-RAG system performs slightly below the underlying KGQAn baseline, it significantly outperforms CONVINSE across all three metrics for YAGO.

Chapter 4

JIRA RAG Chatbot

4.1 System Overview

4.1.1 Introduction

JIRA (Atlassian (2025a)) is a widely used project management tool that allows organizations to store critical data related to tasks, projects, and workflows. It has become a vital part of modern companies for managing and tracking initiatives, features, epics, and associated tasks. However, as the volume of data in JIRA grows, navigating and analyzing it becomes increasingly challenging, especially for users unfamiliar with JQL (JIRA Query Language)(Atlassian (2025b)). This challenge intensifies as organizations grow, making manual tracking and review inefficient and increasing the risk of errors and oversights.

The JIRA RAG Chatbot System addresses this problem by leveraging large language models (LLMs) to assist users in navigating, querying, and understanding JIRA data effectively. The system aims to improve productivity and decision-making by offering precise, context-aware responses to queries about JIRA issues, summaries, and linked issue relationships.

4.1.2 Motivation

The motivation behind the JIRA RAG Chatbot System stems from the limitations of naive approaches to JIRA data retrieval and analysis. Traditionally, a simple approach

involves retrieving data, serializing it, and using it directly for question-answering (QA). While straightforward, this method is less scalable nor accurate due to several drawbacks:

- Context Length Problems: Large language models (LLMs) struggle with extensive context lengths, leading to incomplete or inaccurate responses.
- Hallucination Risks: Unconstrained access to data increases the likelihood of generating irrelevant or fabricated answers.
- Inefficient Token Usage: Processing excessive context wastes computational resources.
- Continuously Evolving Data: JIRA data is dynamic, requiring a system that can handle frequent updates efficiently.

To address these challenges, we developed an advanced, modular Retrieval-Augmented Generation (RAG) Conversational QA system with an intent-based query understanding module. This module performs a pre-retrieval step to generate precise input for the retrieval process. The retrieval mechanism then selects relevant, partial data to minimize unnecessary context. This modular approach improves accuracy, optimizes token usage, and reduces hallucinations. The system provides significant benefits, especially in two primary use cases:

(1) Higher-Management Assistance: For individuals in senior organizational roles, the system can:

   (a) Summarize multiple JIRA issues efficiently.

   (b) Analyze and derive insights from linked issues (e.g., epics and features with associated tasks).

   (c) Support critical tasks such as sprint planning and identifying potential resource dependencies or deadlocks between teams.

(2) Information Bot (Wiki Bot): For team members or stakeholders seeking information, the chatbot serves as an efficient query tool to:

   (a) Navigate and retrieve information on specific JIRA issues.

(b) Answer detailed questions such as issue status, priority, summaries over specific periods, domain, responsible teams, deliveries, and deadlines.

4.1.3 System Architecture

The proposed system architecture is designed to enable conversational question answering over structured project management data, particularly within a JIRA data. This architecture is composed of several core components that work together to interpret user intent, retrieve relevant information, and generate contextually accurate responses. The system architecture comprises three main components:

(1) **Natural Language Query (NLQ) Parsing:** Transforms user input into a structured representation by classifying intent, extracting JIRA Issue IDs, and resolving contextual dependencies.

(2) **Context Retrieval:** Dynamically retrieves relevant JIRA Issue data using structured query generation, leveraging conversation history and JIRA metadata to ensure accurate results.

(3) **Answer Generation:** Synthesizes human-readable responses by grounding them in retrieved data, ensuring consistency, coherence, and factual correctness.

The overall architecture and its interactions are illustrated in Figure 4.1, providing a visual representation of the system's workflow and data flow. A key aspect of this architecture is its ability to maintain conversation continuity, allowing users to ask follow-up questions without repeating context. By integrating a structured conversation history module, the system ensures that responses remain relevant even in multi-turn dialogues. Additionally, by leveraging both graph-based retrieval mechanisms and JQL query execution, the architecture balances efficiency with real-time data accuracy. Each component is designed to be modular and operate independently, allowing for easier development and maintenance, while their seamless integration ensures a cohesive end-to-end conversational experience. The following sections provide a deeper dive into each stage of the pipeline, detailing how the system processes and answers user queries efficiently.

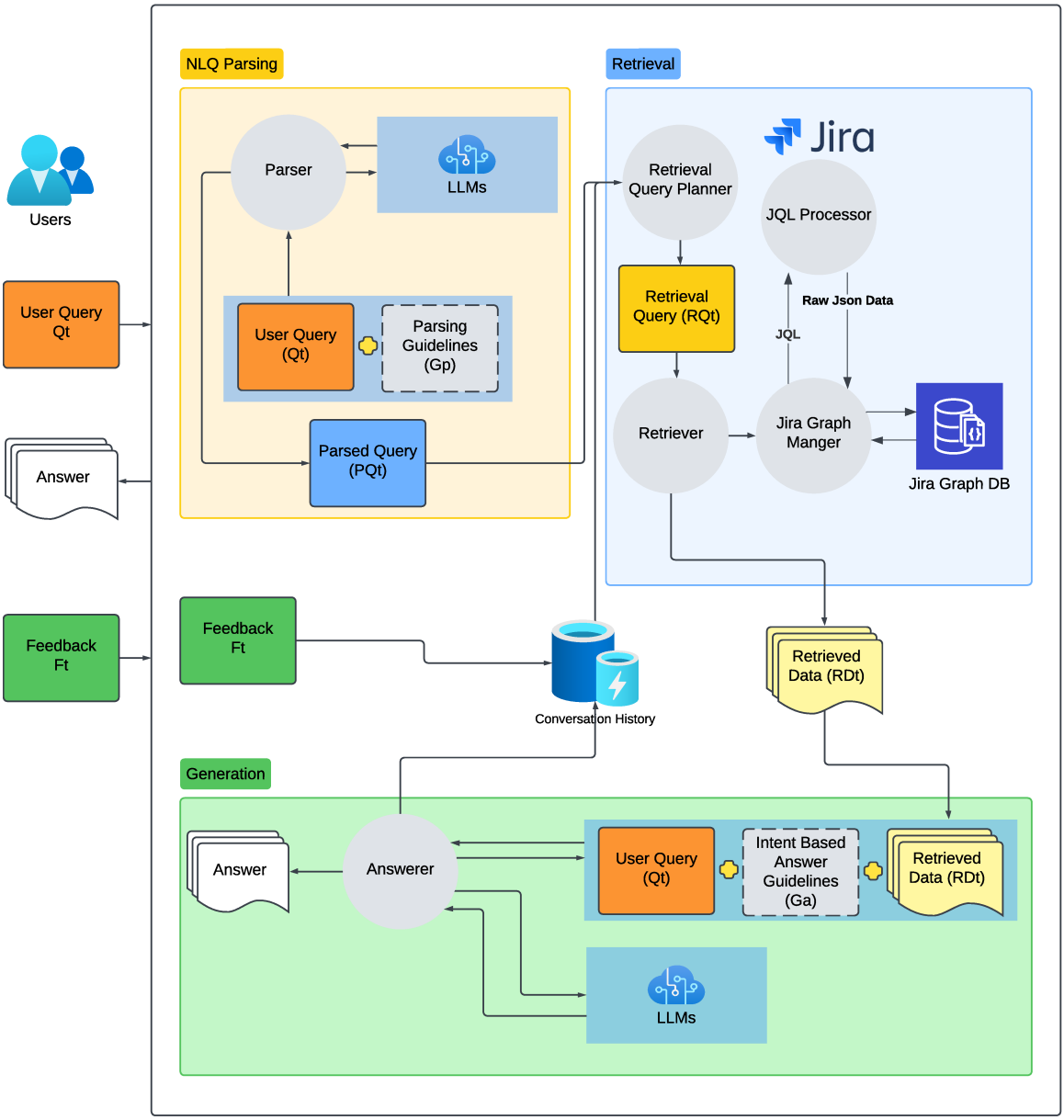

Figure 4.1: JIRA RAG Chatbot System Architecture

Natural Language Query (NLQ) Parsing

The NLQ Parsing component serves as the system's entry point, transforming raw user queries into structured representations suitable for retrieval and response generation. Since users often express queries in an unstructured and ambiguous manner, this component is responsible for interpreting intent, extracting JIRA Issue IDs, and resolving contextual dependencies before passing the query forward for processing. To achieve this, As depicted in the yellow box on the top left of Figure 4.1, the NLQ Parser categorizes queries into predefined intent types, ensuring that downstream components receive a structured and predictable input format. The primary query types include:

- **Direct Metadata Questions:** Queries seeking specific metadata properties of an issue (e.g., "What is the priority of ISSUE-123?").
- **Reasoning Questions:** Requests for summarized insights or analytics (e.g., "Summarize the current status of ISSUE-456.").
- **Changelog Questions:** Queries about historical changes within a specific timeframe (e.g., "What updates were made to ISSUE-789 last week?").
- **Structure-Related Questions:** Inquiries regarding relationships, dependencies, or hierarchical issue structures (e.g., "Which features are linked to INITIATIVE-001?").

