Master’s Thesis

Deggendorf Institute of Technology, Campus Deggendorf

Faculty of Mechanical Engineering and Mechatronics

Master of Mechatronic and Cyber-Physical Systems

Entwicklung eines Chatbots zur Wissensabfrage durch Integration von Elastic Search Based Retrieval Augmented Generation (RAG) mit Large Language Model (LLMs)

Development of a Knowledge Retrieval Chatbot By integrating Elastic Search Based Retrieval Augmented Generation (RAG) with Large Language Model (LLMs)

This thesis is submitted to obtain the academic degree:

Master of Engineering (M.Eng.)

Submitted by : Muhammed Saleem Kallan

Matriculation Number : 22102058

First Examiner : Ing. Ginu Alunkal

Deggendorf, March 25, 2025

Declaration of original authorship

Name of student: Muhammed Saleem Kallan

Professor/supervisor at DIT: Ing. Ginu Alunkal

If external: name of supervisor: Andre Brueckner

Topic of the bachelor's / master's thesis (final thesis) - German and English title:

Entwicklung eines Chatbots zur Wissensabfrage durch Integration von Elastic Search Based Retrieval Augmented Generation (RAG) mit Large Language Model (LLMs) |
Development of a Knowledge Retrieval Chatbot By integrating Elastic Search Based Retrieval Augmented Generation (RAG) with Large Language Model (LLMs)

1. I hereby declare that this final thesis is my own work, has not been submitted for any other degree at any other university or institution, does not contain or use any sources or resources other than those referenced, and that all direct and paraphrased quotes have been duly cited as such.

Deggendorf, 21.03.2025
(Date)

(signature student)

2. hereby consent to my completed final thesis being made available to a broader public through the library of Deggendorf Institute of Technology.

☒ Yes
☐ No

If yes:
I certify and warrant that I am the sole owner and holder of all rights to the final thesis, including the right to use the tables, figures, illustrations, plans, or similar, submitted therein and that no third-party rights or claims, nor any statutory provisions, will be infringed as a result of its publication.

Deggendorf, 21.03.2025
(Date)

(signature student)

Only to be completed by the examiner if the author consents to the final thesis being published:

Collection of a copy of the final thesis by the library, including the lending of the copy, is hereby endorsed / not endorsed.

☒ endorsed.
☐ not endorsed.

Deggendorf, 21.03.2025
(Date)

(signature examiner)

Acknowledgement

This thesis began as a proof of concept without a clear direction. Despite the uncertainties and challenges along the way, I am proud to have brought it to a successful conclusion.

I would like to express my sincere gratitude to Ing. Ginu Paul Alunkal for his exceptional guidance throughout this research. His deep knowledge, insightful advice, and unwavering support helped me navigate the complexities of this work. His direction provided clarity at critical times when progress seemed challenging.

I am also deeply thankful to my company supervisor, Ing. Andre Brückner, for giving me the opportunity to undertake this thesis at BSH Hausgeräte and for having faith in my potential. His trust, thoughtful feedback, and valuable suggestions significantly shaped the quality and direction of this project. Knowing that he believed in me gave me the motivation to push through even the toughest phases.

My heartfelt thanks also go to Mr. Roman Dolinsky for his critical support in deploying the chatbot for BSH users. His technical expertise and problem-solving skills were instrumental in ensuring the successful implementation of the project. His willingness to step in and offer help made a meaningful difference.

Finally, I want to Thank myself for believing in my abilities and staying resilient through the difficult moments. I am especially proud that this project, which I have devoted more time to than any other work, is now alive and actively being used in the real world atleast inside the department. This journey has tested my limits, but the persistence and dedication that carried me through have been truly rewarding. For me, Completing this thesis stands as both a professional and personal achievement.

Abstract

This master's thesis investigates the design and implementation of an AI driven chatbot that leverages Retrieval Augmented Generation (RAG) techniques with Elasticsearch and Large Language Models (LLMs). Prompted by the limitations of traditional SQL based retrieval particularly the challenges of handling large volumes of text, complex joins, and limited semantic search the research explores Elasticsearch as a more suitable alternative for storing and querying both structured and unstructured data. The thesis outlines a multi phase methodology that begins with data preparation, including direct SQL to Elasticsearch migration and web scraping to handle restricted network environments. It then describes the chatbot's architecture, which integrates a Flask based frontend, Elasticsearch for high speed data retrieval, and internal or external LLM APIs for natural language processing and response generation.

Comprehensive testing and user evaluations indicate that combining Elasticsearch's inverted index and vector search capabilities with LLM driven language understanding yields faster, more accurate, and more contextually relevant answers compared to SQL centric approaches. The thesis concludes that Elasticsearch-powered retrieval augmented by LLMs can significantly enhance knowledge discovery in enterprise settings, reducing search overhead and making complex information more accessible to non-technical users. Finally, it discusses potential future work, such as improving model fine tuning, expanding the data pipeline with additional data sources, and incorporating advanced NLP techniques to further enhance chatbot reliability and user experience.

Contents

List of Abbreviations

Abbreviation	Expansionfig
AI	Artificial Intelligence
ANN	Approximate Nearest Neighbors
API	Application Programming Interface
BERT	Bidirectional Encoder Representations from Transformers
BERTScore	BERT-based text evaluation metric
BLEU	Bilingual Evaluation Understudy
BM25	Best Matching 25
CSV	Comma-Separated Values
DSL	Domain-Specific Language
ELK	Elasticsearch, Logstash, Kibana
ETL	Extract, Transform, Load
GPT	Generative Pre-trained Transformer
HTML	HyperText Markup Language
HTTP	Hypertext Transfer Protocol
HTTPS	Hypertext Transfer Protocol Secure
IDF	Inverse Document Frequency
JSON	JavaScript Object Notation
JDBC	Java Database Connectivity
k-NN	k-Nearest Neighbors
LLM	Large Language Model
NER	Named Entity Recognition
NLP	Natural Language Processing
NoSQL	Not Only SQL
OCR	Optical Character Recognition
PDF	Portable Document Format
PoC	Proof of Concept
QA	Quality Assurance
RAG	Retrieval-Augmented Generation
RDBMS	Relational Database Management System
REST	Representational State Transfer
SL	Security Layer
SME	Small and Medium-sized Enterprise
SOAP	Simple Object Access Protocol
SQL	Structured Query Language
SSL	Secure Sockets Layer
SSO	Single Sign On
SVG	Scalable Vector Graphics
TF	Term Frequency
TF-IDF	Term Frequency-Inverse Document Frequency
TLS	Transport Layer Security

UI	User Interface
URL	Uniform Resource Locator
XML	Extensible Markup Language

1 Introduction

Artificial intelligence (AI) and natural language processing (NLP) have transformed the way humans interact with technology. The ability to communicate with machines through conversational interfaces has evolved significantly, shaping industries such as customer service, healthcare, and enterprise automation. AI powered chatbots have become an integral tool for streamlining interactions, automating responses, and providing real time assistance.

The history of chatbots dates back to the 1960s when ELIZA, one of the first conversational programs, simulated human like responses through simple pattern matching. Later advancements, including PARRY in the 1970s and A.L.I.C.E. in the 1990s, introduced structured response mechanisms. However, these early models lacked contextual awareness and relied on pre scripted interactions [1]. The introduction of voice assistants like Siri (2011), Alexa (2014), and Google Assistant (2016) marked a significant leap forward. With the emergence of deep learning and large language models (LLMs), AI chatbots have reached unprecedented levels of contextual understanding and response generation, enabling more human-like interactions [10].

One of the key challenges in industrial and technical environments is managing and retrieving large volumes of data efficiently. Traditional approaches, such as manually searching extensive documentation or using keyword based queries, are often inefficient and time consuming. AI driven chatbots provide a promising solution by offering intuitive, conversational access to complex information repositories.

This thesis focuses on developing an AI powered chatbot for the Stages Platform, an enterprise project management solution used at BSH Hausgeräte. The chatbot is designed to assist project engineers, process managers, and many other Stages users in retrieving relevant information efficiently. By integrating Elasticsearch for optimized data retrieval and leveraging BSH AIM, an internal LLM API, the chatbot can process natural language queries and provide precise, contextually relevant responses. The system aims to improve productivity by reducing search time, automating repetitive queries, and enhancing user accessibility to project management data.

1.1 Problem Statement

BSH Hausgeräte relies on the Stages platform, a web-based UI for structured project management, offering workflows, data requirements, personnel responsibilities, checklists, and documentation. However, navigating and retrieving relevant information within Stages is a challenge, as it consists of more than five inner layers of web pages, with the final data residing in the fifth layer or later depending on the information required, particularly for new employees and project managers unfamiliar with the system. Several key issues hinder efficiency:

- **Complex Data Structure:** The vast interlinked data makes locating specific project/process related information difficult.
- **Steep Learning Curve:** New users struggle to navigate the platform effectively, leading to inefficiencies.
- **Time Constraints:** Manually searching for workflows and documentation consumes valuable time.
- **Inconsistent Interpretation:** Variations in understanding project management procedures result in inconsistencies.

- **Limited Search Capabilities:** The existing search function does not support natural language queries, making precise information retrieval difficult. A single search term may generate a large number of results, and at times, the relevant data may not be displayed, making it challenging to find the required information.

To address these challenges, this research explores the development of an AI-driven chatbot that serves as an intelligent assistant for Stages users. The chatbot enhances accessibility, allowing users to retrieve information using natural language rather than complex queries. It aims to improve efficiency through instant, accurate responses while streamlining onboarding processes and reducing manual search efforts.

1.2 Research Objectives

The objective of this research is to design and develop a proof-of-concept chatbot capable of efficiently retrieving and structuring responses to user queries based on a predefined dataset. The chatbot should automate data retrieval, reduce manual effort, and streamline complex workflows while ensuring accuracy and contextual relevance.

To achieve these objectives, the system will integrate Natural Language Processing (NLP) techniques using Large Language Models (LLMs) for query interpretation and response formulation. Additionally, the system will utilize Elasticsearch, a distributed search and analytics engine, to enable efficient data storage and retrieval. The primary research objectives include:

1. **Improved Accessibility:** Enables users to search for information using natural language rather than rigid query syntax.
2. **Natural Language Understanding:** Utilize LLMs to enhance the chatbot's ability to interpret and generate precise responses.
3. **Reduced Manual Effort:** Minimizes dependency on document browsing, improving workflow efficiency.
4. **Efficient Information Retrieval:** Implement Elasticsearch to optimize query processing and data access.
5. **Security and Data Privacy:** Implement security measures to protect sensitive data while maintaining efficient processing.
6. **Standardized Responses:** Ensures consistency in the retrieved information.
7. **Enhanced Onboarding:** Assists new employees in understanding project workflows efficiently.

As a proof-of-concept, the chatbot is designed to validate the feasibility of integrating conversational AI within enterprise project management systems. This research highlights how conversational AI enhances enterprise systems by bridging the gap between complex data structures and user-friendly access, ultimately contributing to a more efficient and standardized approach to project management.

1.3 Approach

This thesis adopts a structured approach to designing, developing, and evaluating the chatbot. After establishing the context and motivation in the introductory chapter, Chapter 2 (Groundwork and Fundamental Principles) reviews the key principles driving modern information retrieval covering fundamental NLP concepts, retrieval augmented generation, and the limitations of SQL based systems. Chapter 3 (Methodology) then introduces the chatbot's architecture, comparing a proof of concept SQL RAG chatbot with the more advanced Elasticsearch-based system, illustrating why a dedicated search engine is ultimately more suitable for large-scale, unstructured data.

Building on that foundation, Chapter 4 (Data Preparation) details the extraction and transformation of raw data whether migrated from SQL or web crawled in restricted environments into a format optimized for Elasticsearch indexing. Chapter 5 (Implementation) delves into the practicalities of integrating these data pipelines with a Flask based user interface and LLM-based language understanding. This is followed by Chapter 6 (Testing and Results), which assesses the chatbot's accuracy, speed, and user experience in real world scenarios. Finally, Chapter 7 (Conclusion) summarizes the project's contributions and outlines potential avenues for extending this research, such as integrating more advanced NLP models or expanding support for additional data sources.

Additional Resources

All the code from the data preparation phase to chatbot development has been added to the Appendix section for reference. This includes scripts for extracting and processing data, setting up the user interface, and integrating the chatbot with Elasticsearch and the LLM model. The provided code serves as a valuable resource for understanding the implementation details and can be used as a foundation for future improvements or similar projects. The code has been screened to remove any company-specific data or confidential details to comply with corporate data protection and privacy policies.