Since user queries may lack explicit references to issue keys or relevant metadata, the NLQ Parser incorporates conversation history to resolve ambiguities. For example, in a multi-turn conversation, if a user asks "What is its status?" after a prior query referencing ISSUE-123, the system infers the missing JIRA Issue Id based on stored context. This ensures that extracted intents are mapped to a well-defined schema before moving to the next phase. The process can be formalized as:

$$P_{Qt} = \text{LLM}(Q_t + G_p)$$

where Q_t is the user's natural language query, and G_p is a predefined guideline that standardizes intent classification and JIRA Issue IDs extraction. By enforcing structured JSON schema adherence and integrating contextual resolution mechanisms, this module ensures that the system can handle both explicit and implicit queries while maintaining accuracy and efficiency. The parsed representation serves as a foundation for retrieval planning and structured query generation, ensuring that subsequent components receive well-formed inputs.

Context Retrieval

The Context Retrieval component plays a critical role in ensuring that the system fetches relevant information to accurately respond to user queries. Since queries may reference specific issues, require historical insights, or depend on contextual factors from prior interactions, this component is responsible for dynamically identifying and retrieving the necessary data while minimizing redundant lookups. To achieve this, As depicted in the blue box on the top right in Figure 4.1, Context Retrieval leverages structured retrieval planning and query execution to fetch relevant issue details from JIRA. It ensures that responses are generated based on the most up-to-date and comprehensive data available while optimizing retrieval efficiency.

Retrieval Query Planner The Retrieval Query Planner (RQP) serves as the core mechanism for constructing structured retrieval queries based on the parsed intent and extracted entities from the NLQ Parsing stage. Its primary function is to translate natural language intent into a well-defined retrieval query that adheres to the JIRA Query Language (JQL) syntax. The planner generates a structured representation called the Retrieval Query (R_{Qt}), which consists of:

- **JQL Query Formulation** – A structured query for fetching relevant issue data from JIRA.
- **Sub-schema Field Selection** – A specification of the exact fields to retrieve (e.g.,

"a pre-defined subset of the JIRA issue data schema"), ensuring that only necessary data is fetched.

For instance, consider the user query: "What are the features with high priority under ISSUE-XYZ?", the Retrieval Query Planner generates a JQL query that retrieves all feature-type issues under ISSUE-XYZ and selects relevant fields such as:

```
{
  "jql": "project = XYZ AND type = Feature AND priority = High",
  "select_fields": ["name", "priority", "status"]
}
```

One of the key challenges in retrieval planning is handling queries without explicit issue references. If a user omits an issue key—such as in a follow-up query ("List the remaining tasks")—the planner infers the missing context by referencing the conversation history. This enables the system to maintain continuity in multi-turn interactions and retrieve relevant data even when queries are phrased ambiguously. The retrieval planning process can be formalized as:

$$R_{Qt} = \text{RQP}(P_{Qt} + \text{ConvH}_{t-1})$$

where:

- P_{Qt} represents the parsed query output from the NLQ Parser.
- ConvH_{t-1} is the structured conversation history at the previous turn.
- R_{Qt} is the final structured retrieval query.

Retriever The Retriever component is responsible for executing the retrieval query (R_{Qt}) to fetch the required data from JIRA. It operates by:

- Retrieving raw data from the **JIRA Graph Manager**, by utilizing the JQL query in received R_{Qt}.
- Post processing the retrieved raw data from JIRA, by applying sub-schema selection defined in the R_{Qt}.

The final output is structured **Retrieved Data** (R_{Dt}), formatted as a JSON response, ensuring that the extracted information is structured and ready for downstream processing.

JIRA Graph Manager The JIRA Graph Manager provides CRUD operations on a database that stores processed JIRA issue data. Instead of always querying JIRA's APIs, this component allows for optimized access by maintaining a graph-structured representation of JIRA issues, their relationships, and metadata. It processes JQL queries received from the Retriever and either serves data from the database or routes the query to the JQL Processor when when the requested data is not available in the database or needs to be updated. This caching mechanism helps reduce API call latency and enhances system performance.

JQL Processor The JQL Processor serves as an interface to JIRA's server APIs, responsible for executing JQL queries. It is designed for high-efficiency data retrieval using:

- **Asynchronous request handling** – Optimizes performance by executing multiple queries in parallel.
- **Batch processing** – Groups related JQL queries into batch requests, reducing API overhead.
- **Error handling and retry mechanisms** – Ensures resilience against transient failures in API responses.

The combined retrieval operation can be formalized as:

$$R_{Dt} = \text{JDR}(R_{Qt})$$

where JDR represents the unified execution process JIRA Data Retriever involving the Retriever, JIRA Graph Manager, and JQL Processor. This ensures that the system can efficiently retrieve, process, and structure relevant JIRA issue data while optimizing for both speed and accuracy.

Answer Generation

The Answer Generation component, as depicted in the green box at the bottom in Figure 4.1, is responsible for synthesizing a coherent and contextually relevant response to the user query (Q_t). It leverages the retrieved data (R_{Dt}) obtained from the JDR retrieval pipeline and intent-based answer guidelines (G_a), which defines response formatting, detail prioritization, and phrasing conventions. To ensure the generated answer is accurate and aligned with JIRA issue structures, an LLM prompt is dynamically constructed using Q_t, R_{Dt} and G_a.

Answer Generation Example

User query (Q_t): *What is the status of high-priority features under 'ISSUE-XYZ'?*

Retrieved Data (R_{Dt}):

ID: ISSUE-ABC, User Authentication – Priority: High, Status: In Progress

ID: ISSUE-PQR, Payment Gateway Integration – Priority: High, Status: Blocked

Guidelines (G_a): Summarize the retrieved information concisely.

Generated Response:

There are two high-priority features under 'ISSUE-XYZ'. 'User Authentication' is currently 'In Progress', while 'Payment Gateway Integration' is 'Blocked'. Let me know if you need further details.

This response is then:

- **Returned to the user** as the final answer.
- **Appended to the conversation history** to maintain context for follow-up queries.

By structuring responses with both retrieved data and answer guidelines, the Answer Generation component ensures that responses are precise, informative, and easy to understand.

4.2 Design and Implementation

4.2.1 NLQ Parsing

The NLQ Parsing component is designed to process user-provided natural language queries (NLQ) and transform them into a structured format suitable for downstream processing. This transformation is achieved through the utilization of a pre-defined natural language understanding (NLU) prompt template, which incorporates parsing guidelines, intent definitions, and examples from the intent map to understand the user's query and intent.

Intent Map The Intent Map defines the available intents for user queries and provides associated query examples, a sub-schema selector function, and answer guidelines. It is defined as python dataclass, which is a convenient way to define classes primarily for data storage and used by different components in the system as follows. The **Parser** consults the Intent Map to interpret and identify the user's intent by matching query patterns with the defined examples and intent definitions. The **Retriever** leverages the Intent Map to guide the post-processing of the retrieved JIRA data, using the sub-schema selector function to ensure the correct data subset is selected. Finally, the **Answerer** utilizes the answer guidelines provided in the Intent Map to construct and format the final response in a manner consistent with the predefined instructions. The Intent Map includes the following properties:

- `intent-name:` The name of the intent.
- `intent-description:` The description of the intent.
- `examples:` A list of query examples associated with the intent.
- `sub-schema-selector:` The post-processing function.
- `answer-guideline:` Instructions for forming the answer.

NLU Prompt Template The NLU prompt template is a carefully crafted textual prompt that guides the parsing process. It defines the *Parsing Guidelines (Gp)*, and ensures consistency in extracting structured information from NLQs. The prompt template is designed to handle queries within the context of JIRA issue management, with explicit placeholders for user queries and descriptions of intents. It also incorporates examples for consistency across various queries. The structure of the prompt template includes:

- **Intent Definitions:** Enumerated set of predefined intent types, each representing a distinct class of query, such as *METADATA*, *CHILDREN*, and *RELATED*. All the intents are defined in the IntentMap 4.2.1, which are used to fill the prompt dynamically described in algorithm 2.
- **Issue Key Recognition:** A strict format (e.g., `ISSUE-abc`) to identify issue keys within the query, enforced through a regular expression pattern.
- **Parsing Guidelines (Gp):** Detailed instructions embedded within the prompt to guide the identification and extraction of intents and associated information.
- **Examples:** Query examples for each intent type to improve consistency and accuracy.

Parsed Query Representation The output of the NLQ Parsing component is referred to as the *Parsed Query (PQt)*. It encapsulates the structured representation of the user's intent and associated information, which simplifies downstream data retrieval and processing by encapsulating high-level intent and relevant attributes in a standardized format. The `PQt` is implemented as a `dataclass` in Python, with the following fields:

- `issue_key:` Extracted identifier of the JIRA issue, adhering to the defined format.
- `intent_type:` Enum value derived from the `IntentMap`, representing the user's intent.

Parsing Workflow The NLQ parsing workflow begins with the injection of the user query Q_t into the prompt template T. The template is dynamically populated with the intent definitions and examples from the Intent Map (IM), as well as the parsing guidelines

(G_p). This populated template is then processed by a Large Language Model (LLM) to generate a parsed output. From this output, key information such as the `issue_key` and `intent_type` is extracted using regular expressions and predefined mappings. Finally, the extracted data is validated against the PQt schema to ensure correctness, and the validated data is encapsulated as a Parsed Query object. The detailed steps of this workflow are summarized in Algorithm 2.

Algorithm 2: NLQ Parsing Algorithm

Input: Q_t: User Query, T: Prompt Template, IM: Intent Map, G_p: Parsing Guidelines

Output: PQt: Parsed Query

Step 1: Dynamically fill template T with:

- Intent definitions and examples from IM,
- Parsing Guidelines G_p, and
- The user query Q_t.

Step 2: Process the filled template using an LLM to generate the parsed output.

Step 3: Extract `issue_key` and `intent_type` from the output using regular expressions and mappings.

Step 4: Validate the extracted data against the PQt schema.

return PQt.

Design Considerations and Challenges The primary challenge in NLQ Parsing is ensuring disambiguation between overlapping intents, particularly when the user's query is vague or incomplete. This was addressed by:

- Providing clear and non-overlapping intent definitions in the `IntentMap`.
- Including diverse and representative examples in the prompt template.
- Relying on a robust schema validation mechanism to ensure the correctness of the parsed output.

NLQ Parsing Prompt The following prompt guides the NLQ Parsing process:

Prompt Template : NLQ Parsing Prompt

Task: You are an assistant trained to classify queries about JIRA server issues into one of the following intents based on the user query. Each intent has a clear definition and associated examples. Your job is to:

1. Identify the intent of the query.

2. Extract the mentioned issue ID(s) using the regex pattern ‘ISSUE-[a-zA-Z0-9]+‘.

Intents and Definitions:

1. **Direct Metadata Query (MetadataIntent):**

Queries seeking information directly about the metadata or fields of a specific issue. For example:

- ”What is the status of initiative ‘ISSUE-abc‘?”

- ”Who is the assignee of ‘ISSUE-xyz‘?”

- ”When was ‘ISSUE-def‘ created?”

2. **Children Issues Query (ChildrenIntent):**

Queries asking about the child issues of a specific parent issue. For example:

- ”What is the status of all epics under project ‘ISSUE-abc‘?”

- ”List all features under initiative ‘ISSUE-xyz‘.”

3. **Linked Issues Query (LinkedIssuesIntent):**

Queries about issues that are explicitly linked to a given issue, such as related risks or blocked issues. For example:

- ”Are there any risks associated with ‘ISSUE-abc‘?”

- ”List all issues blocked by ‘ISSUE-xyz‘.”

Regex Pattern for Issue ID:

Use the regex pattern ‘ISSUE-[a-zA-Z0-9]+‘ to extract issue IDs from the user query.

Examples:

Query: ”What is the status of initiative ‘ISSUE-abc‘?”

Intent: MetadataIntent

Extracted Issue ID(s): ‘ISSUE-abc‘

Query: ”What is the status of all epics under project ‘ISSUE-xyz‘?”

Intent: ChildrenIntent

Extracted Issue ID(s): ‘ISSUE-xyz‘

Query: ”Are there any risks associated with ‘ISSUE-abc‘?”

Intent: LinkedIssuesIntent

Extracted Issue ID(s): ‘ISSUE-abc‘

—

Given the user query: {query}

1. Identify the intent.
2. Extract the mentioned issue ID(s) using the regex pattern.

4.2.2 Context Retrieval

The Data or Context Retrieval component is responsible for extracting relevant information from the JIRA server, utilizing the *Parsed Query (PQt)* generated in the NLQ Parsing step. It outputs the *Retrieved Data (RDt)*, which encapsulates the requested information in a structured format. This component consists of multiple submodules, each with distinct roles to ensure efficient and accurate data retrieval.

Retrieval Query Planner The Retrieval Query Planner is the first step in the data retrieval process. Its primary function is to form the *Retrieval Query (RQt)* by leveraging

the *Parsed Query (PQt)*, the conversation history and IntentMap. If the current query lacks an issue key and is in continuation of the previous conversation, the planner retrieves the issue key from the conversation history. Based on these inputs, the planner formulates the `RQt` as:

```
RQt = {issue_key, jql_query, post_processing_fn}
```

The planner categorizes retrieval queries into four primary types, depending on the `intent_type`:

(1) **Issue Metadata:** Retrieves core attributes of an issue, such as its status, priority, description, and custom fields.

```
JQL: "issueKey = {issue_key}"
```

(2) **Issue Changelog:** Fetches the historical changes of an issue, including updates to status, assignees, and custom fields.

```
JQL: "issueKey = {issue_key} ORDER BY updated DESC"
```

(3) **Issue's Children of Specific Type:** Retrieves child issues based on a predefined hierarchy (e.g., stories under an epic).

```
JQL: "'Parent' = {issue_key} AND type = {child_type}"
```

(4) **Issue's Linked Issues:** Extracts issues that are related via JIRA link types (e.g., "blocks", "is blocked by").

```
JQL: "issue in linkedIssues({issue_key})"
```

The appropriate `post_processing_fn` is selected from the `IntentMap` using the `sub-schema-selector` associated with the given `intent_type`. This function will be utilized by Retriever to do schema-selection from retrieved data from JiraGraphManager.

JQLProcessor The `JQLProcessor` executes JQL queries asynchronously against the JIRA REST API and is implemented in Python using the `aiohttp` and `asyncio` libraries. Its design incorporates batch processing, pagination, and robust error handling to efficiently retrieve large datasets.

- **Initial Query and Metadata Retrieval:** The first query retrieves search metadata, including the total number of matching issues, `startAt`, and `maxResults`. This metadata determines the batch size and the number of subsequent paginated requests required.
- **Batch Processing and Concurrency:** Following the initial query, subsequent requests are grouped into batches and executed concurrently using `asyncio.gather`. This parallel execution minimizes overall query time and reduces API overhead.
- **Pagination:** The processor utilizes the `startAt` and `maxResults` parameters to manage pagination. It incrementally retrieves data until all matching issues have been fetched.
- **Error Handling and Retry Mechanisms:** The implementation includes a retry mechanism with exponential backoff. In case of transient failures or non-200 HTTP responses, the processor retries the request up to a configurable maximum number of attempts, ensuring resilience against temporary issues.
- **Asynchronous Session Management:** A persistent `aiohttp.ClientSession` is maintained during batch execution to optimize resource usage and minimize the overhead associated with creating multiple sessions.

This modular implementation allows the `JQLProcessor` to be seamlessly integrated with components such as the `JiraGraphManager`, which converts the raw JSON responses into `Node` instances for further processing and caching.

JiraGraphManager The `JiraGraphManager` is responsible for orchestrating JIRA data retrieval while leveraging caching mechanisms to optimize performance. It serves as an intermediary between the `Retriever` and the `JQLProcessor`, ensuring that redundant queries are avoided and previously retrieved data is efficiently stored and reused. It employs a two-layered caching strategy to enhance efficiency:

- **JQL Cache:** A key-value store where the key is a JQL query string, and the value is a list of JIRA issue keys (strings). This cache prevents redundant execution of identical JQL queries.
- **JIRA Graph Database:** A separate in-memory key-value store that maps issue keys to `Node` instances. This structure allows for fast lookups and maintains relationships between issues.

By separating the storage of query results and issue data, the system minimizes redundant API calls and ensures quick access to previously retrieved information.

The retrieval process in `JiraGraphManager` follows a structured sequence:

(1) **Cache Lookup:** When a request is received from the `Retriever` in the form of `(issue_key, jql_query)`, the manager first checks the JQL cache.
   - If the query exists in the cache, the stored list of issue keys is retrieved.
   - The issue keys are used to fetch corresponding `Node` instances from the JIRA Graph Database, which are returned immediately.

(2) **JQL Execution (Cache Miss):** If the JQL query is not found in the cache:
   - The `JQLProcessor` is invoked to execute the query on the JIRA server.
   - The response provides raw JSON data, from which issue keys are extracted.
   - These issue keys are stored in the JQL cache for future use.
   - The retrieved data is transformed into `Node` instances and stored in the JIRA Graph Database.