2 Groundwork and Fundamental Principles

This chapter is organized into two main sections. The first section introduces key terms and definitions, ensuring a clear understanding of the terminology used throughout this thesis. The second section outlines the fundamental principles and technological background essential for the chatbot's development. Together, these sections provide a solid foundation for understanding the methodologies and technologies employed in this research.

2.1 Key Terms and Definitions

To ensure a clear and consistent understanding, this section defines essential concepts that form the basis of the research. These definitions are critical for comprehending the methodologies and technological frameworks employed in the chatbot's development.

1. **Chatbot**
   A chatbot is an AI-driven conversational agent designed to simulate human-like interactions. It can be rule-based, following predefined responses, or AI-based, utilizing large language models (LLMs) and retrieval models to generate dynamic responses [18]. The chatbot developed in this research integrates:

   - Web Scraping for automated data extraction.
   - Elasticsearch for efficient data storage and retrieval.
   - LLMs for response generation and paraphrasing.
   - A Flask Framework for front-end UI generation.

   This framework enables an intelligent and interactive knowledge retrieval system suitable for enterprise applications.

2. **Knowledge Retrieval Systems**
   A knowledge retrieval system is a technology that stores, indexes, and retrieves relevant information efficiently. It includes a database, indexing mechanisms, and a query-processing module that enables users to access precise information based on their input [5].

3. **Retrieval-Augmented Generation (RAG)**
   Retrieval-Augmented Generation (RAG) is an approach that enhances AI-generated responses by integrating information retrieval with language models. Instead of solely relying on pre-trained models, RAG retrieves relevant documents before generating responses, providing more accurate and contextually relevant answers [15].

4. **Large Language Models (LLMs)**
   LLMs are deep learning-based models trained on vast amounts of textual data. Models such as GPT (Generative Pre-trained Transformer) are capable of understanding and generating human-like text. These models are widely used in chatbots, question-answering systems, and text summarization [4].

5. **Natural Language Processing (NLP)**
   NLP is a branch of AI that enables computers to understand, interpret, and generate human language. Key NLP techniques include tokenization, named entity recognition (NER), and sentiment analysis, all of which contribute to query understanding and document retrieval in chatbot applications [11].

6. **Structured Query Language (SQL)**
   SQL is a programming language used for managing structured databases. It allows for data storage, retrieval, and manipulation using commands such as SELECT, INSERT, UPDATE, and DELETE. Traditional relational databases, such as MySQL, are widely used for structured data management.

7. **MySQL Database**
   MySQL is an open-source relational database management system (RDBMS) that organizes data into structured tables with predefined schemas. While effective for structured queries, it is less suited for full-text or semantic search compared to NoSQL databases like Elasticsearch.

8. **Elasticsearch**
   Elasticsearch is a distributed search engine optimized for full-text search and analytics. Unlike relational databases, it stores data as JSON documents, enabling real-time indexing and retrieval. It supports semantic and vector-based search, making it a suitable alternative to SQL-based data retrieval for chatbot applications [7].

9. **Vector Search**
   Vector search retrieves information based on semantic similarity rather than exact keyword matching. It represents text as dense vector embeddings in multi-dimensional space, and similarity is determined using distance metrics such as cosine similarity or Euclidean distance [22].

10. **Semantic Search**
    Semantic search improves the accuracy of search results by interpreting the intent and contextual meaning of queries. Unlike traditional keyword-based searches, which rely on exact matches, semantic search employs NLP techniques to understand word relationships, synonyms, and contextual nuances [2].

11. **Application Programming Interface (API)**
    An API is a standardized set of protocols that enables different software applications to communicate. APIs define request and response structures, allowing seamless integration of external data sources, services, and functionalities [19]. Common API types include REST, SOAP, and GraphQL, facilitating scalable and interoperable software solutions.

12. **Web Scraping**
    Web scraping is an automated technique for extracting data from websites. Tools such as BeautifulSoup (for parsing HTML) and Selenium (for handling dynamic content) are commonly used for this purpose. Extracted data is often stored in structured formats like JSON and integrated into databases like Elasticsearch.

13. **JSON (JavaScript Object Notation)**
    JSON is a lightweight data format used for data storage and exchange. It is widely adopted in NoSQL databases like Elasticsearch and API-based applications due to its flexibility and readability.

    Example JSON document:

```
{
  "name": "Alice",
  "age": 25,
  "city": "Berlin"
}
```

14. **Query DSL (Domain-Specific Language)**

    Elasticsearch Query DSL is a JSON-based language used for advanced search operations, including full-text search, filtering, aggregation, and ranking [13].

    Example Query DSL request:

```
{
  "query": {
    "match": {
      "title": "chatbot"
    }
  }
```

    This query searches for documents where the title contains ”chatbot.”

    JSON is a data format for storing and exchanging information, while Query DSL is a JSON-based language used to define search operations in Elasticsearch. JSON represents data, whereas Query DSL defines how to search it.

2.2 Introduction to the Groundwork Section

This section establishes the foundational principles necessary for the chatbot's development. These concepts will be further explored in subsequent sections detailing data preparation, system implementation and evaluation.

2.2.1 Background and Importance of Knowledge Retrieval Systems

In today's business landscape, the ability to efficiently retrieve and process vast amounts of structured and unstructured data is crucial for decision-making and process optimization. Many organizations utilize relational databases, such as MySQL, to store structured operational data, while websites serve as a medium to present and manage this information. However, querying large relational databases using natural language remains challenging due to the complexity of SQL queries and the structured nature of database schemas, which are not inherently designed for retrieval-augmented generation (RAG) techniques [23].

Recent advancements in Large Language Models (LLMs) have enabled natural language querying of structured data. However, the direct implementation of RAG techniques with SQL databases has limitations due to schema dependency, query inefficiencies, and difficulties in mapping unstructured questions to structured query languages [23]. Alternative approaches, such as semantic search and vector-based retrieval, have gained traction for improving knowledge retrieval from large datasets.

In modern enterprise environments, effective data retrieval plays a crucial role in improving intelligent chatbot interactions. While integrating Large Language Models (LLMs) with structured databases presents a promising solution, traditional Relational Database Management Systems (RDBMS) like MySQL are not inherently optimized for semantic search or natural language understanding [14]. This section explores various knowledge retrieval approaches, including retrieval-augmented generation (RAG), Elasticsearch-based vector search, and web scraping, which provide alternative methodologies for enhancing chatbot efficiency and accuracy.