(3) **Response Generation:** The retrieved `Node` instances are returned to the `Retriever` and stored for future queries.

Node Class and JIRA Graph Database Implementation The `Node` class encapsulates the structure of a JIRA issue. It includes:

- **key:** A unique identifier for the JIRA issue.
- **link:** The direct URL to the issue on the JIRA server.
- **nodetype:** The type of the issue (e.g., *Initiative*, *Feature*, *Risk*, or *Epic*).
- **metadata:** A dictionary containing key-value pairs that describe the issue.
- **children:** A list of `Node` instances representing child issues.
- **related:** A dictionary mapping relation types (e.g., *associated risk*) to lists of related `Node` instances.

The class provides methods to add child nodes via `add_child` and to associate related nodes via `add_related`, enabling the construction of a graph that mirrors the hierarchical and relational structure inherent in JIRA data.

JIRA Graph Database The JIRA graph database is implemented as an in-memory key-value store using Python's `dict` type. Each entry in the dictionary maps a JIRA issue's unique `key` to its corresponding `Node` instance. This design supports:

- **Efficient Lookup:** Direct access to issue data via the unique issue key.
- **Scalable Storage:** Management of all retrieved JIRA issues in memory for quick reference.
- **Simple Maintenance:** Easy addition, removal, and update of nodes within the graph.

The graph database provides methods such as `add_node` to insert a new `Node`, `get_node` to retrieve an existing node by its key, and `remove_node` to delete a node. This mechanism is integral to the system's caching strategy, avoiding redundant JQL queries by storing and reusing previously retrieved data.

Together, the `Node` class and the in-memory graph database form the backbone of our data storage solution, enabling efficient construction, traversal, and management of the JIRA issue graph.

Retriever The `Retriever` class executes the final selection logic. It receives the *Retrieval Query (RQt)* from Retrieval Query Planner and interacts with the `JiraGraphManager` to retrieve relevant data. Based on the JQL specified in the `RQt`, the `Retriever` operates in two key stages:

(1) **Retrieving Raw Data:** The `Retriever` receives a *Retrieval Query* (`RQt`) from the `Retrieval Query Planner`, which consists of the following elements:

- `issue_key`: The primary JIRA issue key used as an entry point for data retrieval.
- `jql_query`: A structured JQL query used to fetch the necessary data.
- `post_processing_fn`: A function that transforms retrieved data into a final structured format.

The `Retriever` forwards this query to the `JiraGraphManager`, which is responsible for checking the cache, executing the JQL query (if necessary), and returning a list of `Node` instances representing the retrieved JIRA issues.

(2) **Post-Processing the Retrieved Data:** Once the raw data is obtained from `JiraGraphManager`, the `Retriever` applies the specified `post_processing_fn`. This function processes the list of retrieved `Node` instances and converts them into a structured list of dictionaries (`List[Dict]`) that align with the expected output format. Depending on the context, the function may:

- Extract specific metadata fields from each `Node`.

- Filter the list based on issue relationships, types, or user-defined conditions.
- Transform hierarchical structures into flattened or summarized formats.

The overall retrieval process is summarized in Algorithm 3:

Algorithm 3: JIRA Data Retrieval Process

Input: Parsed Query $PQt = (issue\_key, intent\_type)$, IntentMap, Conversation History $ConvH$

Output: Retrieved Data RDt

Step 1: Generate Retrieval Query

1.1 Access conversation context from $ConvH$.
1.2 If $issue\_key$ is missing in PQt, extract it from $ConvH$.
1.3 Select appropriate JQL template from $IntentMap$ based on $intent\_type$.
1.4 Construct JQL query jql_query using $issue\_key$ and template.
1.5 Retrieve associated post-processing function $post\_processing\_fn$ from $IntentMap$.
1.6 Formulate Retrieval Query:
$RQt = (issue\_key, jql\_query, post\_processing\_fn)$.

Step 2: Retrieve Data from JIRA Graph Manager

2.1 Invoke `Retriever` with RQt.
2.2 Extract $(issue\_key, jql\_query, post\_processing\_fn)$ from RQt.
2.3 Query `JiraGraphManager` via `get_data(issue_key, jql_query)`.

Step 3: Check and Retrieve from Local Cache

3.1 If jql_query exists in cache:
3.1.1 Retrieve list of issue keys associated with jql_query.
3.1.2 Fetch corresponding `Node` instances from `JiraGraphDB`.
3.1.3 Return retrieved `Node` instances.
3.2 Else, proceed to Step 4.

Step 4: Query JIRA and Update Cache

4.1 Invoke `JQLProcessor` to execute jql_query.
4.2 Retrieve raw issue data from JIRA.
4.3 Convert raw issue data into `Node` instances.
4.4 Store retrieved `Node` instances in `JiraGraphDB` (keyed by issue_key).
4.5 Update cache: Map jql_query to retrieved issue keys.

Step 5: Post-Process Retrieved Data

5.1 Apply $post\_processing\_fn$ to refine `Node` instances into structured output.
5.2 Return the final retrieved data RDt.

return RDt

4.2.3 Answer Generation

The Answer Generation step is the final component of the pipeline, designed to produce a human-friendly response that aligns with the user query (Qt). This step synthesizes the

retrieved contextual information (RDt), the predefined intent-based answer guidelines (Ga), and the user query to construct the final answer. This step ensures that the final response is both informative and aligned with the user's intent. The core of the Answer Generation step lies in the Answer Genertation Prompt Template, which serves as a blueprint for forming the response. It utilizes the identified query intent from (PQt) and answer_guideline property of intent in IntentMap to formulate the prompt.

Prompt Template : Answer Generation

You are an AI assistant that answers user queries concisely and accurately using the retrieved data. Based on the retrieved information, generate a human-friendly response that directly addresses the user's query. Ensure the response is clear, structured, and aligned with the provided guidelines. If multiple data points exist, summarize them concisely while maintaining clarity. If no relevant data is available, state that explicitly in a helpful manner. User Query: "{user_query}"

Retrieved Data:

{retrieved_data}

Guidelines: {guidelines}

Answer:

The generated response is also stored in the contextual memory history to support future queries in the conversation.

Conversation Context History and Feedback The *Conversation Context History* module serves as an in-memory structured cache, designed to maintain a record of the ongoing conversation. This module plays a critical role in enabling coherent multi-turn interactions by storing and retrieving relevant information from previous queries. The history facilitates the generation of updated retrieval queries and ensures continuity in the dialogue.

Components of Context History The context history is represented as a list of tuples, where each tuple contains the following elements:

- **Active Issue Keys:** The issue keys referenced in the current or previous queries.
- **User Queries and Corresponding LLM Answers:** Captures the natural language queries provided by the user and the generated answers.
- **Retrieved Data:** The contextual information fetched from JIRA during the retrieval step.
- **User Feedback:** Feedback provided by the user on the generated response, which includes:
  - *Thumb Up/Down:* Indicates positive or negative feedback.
  - *Text Comment:* Additional details provided by the user for clarification or refinement.

By maintaining this structured cache, the module enables efficient tracking and retrieval of prior conversation data.

Context History Representation The context history is stored as a list of tuples, structured as follows:

```
ConvH = [
    (issue_key_1, user_query_1, answer_1, retrieved_data_1, feedback_1),
    (issue_key_2, user_query_2, answer_2, retrieved_data_2, feedback_2),
    ...
]
```

Each tuple encapsulates all necessary elements to reconstruct the state of the conversation at any given point.

4.2.4 User Interface

Figure 4.2 showcases the user interface of the JIRA-RAG Chatbot System. The interface allows users to query JIRA-related information using natural language, with responses categorized by detected intents. On the left, users can configure the JIRA server URL and optionally provide an OpenAI API key. The main chat area displays user queries and corresponding chatbot responses, which extract relevant metadata, issue relationships, and status details from JIRA. This UI serves as an access point for interacting with the underlying retrieval-augmented generation (RAG) system, making JIRA queries more accessible and efficient.

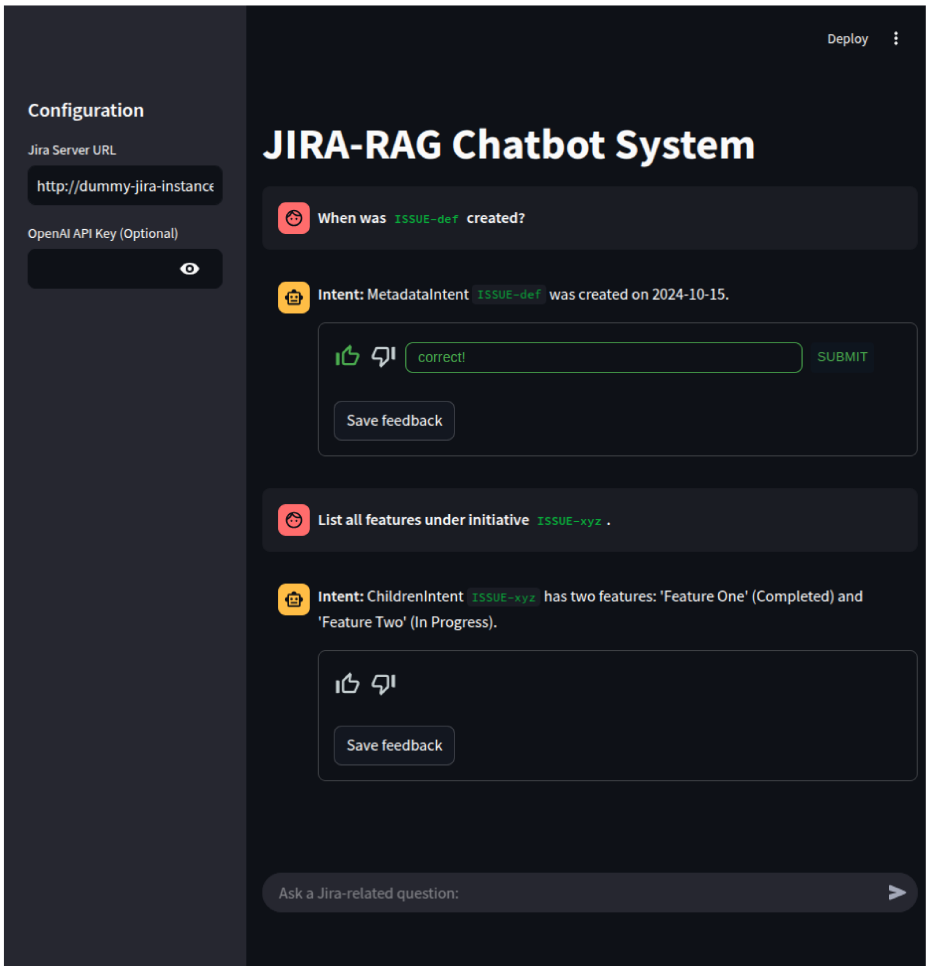

Figure 4.2: User interface of the JIRA-RAG Chatbot System.

4.3 Related Work

Recent advancements in retrieval-augmented generation (RAG) have explored various strategies to enhance large language models (LLMs) for question answering and summarization tasks. While conventional RAG methods focus on retrieving relevant text from external sources, newer approaches have sought to extend RAG capabilities by integrating structured representations, such as graphs and knowledge bases, to improve retrieval efficiency and answer quality. In this section, we discuss two notable approaches that incorporate structured knowledge into RAG: graph-based indexing for query-focused summarization and knowledge graph-enhanced retrieval for customer service question answering. We also highlight how our system builds upon these methods by leveraging a graph-based representation of JIRA issues for more effective retrieval and answer synthesis.

4.3.1 Graph-RAG for Query-Focused Summarization

Traditional RAG techniques struggle with global queries that require summarization rather than explicit retrieval. To address this limitation, Edge et al. (2024) propose a Graph-RAG approach designed for scalable query-focused summarization (QFS). Their method constructs an entity knowledge graph from the source corpus and generates community-based summaries, allowing it to effectively handle broad, high-level questions. While this approach provides a structured way to aggregate information, it relies on a predefined knowledge graph structure. In contrast, our system operates on JIRA issue data, representing issues as nodes in a graph where relationships are derived from inter-issue links, such as parent-child and related issue connections. Unlike Graph-RAG, which generates summaries from community-based entity clusters, our system focuses on retrieving issue-specific context based on query intent without direct summarization.

4.3.2 RAG with Knowledge Graphs for Customer Service QA

While standard RAG methods treat historical customer service tickets as plain text, they often fail to capture the inherent structure and relationships between issues. To overcome this, Xu et al. (2024) introduce a novel approach that integrates a knowledge graph (KG) into RAG-based question answering. By constructing a KG from past issue-tracking data, their method preserves intra-issue structures and inter-issue relations, leading to improved retrieval accuracy and response generation. Similarly, our system also models issue relationships as a graph structure, but instead of a traditional KG, we utilize a graph data structure where each node represents a JIRA issue and edges encode issue dependencies. Our retrieval strategy further refines context selection by mapping query intent to relevant issue properties and metadata, reducing context length and mitigating hallucination risks. Moreover, our system is designed for scalability, allowing users to query JIRA issues at varying granularities, from task-level details to higher-level epic-level insights. Finally, answer synthesis in our system is guided by intent-specific heuristics to generate human-friendly responses, ensuring clarity and contextual relevance.

4.4 Evaluation

The JIRA RAG Chatbot system is comprised of three primary components: NLQ Parsing, Retrieval, and Generation. Among these, the NLQ Parsing component is the most critical, as a correct understanding of the user's query directly influences the subsequent retrieval of relevant data and the quality of generated responses. The retrieval step is implemented as rule-based JQL query-based data extraction from JIRA issues and doesn't require evaluation. Whereas the Answer Generation step is driven by the capabilities of the used LLM and answer guidelines to form a cohesive response. In this section, we evaluate the robustness and effects of prompt-based query understanding within the LLM-based NLQ Parsing component. We first describe our approach to obtaining real-world query data and then detail the experimental setup.

4.4.1 Synthetic Data Generation for NLQ Parsing evaluation

Due to the limited availability of domain-specific query data, we constructed a synthetic dataset that mimics real-world user queries. This dataset is used to evaluate the NLQ Parsing step, which leverages a predefined set of intents from an `IntentMap` along with an issue-key regex pattern. Our synthetic data generation accounts for both single-intent and multi-intent queries. For example, a query such as

> `Can you identify the person assigned to ‘ISSUE-106‘?`

is treated as a single-intent query (MetadataIntent), whereas a query like

> `What risks are linked to ‘ISSUE-106‘? Can you identify the person assigned to ‘ISSUE-106‘?`

comprises multiple intents (MetadataIntent and LinkedIssuesIntent).

Two-Step Data Generation Process

Our dataset is generated using a two-step approach that combines deterministic template-based generation with LLM-driven query variation:

Step 1: Base Query Generation

(1) **Issue Key Generation:** Generate a random issue key (e.g., using the `faker` library).

(2) **Intent Selection:** Randomly select up to two intents from a predefined list.

(3) **Template Substitution:** For each selected intent, substitute the generated issue key into the corresponding template to create a base query.

(4) **Concatenation:** Concatenate all generated base queries to form the final base query.

Step 2: LLM-Based Query Variation

(1) **Template Integration:** Insert the base query into a Synthetic Query Generation Template.

(2) **LLM Query:** Use a large language model (LLM) with an appropriate prompt to generate a coherent variation of the base query.

(3) **Validation:** Validate the generated query by ensuring that the issue key remains unchanged and that the output is a well-formed question. Discard any output that deviates (i.e., exhibits hallucinations or loses the intended query structure).

4.4.2 Predefined Intents

The NLQ Parsing component relies on a set of predefined intents encapsulated in an `IntentMap`. These intents determine the semantic interpretation of user queries and are defined as follows:

```
Available Intents : [
    "MetadataIntent": {
        "definition": "Queries␣seeking␣information␣directly␣about␣
           the␣metadata␣or␣fields␣of␣a␣specific␣issue.",
        "templates": [
            "What␣is␣the␣status␣of␣initiative␣`ISSUE-abc`?",
            "Who␣is␣the␣assignee␣of␣`ISSUE-xyz`?",
            "When␣was␣`ISSUE-def`␣created?"
        ],
    },
    "ChildrenIntent": {
        "definition": "Queries␣asking␣about␣the␣child␣issues␣of␣a␣
           specific␣parent␣issue.",
        "templates": [
            "What␣is␣the␣status␣of␣all␣epics␣under␣project␣`ISSUE-
               abc`?",
            "List␣all␣features␣under␣initiative␣`ISSUE-xyz`."
        ],
    },
    "LinkedIssuesIntent": {
```

```
        "definition": "Queries about issues explicitly linked to a
            given issue, such as related risks or blocked issues.",
        "templates": [
            "Are there any risks associated with `ISSUE-abc`?",
            "List all issues blocked by `ISSUE-xyz`."
        ],
    },
    "ChangelogIntent": {
        "definition": "Queries about the change history of an issue,
            including status updates and field modifications.",
        "templates": [
            "Show the changelog for `ISSUE-123`.",
            "Who changed the status of `ISSUE-456` last?",
            "When was `ISSUE-789` assigned to Alice?"
        ],
    },
    "ReasoningIntent": {
        "definition": "Requests for summarized insights or analytics
            about an issue.",
        "templates": [
            "Summarize the current status of `ISSUE-456`.",
            "Give me a brief overview of `ISSUE-789`.",
            "What is the overall progress of `ISSUE-123`?"
        ],
    }
]
```

4.4.3 NLU Prompt Experiments

The performance of the NLQ parsing pipeline is highly sensitive to the semantics and structure of the prompt provided to the large language model (LLM). In particular, factors such as wording, inclusion of examples, prompt length, and ambiguity can significantly

influence the quality of intent identification. To investigate these effects, we designed a series of experiments focused on prompt engineering. We evaluated the effectiveness of three distinct prompt configurations using our synthetic dataset for both single-intent and multi-intent classification tasks.