2.3 Retrieval-Augmented Generation (RAG)

Retrieval-Augmented Generation (RAG) is an advanced AI framework that enhances response accuracy by integrating information retrieval with language model generation. Unlike traditional generative AI models, which rely solely on pre-trained knowledge, RAG actively retrieves relevant external data before generating a response. This hybrid approach improves factual accuracy, reduces hallucinations, and enhances contextual awareness, making it particularly effective for chatbots, virtual assistants, and question-answering systems [15].

Concept and Mathematical Foundation of RAG

RAG is designed to improve open-domain question answering by combining retrieval-based and generative components. Instead of generating responses solely from a language model, RAG first retrieves relevant documents from a knowledge base, which are then used as input to the model to produce contextually enriched responses [15].
Mathematically, the RAG framework is represented as:

$$P(y|x) = \sum_{z \in \mathcal{Z}} P(y|x, z) \cdot P(z|x) \tag{1}$$

Where:

- x represents the input query
- z denotes the retrieved document from the knowledge base $\mathcal{Z}$
- $P(z|x)$ is the probability of retrieving document z given query x, and
- $P(y|x, z)$ is the probability of generating response y given both the query x and the retrieved document z

The retrieval step is typically implemented using dense vector search, where document embeddings are compared using cosine similarity or other distance metrics. This ensures that the most relevant documents are passed into the generation phase, improving response reliability.

How RAG Works: A Two-Step Process

RAG operates through two core phases:

1. **Retrieval Phase**
   - When a user submits a query, the model searches a knowledge repository, external document collection (e.g., Wikipedia, scientific papers), or company-specific database.
   - It retrieves the most relevant information using vector search, BM25 ranking, or hybrid retrieval techniques.
2. **Generation Phase**
   - The retrieved documents are incorporated as contextual input to the language model.
   - The model then generates a response based on the retrieved knowledge, rather than relying solely on pre-trained data.

This process enables dynamic, contextually aware AI interactions, making chatbots and question-answering systems more accurate, reliable, and adaptable.

Key Advantages of RAG

- Real-Time and Up-to-Date Information: Unlike static LLMs, RAG retrieves real-time data from continuously updated sources, enhancing relevance and adaptability[8].
- Reduction in AI Hallucinations: By grounding responses in retrieved evidence, RAG mitigates AI-generated inaccuracies and improves trustworthiness[6].
- Enhanced Contextual Awareness and Personalization: RAG can be fine-tuned for domain-specific retrieval, allowing AI systems to provide industry-relevant, customized insights[6].
- Improved Handling of Complex Queries: The retrieval process enhances multi-step reasoning, making it more effective for processing highly specific queries[6].

Future of RAG: Towards AI-Augmented Knowledge

As AI technology advances, RAG is expected to become a foundational approach for intelligent, reliable, and fact-driven AI interactions. Future developments in real-time web retrieval, multimodal AI integration (text, image, video), and memory-enhanced learning will further improve its capabilities. By bridging the gap between static AI knowledge and real-time external information, RAG represents a significant step toward more responsive and informed AI systems.

2.4 Elasticsearch for Knowledge Retrieval

Elasticsearch is a distributed search and analytics engine designed for real-time data retrieval, full-text search, and advanced data analysis. Built on Apache Lucene, it enables horizontal scalability, fault tolerance, and fast indexing, making it widely used in academic and industrial applications. Unlike traditional relational databases that rely on structured SQL queries, Elasticsearch is optimized for semantic search, vector-based retrieval, and machine learning integration, making it a powerful tool for knowledge retrieval in AI applications, including chatbot systems.

2.4.1 Architecture of Elasticsearch

Elasticsearch is designed to handle large scale data efficiently by providing fast and scalable search and analytics capabilities. Its architecture is built to support high availability, fault tolerance, and distributed processing. The key components of Elasticsearch include:

Cluster

- A cluster is a collection of one or more nodes that work together to store and process data. It is identified by a unique name.
- Clusters enable horizontal scaling, allowing Elasticsearch to handle petabytes of data by distributing it across multiple nodes.

Node

A node is a single server in the cluster that stores data and participates in indexing and search operations. Nodes can have different roles:

- **Master Node:** Manages cluster-wide operations like creating/deleting indices and shard allocation.
- **Data Node:** Stores data and performs data-related operations (e.g., search, indexing).
- **Coordinating Node:** Routes requests and aggregates results from other nodes.

Index

- An index is a collection of documents with similar characteristics, analogous to a database in relational systems.
- Indices are partitioned into shards for distributed storage and processing.

Shards and Replicas

- **Shards:** Each index is divided into multiple shards, which are distributed across nodes. Shards enable parallel processing and scalability.
- **Replicas:** Each shard can have one or more replicas, which are copies of the shard. Replicas provide fault tolerance and high availability.

Inverted Index

- Elasticsearch uses an inverted index to enable fast full-text search. An inverted index maps terms to the documents that contain them, allowing for efficient querying.
- This data structure is inherited from Apache Lucene, the underlying library powering Elasticsearch.

2.4.2 Working Principles of Elasticsearch

Data Indexing

Elasticsearch stores data in the form of JSON documents, offering a schema-free and highly flexible structure. During the indexing process, documents undergo analysis, including tokenization, filtering, and normalization, before being stored in an inverted index. This ensures efficient retrieval and searching. The indexing process operates in near real-time, allowing documents to become searchable within seconds of being indexed.

Search and Query Execution

Elasticsearch utilizes a Domain-Specific Language (DSL) for constructing complex queries. The query execution process supports various search types, including:

- **Full-text search:** Identifies documents based on relevance scoring.
- **Structured search:** Filters documents using exact matches, such as date ranges and numerical values.
- **Aggregations:** Performs data summarization, including statistical computations such as averages and histograms.

The search process in Elasticsearch follows these steps:

1. Query parsing and validation.
2. Distributed execution across multiple shards.
3. Relevance scoring, employing either the TF-IDF (Term Frequency-Inverse Document Frequency) or BM25 ranking algorithm [21].
4. Aggregation and ranking of search results before returning them to the user.

Distributed Execution

Elasticsearch distributes search queries across multiple nodes and shards, ensuring parallel processing and high-performance execution. The coordinating node manages query distribution and collects results from the relevant shards before compiling and returning the final output to the client.
Elasticsearch is designed to maintain fault tolerance through:

- **Replication:** Each shard has at least one replica stored on a different node, ensuring data redundancy.
- **Automatic recovery:** In the event of node failure, Elasticsearch automatically reassigns shards to other available nodes to maintain system stability and data integrity.