Prompt Configurations. We engineered three prompt variants that differ in the level of detail and guidance provided:

- **Prompt1(intent-list) :** A minimal prompt that includes only a list of intents without their definitions.
- **Prompt2(definition-only):** A zero-shot prompt that explicitly provides intent definitions.
- **Prompt3(full):** A comprehensive prompt that includes both explicit intent definitions and illustrative examples.

Experimental Setup. For each prompt variant, we evaluated the NLQ parsing performance on two classification tasks:

(1) **Single-Intent Classification:** Parsing queries associated with a single intent.

(2) **Multi-Intent Classification:** Parsing queries that combine multiple intents.

The synthetic dataset, generated as described in Section 4.4.1, includes a balanced mix of single-intent and multi-intent queries to mimic real-world user inputs.

Evaluation Metrics. We employed three standard evaluation metrics to assess the performance of each prompt configuration:

- **Precision (P):** The fraction of correctly identified intents among all intents predicted.
- **Recall (R):** The fraction of correctly identified intents among all actual intents.
- **F1-score (F1):** The harmonic mean of precision and recall, providing a balanced measure of performance.

Results. Table 4.1 summarizes the performance of the NLQ parsing pipeline under the three prompt configurations for both single-intent and multi-intent scenarios. Table highlights key differences in performance across prompt configurations. The minimal prompt (Prompt1) shows modest scores (around 0.72 for single-intent and 0.81 for multi-intent F1), while the definition-only prompt (Prompt2) slightly improves single-intent performance to 0.80 with little change for multi-intent. In contrast, the full prompt (Prompt3), which provides both definitions and examples, dramatically boosts performance—achieving 0.92 for single-intent and near-perfect score 0.96 for multi-intent scenarios.

Table 4.1: Performance Metrics for NLQ Parsing under Different Prompt Configurations

Prompt Type	**Single-Intent (P, R, F1)**	**Multi-Intent (P, R, F1)**
Prompt1 (intent-list)	(0.72, 0.72, 0.72)	(0.84, 0.80, 0.81)
Prompt2 (definition-only)	(0.80, 0.80, 0.80)	(0.79, 0.76, 0.77)
Prompt3 (full)	(0.92, 0.92, 0.92)	(0.97, 0.96, 0.96)

4.4.4 Taxonomy for Comparing JIRA Plugins with the JIRA RAG Chatbot System

This section presents a taxonomy to compare existing JIRA plugins and conversational tools with the proposed JIRA RAG Chatbot System. The taxonomy highlights gaps in current solutions and positions our System as addressing these limitations through semantic data interaction, domain-specific intent recognition, and efficient context awareness for information retrieval. Existing JIRA plugins, as discussed in Botpress (2024a), Botpress (2024b), Workativ (2024), predominantly function as API wrappers primarily designed for CRUD operations on JIRA issues. In contrast, the JIRA RAG Chatbot System is engineered to overcome these limitations by leveraging the semantic richness of JIRA data and providing more intelligent and context-aware functionalities. The key differentiators between existing plugins and the JIRA RAG Chatbot System are summarized in Table 4.2.

Existing Tools Many JIRA integrations, including chatbots like Botpress and Workativ, primarily act as API wrappers. These tools facilitate basic CRUD operations such as issue creation or updates but typically lack deep interaction with JIRA's underlying semantic

structured data. For instance, while Botpress can create JIRA issues Botpress (2024b), its interaction is limited to API calls rather than leveraging the inherent relationships and context within JIRA's data. Similarly, workflow automation tools like ScriptRunner Adaptavist (2024) automate processes but often operate without real-time data contextualization at a semantic level.

JIRA RAG Chatbot System The JIRA RAG System distinguishes itself by directly integrating with JIRA's semantic data layer. This allows for dynamic and context-aware decision-making by understanding issue hierarchies, custom fields, and inter-issue relationships. For example, it predicts sprint achievements by analyzing historical changelogs of it.

Table 4.2: Feature Comparison: Existing JIRA Plugins vs. JIRA RAG Chatbot System

Feature	**Existing JIRA Plugins**	**JIRA RAG Chatbot System**
Data Interaction	API wrappers focused on CRUD operations	Direct, real-time interaction with semantic data
Semantic Understanding	Limited use of structured data	Leverages issue relationships and metadata
Intent Recognition	Generic, non-domain-specific actions	Context-aware, domain-specific intent mapping
User Experience	Basic functionality and visualization	Conversational interface with session management and user feedback

Chapter 5

Conclusion and Future Work

5.1 Conclusion

In this thesis, I developed and evaluated two distinct applications that successfully demonstrate the efficacy and versatility of Graph-based Retrieval Augmented Generation for conversational question answering over semantically structured data. Through the creation of the KGQA-RAG Chatbot for knowledge graphs and the JIRA-RAG Chatbot for JIRA issue tracking data, my research provides compelling evidence for the benefits of leveraging structured knowledge to enhance RAG based conversational AI systems.

Both applications I developed showcase the core principle of Graph RAG: maintaining the inherent data relationships during retrieval and generation, moving beyond traditional flat text representations. By employing structured queries and graph traversal in KGQA-RAG, and an intent-based design that understands JIRA's semantic data layer in JIRA-RAG, I demonstrated the power of preserving semantic context. The KGQA-RAG Chatbot not only outperformed the baseline CONVINSE method but also achieved performance comparable to a stand-alone KGQA system when evaluated on self-contained questions. Crucially, adding a conversational layer to an existing KGQA approach did not degrade performance, affirming the feasibility of extending KGQA solutions with multi-turn, context-aware capabilities. The JIRA-RAG Chatbot established an innovative approach for creating informative conversational interfaces for complex enterprise data like JIRA issues.

This application highlighted the significant value of contextually interpreting issue hierarchies and custom fields, especially in intricate project management scenarios, through its intent-based and extensible design.

5.2 Limitations

The KGQA-RAG and JIRA-RAG chatbots developed and evaluated in this thesis show considerable promise. However, several limitations emerged during their development and testing, which provide clear opportunities for future improvements. These challenges can be broadly grouped into three categories: conversational context management, data source coverage and access, and system-specific constraints.

Conversational Context Management

Both applications faced challenges managing multi-turn conversation complexities. KGQA-RAG exhibited issues with question reformulation due to focal entity shifts and increasing context complexity, potentially reducing clarity/accuracy and leading to hallucinations/errors. Similarly, while the JIRA-RAG system can manage user query intents, it struggles with ambiguous multi-intent queries and extended multi-turn interactions, which degrade overall dialogue performance. Addressing these issues will require more advanced dialogue state tracking methods tailored to Graph RAG architectures.

Data Source Coverage and Access

A significant limitation for both systems lies in the completeness and accessibility of their underlying data sources. The performance of the KGQA-RAG system is closely tied to the capabilities of its KGQA component, which can become a bottleneck during raw data retrieval from the knowledge graph. In the case of the JIRA-RAG system, the current inability to search within issue attachments restricts its effectiveness, particularly in fields such as legal, finance, or compliance, where critical information often resides in these documents. Enhancing data source coverage to include a wider variety of data formats is

essential for improving the real-world utility and comprehensiveness of KG-RAG systems.

System-Specific Constraints

Beyond the shared limitations, each application also exhibited system-specific constraints. In the KGQA-RAG system, the phenomenon of word jumbling with long named entities during reformulation highlights limitations in the robustness of its natural language processing components. Addressing this will require further refinement to accurately handle complex or unusual textual inputs. For the JIRA-RAG system, its current implementation as a standalone application rather than a native JIRA plugin within the Atlassian ecosystem can impede seamless deployment and diminish user experience in typical JIRA workflows. A deeper integration into the target platform would significantly enhance usability and accessibility for end-users.

5.3 Future Work

Enhancing Conversational Context Management

Future research should focus on overcoming the challenges associated with managing conversational context. In particular, incorporating more advanced dialogue state tracking and enhanced context representation mechanisms could significantly improve system performance.