2.4.3 Fundamental Principles of Elasticsearch

Elasticsearch operates on an inverted index structure, where documents are tokenized into terms and mapped to their respective document locations. This indexing approach significantly enhances retrieval efficiency compared to traditional relational databases.
Mathematically, Elasticsearch employs the TF-IDF model to rank search results [12]. The TF-IDF score is calculated as follows:

$$TF - IDF(t,d) = TF(t,d) \times IDF(t) \tag{2}$$

where:

- $TF(t,d)$ represents the frequency of term t in document d.
- $IDF(t)$ denotes the inverse document frequency, given by:

$$IDF(t) = \log \frac{N}{n_t} \tag{3}$$

Where N is the total number of documents, and n_t is the number of documents containing term t.

Beyond traditional keyword-based retrieval, Elasticsearch also supports semantic search using dense vector representations. Techniques such as k-nearest neighbors (k-NN) and approximate nearest neighbors (ANN) enable similarity-based retrieval, which enhances context-aware querying. This feature is particularly beneficial in applications such as chatbot integration and recommendation systems.

2.4.4 Comparison with Other Search Engines

Elasticsearch is frequently compared with other search engines such as Apache Solr and Sphinx. Key differentiators include:

- **Real-Time Capabilities**: Elasticsearch supports near real-time search, whereas Solr requires explicit commits, causing slight delays.
- **Scalability**: Elasticsearch's distributed architecture enables superior scalability compared to Sphinx.
- **Ease of Use**: Elasticsearch's RESTful API and schema-free JSON storage simplify usage, while Solr relies on more complex XML-based configuration.

These differences make Elasticsearch a preferred choice for applications requiring scalability, flexibility, and high-speed search capabilities. A detailed comparison of the features of similar search engines is presented in Table 2, highlighting the key factors that make Elasticsearch the preferred choice among alternative search engines.

2.5 Web Scraping and Web Crawling

Web crawling refers to the process of systematically browsing the web to discover and index web pages. It is typically done by automated programs called web crawlers or spiders.The primary goal of web crawling is to gather information about the structure and content of websites, often for indexing purposes (e.g., search engines like Google use web crawlers to index web pages).A web crawler starts with a list of URLs (seed URLs) and visits each page, extracts links from those pages, and follows those links to discover new pages. This process continues recursively.

Web scraping refers to the process of extracting specific data from web pages. It involves parsing the HTML content of a web page and extracting the desired information.The goal of web scraping is to collect structured data from websites for analysis, storage, or other purposes (e.g., extracting product prices, reviews, or news articles).A web scraper targets specific web pages, retrieves the HTML content, and uses tools or libraries (e.g., BeautifulSoup, Scrapy, or Selenium) to extract the required data. Table 3 provides a comparison between web scraping and web crawling.

Feature	Web Scraping	Web Crawling
Objective	Extracts specific data from web pages	Systematically discovers and indexes web pages
Functionality	Pulls structured information (text, images, tables)	Follows links to explore website structures
Scope	Targets predefined web pages	Covers multiple sites and deep links
Output	Data in structured formats (CSV, JSON, API responses)	URL lists, metadata, and website structures
Challenges	Anti-scraping defenses, CAPTCHAs, IP blocking	Handling robots.txt restrictions, duplicate content
Common Tools	Scrapy, BeautifulSoup, Selenium, Puppeteer	Googlebot, Apache Nutch, Scrapy

Table 3: Comparison of Web Scraping and Web Crawling

2.5.1 Prompt engineering

Prompting is the technique of structuring input for a Large Language Model (LLM) to guide its responses effectively. A well-designed prompt acts as an instruction, question, or context that enables the model to generate relevant and coherent outputs based on its pre-trained knowledge[3].

Prompt engineering involves optimizing and structuring prompts to enhance accuracy, contextual relevance, and reliability in model responses. This process includes refining input phrasing, defining constraints, and embedding context-aware instructions to improve performance in tasks such as natural language understanding, information retrieval, and decision support systems.

In this research, prompts play a crucial role in bridging user queries with the chatbot's response, ensuring the generation of structured outputs, project-related documentation, and process insights. Effective prompt engineering directly impacts the chatbot's ability to retrieve, process, and present information from the STAGES system in an accessible and meaningful way for non-technical users.

Feature	Elasticsearch	Apache Solr	OpenSearch	Vespa	Weaviate
Indexing Model	JSON document-based, schema-flexible	Schema-based with XML & JSON	JSON document-based, schema-flexible	Document and tensor-based	Schema-free vector-native
Query Language	Query DSL (JSON-based), full-text & structured search	Solr Query Parser, Lucene-based	Query DSL (JSON), SQL-like support	YQL (Vespa Query Language)	GraphQL, RESTful APIs
Full-Text Search	Yes (BM25, TF-IDF, fuzzy search)	Yes (BM25, Lucene-based)	Yes (BM25, full-text ranking)	Yes (BM25, tensor ranking)	Limited (optimized for vector search)
Semantic Search (NLP)	Vector search via k-NN, ANN support	Limited NLP, not vector-native	Vector search, hybrid retrieval	Native support for embeddings	Optimized for vector-based semantic retrieval
Scalability	Horizontally scalable, multi-node clustering	Distributed but less dynamic scaling	Horizontally scalable, AWS-supported	Highly scalable, built for real-time AI inference	Cloud-native, designed for large-scale vector search
Real-Time Indexing	Yes, near real-time updates	Requires commit/soft commit	Yes, optimized for real-time logs	Yes, designed for low-latency AI	Yes, optimized for real-time vector similarity
Vector Search (k-NN, ANN)	k-NN via plugins (HNSW, FAISS integration)	No native support, requires external tools	k-NN, ANN (HNSW), native ML pipelines	Tensor-based nearest-neighbor search	Fully vector-native, optimized for embeddings
Machine Learning Integration	ML models via Eland, Hugging Face NLP	No built-in ML, requires external processing	ML models via AWS SageMaker, NLP support	Native support for AI inference	Deep learning-based, embedding-centric
Enterprise Security Features	Role-based access, TLS, multi-tenant support	Basic authentication, requires external tools	Role-based access, AWS security integration	Enterprise-ready, supports fine-grained access control	API key-based security, designed for multi-user AI applications
Use Cases	General search, log analysis, AI-powered search	Enterprise search, log processing, website indexing	AWS-hosted search, AI-driven data retrieval	Real-time AI search, recommendation systems	AI-first vector search, deep-learning applications

Table 2: Comparison of Search Engine Features

3 Methodology

The methodology is organized into two chapters, each focusing on a key aspect of the research process. The first chapter outlines the chatbot's architecture and explains the limitations of the initial SQL-based RAG chatbot, which prompted the transition to Elasticsearch despite the availability of a MySQL database. It justifies the decision to invest in a new database system by highlighting the benefits in terms of performance and efficiency. This naturally leads into the second chapter, which details the data preparation process and its importance in enhancing the chatbot's overall functionality and accuracy.

3.1 Background and Context

In modern industrial and corporate environments, efficient data retrieval and knowledge management are essential for optimizing workflows and facilitating informed decision-making. Organizations frequently utilize Relational Database Management Systems (RDBMS) to store structured information in an interconnected format. However, the complexity of these databases often poses significant challenges for non-technical users seeking specific information efficiently.

BSH Hausgeräte GmbH, a leading home appliance manufacturer in Germany, employs an internal web application called STAGES, which serves as both a knowledge database and a process management system. STAGES consolidates critical data related to projects, milestones, process structures, responsible roles, criteria for each step, and associated documentation. This information is stored in a MySQL relational database, organizing data across multiple interconnected tables to ensure data integrity and efficient storage. However, this relational structure inherently restricts accessibility and usability for non-technical users, creating a bottleneck in information retrieval.

3.1.1 Problem Statement

Despite the extensive knowledge base within STAGES, retrieving specific data remains a challenge for end users, particularly project managers and process engineers who lack proficiency in SQL. The key challenges contributing to this issue include:

1. Complexity of the Web Application Interface

- The STAGES application is designed to accommodate a wide range of project management functionalities, requiring users to navigate multiple layers of menus, buttons, and hierarchies.
- Due to the vast amount of stored information, users must manually sift through numerous screens to locate the required data.
- Although the application offers a search functionality, it is often inefficient due to ambiguity in naming conventions and overlapping document titles, leading to difficulties in selecting the correct information.

2. Relational Structure of the Database

- MySQL, as a relational database system, stores data across multiple tables with relationships maintained via foreign keys and hierarchical structures.
- Role assignments, artifacts, process steps, and project dependencies are stored in separate tables, interconnected through parent-child relationships.

- Extracting meaningful insights necessitates complex SQL queries with multiple joins, which are beyond the expertise of most end users.

3. Limitations in Search and Querying Capabilities

- The existing search functionality in STAGES relies on text-based keyword matching, lacking semantic understanding and contextual relevance.
- As a result, searches frequently return multiple entries with similar names, making it difficult for users to identify the most relevant result.
- Traditional database querying via SQL remains an alternative, but it is not accessible to project managers or process engineers without SQL expertise.

4. Lack of Natural Language Interaction for Data Access

- The absence of a natural language interface forces users to either:
  - Manually navigate the web application, which is time-consuming and inefficient.
  - Rely on technical personnel to execute SQL queries, creating dependency and inefficiencies.
- There is a growing need for an intuitive system that allows users to retrieve structured data through natural language processing (NLP) instead of requiring technical expertise in SQL.

3.1.2 Scientific and Industrial Relevance of an NLP-Based Chatbot

Organizations face challenges in enterprise knowledge management due to difficulties in accessing large-scale relational databases. Integrating NLP with structured data retrieval enhances Retrieval-Augmented Generation (RAG) for enterprise Q&A systems, improves semantic search for structured and unstructured data, and enables conversational AI to bridge the gap between human language and database access. From an industrial perspective, inefficient data retrieval reduces productivity, delays decision-making, and increases operational inefficiencies[16].

An AI-powered chatbot can enhance project efficiency by providing instant access to critical data, support data-driven decision-making through seamless insights, and automate routine queries, reducing dependency on technical staff. Implementing an NLP-based solution enables automated SQL query generation from natural language inputs, improves semantic understanding for more accurate retrieval, enhances accessibility for non-technical users, and ensures scalability across enterprise databases. By bridging the gap between human cognition and structured data, this approach significantly improves efficiency in enterprise knowledge management.

3.2 System Architecture

The system is built with modular components, allowing flexibility in data storage, processing, and deployment while ensuring scalability and maintainability (Figure 1). The following sections provide a brief overview of each component and its role within the chatbot architecture.

1. **Flask-Based Web Application**
   - Acts as the front-end interface for user interactions.
   - Handles incoming queries, sends requests to the backend, and returns responses to users.
   - Provides API endpoints for asynchronous communication with other services.