Expanding Data Source Coverage and Reasoning

To overcome the limitations related to data access and completeness and to enhance the reasoning capabilities of KG-RAG systems, future research should explore:

- **Integrated Attachment Search:** For applications like JIRA-RAG, developing mechanisms to index and semantically search within attachments, enabling access to a richer and more complete set of information associated with structured data.

- **LLM-Enhanced KGQA Components:** Investigating the integration of more advanced Large Language Model (LLM) techniques directly into the KGQA component itself. This could involve using LLMs for more sophisticated knowledge graph traversal, reasoning over implicit relationships, and handling incomplete or noisy knowledge graphs, potentially leading to more robust and accurate information retrieval.

Improving System Integration and User Experience

For practical deployment and broader adoption of KG-RAG systems, future work should focus on enhancing system integration and user experience:

- **Native Platform Integrations:** Developing native plugins or integrations for platforms such as JIRA, enterprise knowledge management systems, and other relevant environments. This integration would streamline deployment, improve user workflows, and enhance overall accessibility.

- **Personalization and Adaptability:** Exploring personalization techniques that allow KG-RAG systems to adapt to individual user preferences, roles, and domain expertise. Such adaptability would create a more tailored and efficient conversational experience.

Appendix A

Master's Coursework and Contributions

A.1 Master Coursework

Course	Course Code	Semester	Grade
DISTRIBUTED SYSTEM DESIGN	COMP6231	Fall 2022	A+
INFO. RETRIEVAL & WEB SEARCH	COMP6791	Fall 2022	B+
DEEP LEARNING	COMP691	WINTER 2023	A
FOUNDATIONS/SEMANTIC WEB	COMP6531	Spring 2024	A+

Table A.1: List of Courses and Grades

A.2 Publications

Omar, R., Mangukiya, O., & Mansour, E. (2025). Dialogue Benchmark Generation from Knowledge Graphs with Cost-Effective Retrieval-Augmented LLMs.

Omar, R., Mangukiya, O., Kalnis, P., & Mansour, E. (2023). Chatgpt versus traditional question answering for knowledge graphs: Current status and future directions towards knowledge graph chatbots. arXiv preprint arXiv:2302.06466.

References

Adaptavist. (2024). *Scriptrunner for jira: Automation and workflows.* Retrieved from https://www.adaptavist.com/products/scriptrunner-for-jira

Anantha, R., Vakulenko, S., Tu, Z., Longpre, S., Pulman, S., & Chappidi, S. (2021). Open-domain question answering goes conversational via question rewriting. *Proceedings of the 2021 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies.*

Anthropic. (2023). *Claude 2 model card.* https://www-files.anthropic.com/production/images/Model-Card-Claude-2.pdf.

Asai, A., Wu, Z., Wang, Y., Sil, A., & Hajishirzi, H. (2023). Self-rag: Learning to retrieve, generate, and critique through self-reflection. *arXiv preprint arXiv:2310.11511*.

Atlassian. (2025a). *Jira — issue & project tracking software.* Retrieved from https://www.atlassian.com/software/jira

Atlassian. (2025b). *Jql: The most flexible way to search jira.* Retrieved from https://www.atlassian.com/blog/jira/jql-the-most-flexible-way-to-search-jira-14

Bollacker, K., Evans, C., Paritosh, P., Sturge, T., & Taylor, J. (2008). Freebase: a collaboratively created graph database for structuring human knowledge. In *Proceedings of the 2008 acm sigmod international conference on management of data* (p. 1247–1250). New York, NY, USA: Association for Computing Machinery. Retrieved from https://doi.org/10.1145/1376616.1376746 doi: 10.1145/1376616.1376746

Botpress. (2024a). *Helpdesk vs botpress.* Retrieved from https://www.helpdesk.com/comparison/helpdesk-vs-botpress/

Botpress. (2024b). *Jira integration documentation.* Retrieved from https://botpress.com/integrations/jira

Brown, T. B., Mann, B., Ryder, N., Subbiah, M., Kaplan, J., Dhariwal, P., ... others (2020). Language models are few-shot learners. *NeurIPS*.

Chowdhery, A., Narang, S., Devlin, J., Bosma, M., Mishra, G., Roberts, A., ... others (2022). Palm: Scaling language modeling with pathways. *JMLR*.

Christmann, P., Saha Roy, R., Abujabal, A., Singh, J., & Weikum, G. (2019). Look before you hop: Conversational question answering over knowledge graphs using judicious context expansion. In *Proceedings of the 28th acm international conference on information and knowledge management* (p. 729–738). New York, NY, USA: Association for Computing Machinery. Retrieved from https://doi.org/10.1145/3357384.3358016 doi: 10.1145/3357384.3358016

Christmann, P., Saha Roy, R., & Weikum, G. (2022). Conversational question answering on heterogeneous sources. In *Proceedings of the 45th international acm sigir conference on research and development in information retrieval* (p. 144–154). New York, NY, USA: Association for Computing Machinery. Retrieved from https://doi.org/10.1145/3477495.3531815 doi: 10.1145/3477495.3531815

Chung, H. W., Hou, L., Longpre, S., Zoph, B., Tay, Y., Fedus, W., ... others (2022). Scaling instruction-finetuned language models. *arXiv preprint arXiv:2210.11416*.

Devlin, J., Chang, M.-W., Lee, K., & Toutanova, K. (2019). Bert: Pre-training of deep bidirectional transformers for language understanding. *NAACL*.

Edge, D., Trinh, H., Cheng, N., Bradley, J., Chao, A., Mody, A., ... Larson, J. (2024). *From local to global: A graph rag approach to query-focused summarization.* Retrieved from https://arxiv.org/abs/2404.16130

Gao, Y., Xiong, Y., Gao, X., Jia, K., Pan, J., Bi, Y., ... Wang, H. (2024). *Retrieval-augmented generation for large language models: A survey.* Retrieved from https://arxiv.org/abs/2312.10997

Google. (2023). *https://gemini.google.com*.

Hu, X., Shu, Y., Huang, X., & Qu, Y. (2021). Edg-based question decomposition for

complex question answering over knowledge bases. In *The semantic web – iswc 2021: 20th international semantic web conference, iswc 2021, virtual event, october 24–28, 2021, proceedings* (p. 128–145). Berlin, Heidelberg: Springer-Verlag. Retrieved from https://doi.org/10.1007/978-3-030-88361-4_8 doi: 10.1007/978-3-030-88361-4_8

Kandpal, N., Deng, H., Roberts, A., Wallace, E., & Raffel, C. (2023). Large language models struggle to learn long-tail knowledge. In *Proceedings of the 40th international conference on machine learning*. JMLR.org.

Lehmann, J., Isele, R., Jakob, M., Jentzsch, A., Kontokostas, D., Mendes, P. N., ... Bizer, C. (2015). Dbpedia - a large-scale, multilingual knowledge base extracted from wikipedia. *Semantic Web*, *6*, 167-195. Retrieved from https://api.semanticscholar.org/CorpusID:1181640

Ley, M. (2009). Dblp - some lessons learned. *Proc. VLDB Endow.*, *2*, 1493-1500. Retrieved from https://api.semanticscholar.org/CorpusID:18916931

Li, S., Gao, Y., Jiang, H., Yin, Q., Li, Z., Yan, X., ... Yin, B. (2023, July). Graph reasoning for question answering with triplet retrieval. In A. Rogers, J. Boyd-Graber, & N. Okazaki (Eds.), *Findings of the association for computational linguistics: Acl 2023* (pp. 3366–3375). Toronto, Canada: Association for Computational Linguistics. Retrieved from https://aclanthology.org/2023.findings-acl.208/ doi: 10.18653/v1/2023.findings-acl.208

Liu, N. F., Lin, K., Hewitt, J., Paranjape, A., Bevilacqua, M., Petroni, F., & Liang, P. (2023). *Lost in the middle: How language models use long contexts.* Retrieved from https://arxiv.org/abs/2307.03172

Ma, X., Gong, Y., He, P., Zhao, H., & Duan, N. (2023a). Query rewriting for retrieval-augmented large language models. *arXiv preprint arXiv:2305.14283*.

Ma, X., Gong, Y., He, P., Zhao, H., & Duan, N. (2023b, December). Query rewriting in retrieval-augmented large language models. In H. Bouamor, J. Pino, & K. Bali (Eds.), *Proceedings of the 2023 conference on empirical methods in natural language processing* (pp. 5303–5315). Singapore: Association for Computational Linguistics.