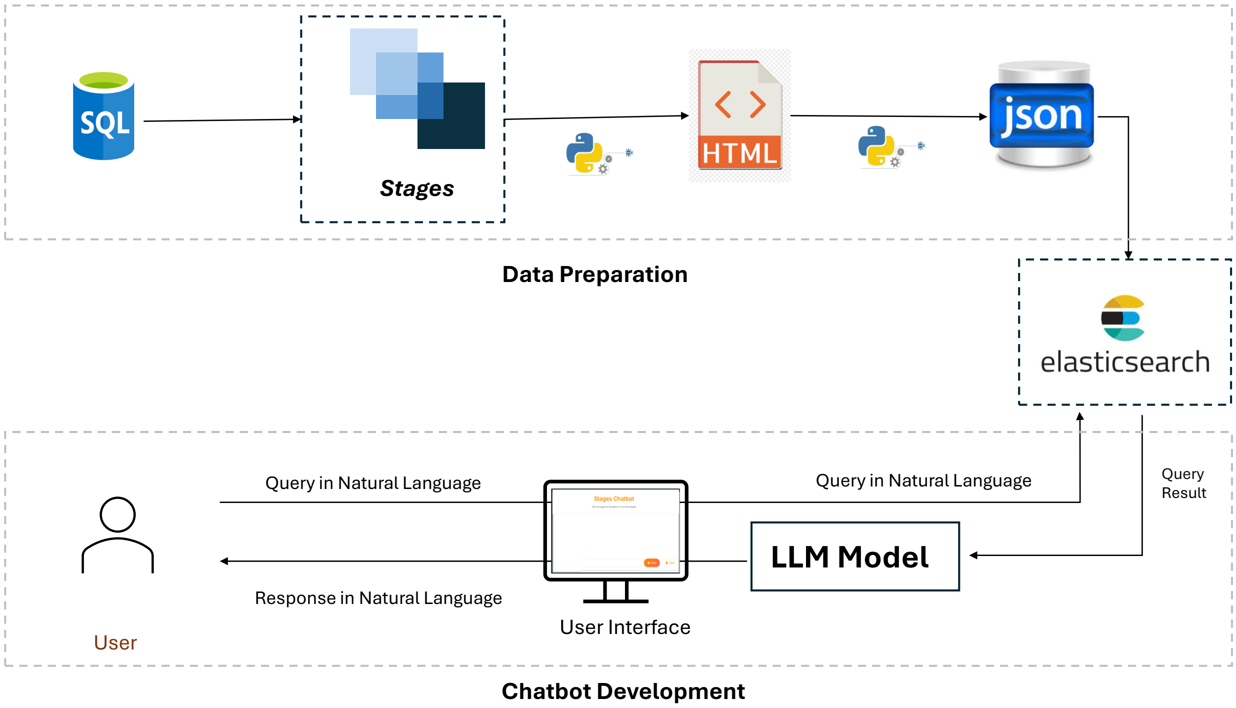

Figure 1: System Architecture

2. **Elasticsearch Database**

   - Serves as the primary data storage for indexing and retrieving information.
   - Enables fast, full-text search and supports semantic retrieval for chatbot queries.
   - Optimized for handling structured and unstructured data efficiently.

3. **Custom LLM Service**

   - Interfaces with the Large Language Model (LLM) API to generate contextually relevant responses.
   - Processes user queries, incorporates relevant retrieved data, and formulates natural language replies.
   - Ensures responses are aligned with the chatbot's knowledge base and user intent.

4. **Session Management**

   - Implements a session-based mechanism to maintain conversation continuity.
   - Tracks previous user interactions, allowing the chatbot to understand context over multiple exchanges.
   - Enhances user experience by avoiding redundant responses and improving personalization.

5. **Environment Configuration & Deployment**

   - Uses dotenv and argparse to manage environment-specific configurations.
   - Ensures smooth deployment by handling API keys, database credentials, and other configurations securely.
   - Facilitates flexible adjustments to settings without modifying the core codebase.

Together, these components create a robust and scalable chatbot system, ensuring fast, intelligent, and context-aware interactions.

Process selection

Since the Retrieval-Augmented Generation (RAG) approach is the core principle in this chatbot development process, a compatibility study of RAG over SQL database and Elasticsearch is discussed, evaluating its effectiveness in enhancing knowledge retrieval and response generation. It begins with an SQL-based retrieval approach, detailing its structure and the integration of Retrieval-Augmented Generation (RAG) in SQL databases. The limitations of this approach, including challenges with unstructured data, semantic search, and scalability, are then examined. To address these issues, the chapter introduces the transition to an alternative search engine, highlighting its advantages in improving retrieval efficiency and accuracy.

3.3 SQL-Based Retrieval (RAG)

3.3.1 Database Structure in SQL

SQL databases store data in a structured, tabular format using a relational model. The data is organized into tables, which consist of rows (records) and columns (attributes). Each column has a defined data type, and relationships between tables are maintained using keys (primary and foreign keys).
SQL databases follow a relational model, where data is structured into tables with predefined schemas. Each table contains:

- **Columns (Fields)**: Define the type of data stored (e.g., INT, VARCHAR, DATE).
- **Rows (Records)**: Represent individual entries in a table.
- **Primary Key (PK)**: A unique identifier for each record.
- **Foreign Key (FK)**: A reference to another table to maintain relationships.

Data Storage in SQL Server

SQL Server stores data using a structured, hierarchical approach to ensure efficient storage and retrieval. At the logical level, data is organized in tables consisting of rows and columns. However, at the physical level, SQL Server uses data pages as the fundamental unit of storage, each of which is 8 KB in size. When new data is inserted into a table, SQL Server distributes and stores this data across multiple data pages, ensuring an optimal structure for fast retrieval [9].

To efficiently manage and retrieve data, SQL Server organizes these pages using a B-tree (Balanced Tree) structure, also known as an Index B-Tree or Clustered Index Structure (Figure 2). In this hierarchical tree structure, the leaf nodes (data pages) contain the actual data, while the root node sits at the top of the tree. Between the root node and the leaf nodes, SQL Server maintains intermediate index nodes that act as navigation points, helping the database engine locate data quickly. This structured approach significantly reduces search time compared to scanning entire tables [9].

One of the most important aspects of data storage in SQL Server is the use of clustered indexes. When a primary key is defined on a table, SQL Server automatically creates a clustered index, which dictates the physical order of rows based on the indexed column. This means that data in the underlying storage is sorted according to the primary key or another designated clustered index column[9].
In conclusion, SQL Server employs a sophisticated data storage mechanism to balance storage efficiency and retrieval speed. By using data pages, B-tree indexing, and clustered and non-clustered indexes, SQL Server optimizes query performance and minimizes search time.