Retrieved from https://aclanthology.org/2023.emnlp-main.322/ doi: 10.18653/v1/2023.emnlp-main.322

Ma, Y., Cao, Y., Hong, Y., & Sun, A. (2023). Large language model is not a good few-shot information extractor, but a good reranker for hard samples! *ArXiv*, *abs/2303.08559*. Retrieved from https://api.semanticscholar.org/CorpusID:257532405

Minaee, S., Mikolov, T., Nikzad, N., Chenaghlu, M., Socher, R., Amatriain, X., & Gao, J. (2024). *Large language models: A survey.* Retrieved from https://arxiv.org/abs/2402.06196

Omar, R., Dhall, I., Kalnis, P., & Mansour, E. (2023). A universal question-answering platform for knowledge graphs. *Proc. ACM Manag. Data*, *1*(1), 57:1–57:25. Retrieved from https://doi.org/10.1145/3588911 doi: 10.1145/3588911

Omar, R., Mangukiya, O., Kalnis, P., & Mansour, E. (2023). *Chatgpt versus traditional question answering for knowledge graphs: Current status and future directions towards knowledge graph chatbots.* Retrieved from https://arxiv.org/abs/2302.06466

Omar, R., Mangukiya, O., & Mansour, E. (2025). *Dialogue benchmark generation from knowledge graphs with cost-effective retrieval-augmented llms.* Retrieved from https://arxiv.org/abs/2501.09928

OpenAI. (2022). *https://openai.com/blog/chatgpt*.

OpenAI. (2023). *Gpt-4 technical report.* arXiv:2303.08774.

Peng, B., Zhu, Y., Liu, Y., Bo, X., Shi, H., Hong, C., ... Tang, S. (2024). *Graph retrieval-augmented generation: A survey.* Retrieved from https://arxiv.org/abs/2408.08921

Raffel, C., Shazeer, N., Roberts, A., Lee, K., Narang, S., Matena, M., ... Liu, P. J. (2020). Exploring the limits of transfer learning with a unified text-to-text transformer. *JMLR*.

Rajpurkar, P., Zhang, J., Lopyrev, K., & Liang, P. (2016, November). SQuAD: 100,000+ questions for machine comprehension of text. In J. Su, K. Duh, & X. Carreras (Eds.),

Proceedings of the 2016 conference on empirical methods in natural language processing (pp. 2383–2392). Austin, Texas: Association for Computational Linguistics. Retrieved from https://aclanthology.org/D16-1264/ doi: 10.18653/v1/D16-1264

Shao, Z., Gong, Y., Shen, Y., Huang, M., Duan, N., & Chen, W. (2023, December). Enhancing retrieval-augmented large language models with iterative retrieval-generation synergy. In H. Bouamor, J. Pino, & K. Bali (Eds.), *Findings of the association for computational linguistics: Emnlp 2023* (pp. 9248–9274). Singapore: Association for Computational Linguistics. Retrieved from https://aclanthology.org/2023.findings-emnlp.620/ doi: 10.18653/v1/2023.findings-emnlp.620

Sinha, A., Shen, Z., Song, Y., Ma, H., Eide, D., & Wang, K. (2015, May). An overview of microsoft academic service (mas) and applications. In *Www - world wide web consortium (w3c).* Retrieved from https://www.microsoft.com/en-us/research/publication/an-overview-of-microsoft-academic-service-mas-and-applications-2/

Speer, R., & Lowry-Duda, J. (2017, August). ConceptNet at SemEval-2017 task 2: Extending word embeddings with multilingual relational knowledge. In S. Bethard, M. Carpuat, M. Apidianaki, S. M. Mohammad, D. Cer, & D. Jurgens (Eds.), *Proceedings of the 11th international workshop on semantic evaluation (SemEval-2017)* (pp. 85–89). Vancouver, Canada: Association for Computational Linguistics. Retrieved from https://aclanthology.org/S17-2008/ doi: 10.18653/v1/S17-2008

Suchanek, F. M., Kasneci, G., & Weikum, G. (2007). Yago: a core of semantic knowledge. In *Proceedings of the 16th international conference on world wide web* (p. 697–706). New York, NY, USA: Association for Computing Machinery. Retrieved from https://doi.org/10.1145/1242572.1242667 doi: 10.1145/1242572.1242667

Team, G. (2024). *Gemini: A family of highly capable multimodal models.* Retrieved from https://arxiv.org/abs/2312.11805

Touvron, H., Martin, L., Stone, K., Albert, P., Almahairi, A., Babaei, Y., ... others (2023). Llama 2: Open foundation and fine-tuned chat models. *arXiv preprint arXiv:2307.09288*.

Trivedi, H., Balasubramanian, N., Khot, T., & Sabharwal, A. (2023, July). Interleaving retrieval with chain-of-thought reasoning for knowledge-intensive multi-step questions. In A. Rogers, J. Boyd-Graber, & N. Okazaki (Eds.), *Proceedings of the 61st annual meeting of the association for computational linguistics (volume 1: Long papers)* (pp. 10014–10037). Toronto, Canada: Association for Computational Linguistics. Retrieved from https://aclanthology.org/2023.acl-long.557/ doi: 10.18653/v1/2023.acl-long.557

Vaswani, A., Shazeer, N., Parmar, N., Uszkoreit, J., Jones, L., Gomez, A. N., ... Polosukhin, I. (2017). Attention is all you need. In I. Guyon et al. (Eds.), *Advances in neural information processing systems* (Vol. 30). Curran Associates, Inc. Retrieved from https://proceedings.neurips.cc/paper_files/paper/2017/file/3f5ee243547dee91fbd053c1c4a845aa-Paper.pdf

Vrandečić, D., & Krötzsch, M. (2014, September). Wikidata: a free collaborative knowledgebase. *Commun. ACM*, *57*(10), 78–85. Retrieved from https://doi.org/10.1145/2629489 doi: 10.1145/2629489

Walsh, B., Mohamed, S. K., & Nováček, V. (2020). Biokg: A knowledge graph for relational learning on biological data. In *Proceedings of the 29th acm international conference on information & knowledge management* (p. 3173–3180). New York, NY, USA: Association for Computing Machinery. Retrieved from https://doi.org/10.1145/3340531.3412776 doi: 10.1145/3340531.3412776

Wang, X., Wei, J., Schuurmans, D., Le, Q., Chi, E., Narang, S., ... Zhou, D. (2022). Self-consistency improves chain of thought reasoning in language models. *arXiv preprint arXiv:2203.11171*.

Wang, X., Yang, Q., Qiu, Y., Liang, J., He, Q., Gu, Z., ... Wang, W. (2023). *Knowledgpt: Enhancing large language models with retrieval and storage access on knowledge bases.* Retrieved from https://arxiv.org/abs/2308.11761

Wei, J., Wang, X., Schuurmans, D., Bosma, M., ichter, b., Xia, F., ... Zhou, D. (2022). Chain-of-thought prompting elicits reasoning in large language models. In S. Koyejo,

S. Mohamed, A. Agarwal, D. Belgrave, K. Cho, & A. Oh (Eds.), *Advances in neural information processing systems* (Vol. 35, pp. 24824–24837). Curran Associates, Inc. Retrieved from https://proceedings.neurips.cc/paper_files/paper/2022/file/9d5609613524ecf4f15af0f7b31abca4-Paper-Conference.pdf

Workativ. (2024). *Jira service desk chatbot.* Retrieved from https://workativ.com/conversational-ai-platform/jira-service-desk-chatbot

Xu, Z., Cruz, M. J., Guevara, M., Wang, T., Deshpande, M., Wang, X., & Li, Z. (2024). Retrieval-augmented generation with knowledge graphs for customer service question answering. In *Proceedings of the 47th international acm sigir conference on research and development in information retrieval* (p. 2905–2909). New York, NY, USA: Association for Computing Machinery. Retrieved from https://doi.org/10.1145/3626772.3661370 doi: 10.1145/3626772.3661370

Yang, Z., Qi, P., Zhang, S., Bengio, Y., Cohen, W., Salakhutdinov, R., & Manning, C. D. (2018, October-November). HotpotQA: A dataset for diverse, explainable multi-hop question answering. In E. Riloff, D. Chiang, J. Hockenmaier, & J. Tsujii (Eds.), *Proceedings of the 2018 conference on empirical methods in natural language processing* (pp. 2369–2380). Brussels, Belgium: Association for Computational Linguistics. Retrieved from https://aclanthology.org/D18-1259/ doi: 10.18653/v1/D18-1259

Yao, S., Yu, D., Zhao, J., Shafran, I., Griffiths, T. L., Cao, Y., & Narasimhan, K. (2023). Tree of thoughts: deliberate problem solving with large language models. Red Hook, NY, USA: Curran Associates Inc.

Zaib, M., Zhang, W. E., Sheng, Q. Z., Mahmood, A., & Zhang, Y. (2021). *Conversational question answering: A survey.* Retrieved from https://arxiv.org/abs/2106.00874