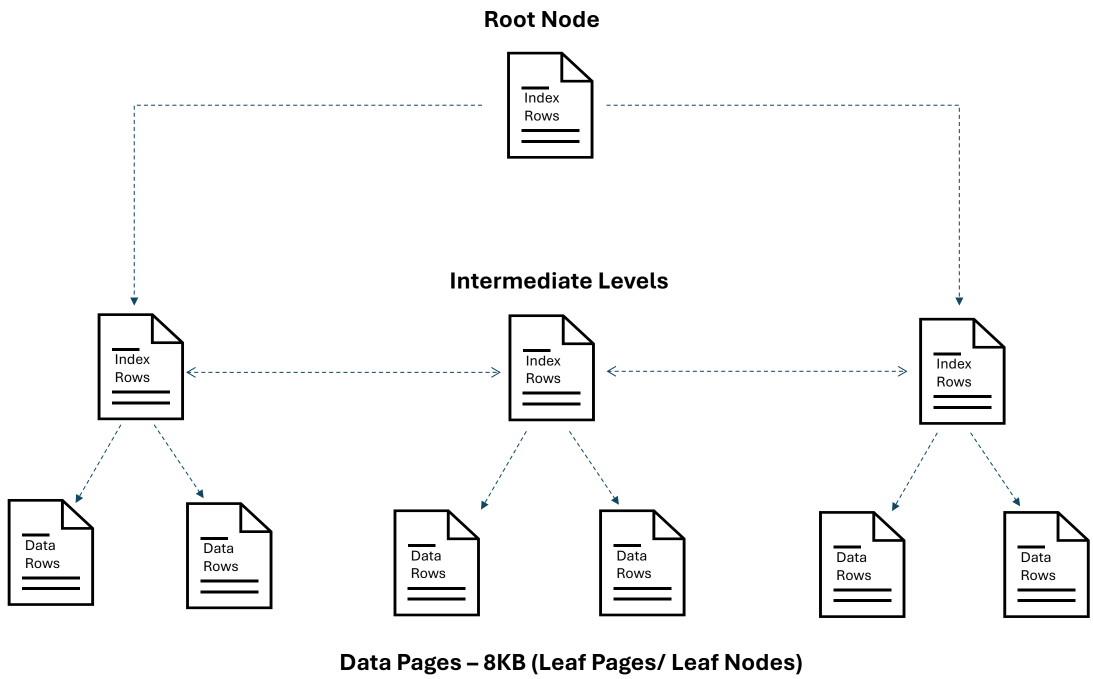

Figure 2: B-Tree Structure

3.3.2 Chatbot Using SQL-Based Retrieval (RAG)

The SQL-based retrieval system for knowledge retrieval in chatbots allows non-technical users to interact seamlessly with structured databases using natural language. This approach leverages RAG that combines retrieval from structured SQL databases with natural language generation to provide meaningful responses. Figure 3 illustrates the Architecture of the chatbot.

How It Works

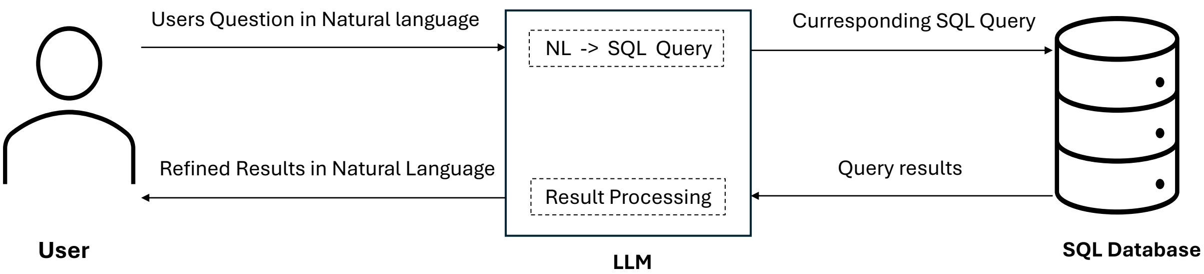

Figure 3: RAG SQL Chatbot architecture

This system integrates Large Language Models (LLMs) with a relational database, enabling chatbots to process user queries intelligently. The workflow typically follows these steps:

1. **Natural Language Input:**
   - The user enters a query in natural language, such as:
     ”What were the total sales for January?”
   - The chatbot processes this input and identifies key terms, such as ”total sales” and ”January.”

2. **Query Interpretation and Conversion:**

   - The LLM interprets the user's intent and converts the natural language query into a structured SQL query.

     Example conversion:

     ```
     SELECT SUM(sales_amount)
     FROM sales_data
     WHERE MONTH(sale_date) = 1;
     ```

   - This step ensures that the query aligns with the database schema and retrieves the correct data.

3. **SQL Query Execution:**

   - The chatbot executes the generated SQL query on the relational database.
   - The database processes the query and returns a structured response, usually in tabular format.

4. **Natural Language Response Generation:**

   - The raw SQL output is transformed into a human-readable format using the LLM.
   - Example transformation:
     SQL result: Total Sales = $150,000
     Chatbot response: "The total sales for January were $150,000."

5. **Context-Aware and Multi-Turn Conversations:**

   - The chatbot can maintain context in multi-turn conversations.
     Example:
     User: "What were the total sales for January?"
     Chatbot: "The total sales for January were $150,000."
     User: "How does that compare to February?"
     Chatbot: "In February, total sales were $175,000, which is a 16.7% increase from January."

3.3.3 Challenges and Limitations of SQL-RAG Chatbot

While SQL databases are traditionally used for structured data storage and retrieval, they face significant limitations when applied in Retrieval-Augmented Generation (RAG) systems for chatbot applications. These challenges arise due to the nature of SQL's structured query mechanisms, lack of semantic capabilities, and scalability constraints [26]. The following sections provide a detailed breakdown of these limitations, along with real-world examples in enterprise applications, ultimately leading to an alternative solution.

1. **Lack of Semantic Understanding in SQL-Based RAG**

   Challenge:

   SQL databases lack semantic search capabilities, meaning they require precise keyword matches to retrieve relevant information. Unlike modern search engines or vector-based databases, SQL Database does not understand relationships between concepts in natural language queries[17].

   Example:

- A project manager in an organization searches for ”milestone approval process” in a chatbot.
- The SQL query looks for an exact match in predefined database fields such as milestone_name, approval_status, or process_steps.
- However, if the stored data refers to the milestone as ”phase completion validation”, SQL will fail to retrieve relevant results.

Limitation:

RAG models require contextual retrieval, but SQL can only perform exact string matches unless additional layers of NLP-based query conversion are added, which introduces computational overhead and inaccuracy.

2. **Difficulty in Handling Unstructured Data**

Challenge:

SQL databases are optimized for structured data stored in tables and predefined schemas. However, most enterprise knowledge, such as process documentation, project reports, artifact descriptions, and workflow guides, is unstructured and difficult to store in SQL tables.

Example:

A software development team maintains user stories, sprint reports, and artifact tracking records in text-based documents. These documents include historical data, such as previous sprint discussions, rejected change requests, and evolving project milestones.

Problem with SQL Based RAG:

- Storing these in an SQL database requires breaking them into structured tables (story_id, sprint_number, artifact_reference), leading to loss of contextual information.
- Retrieving meaningful insights (e.g., ”Find previous rejected change requests in this milestone”) would require complex joins across multiple tables, making queries inefficient.

Limitation:

RAG requires contextual document retrieval, but SQL cannot efficiently store or retrieve free-text documents without additional indexing mechanisms.

3. **Complex Query Formulation**

Challenge:

SQL-based retrieval systems require structured queries, meaning natural language queries must be converted into SQL statements. This requires an additional query conversion layer that is often inaccurate and computationally expensive[17].

Example:

A business analyst asks the chatbot:

”Who approved the last three project artifacts in Phase 2?”

In SQL, this would require dynamically generating the query:

```
SELECT artifact_name, approved_by, approval_date
FROM project_artifacts
```

```
WHERE phase = 'Phase 2'
ORDER BY approval_date DESC
LIMIT 3;
```

However, if the user rephrases the query as "List the latest artifacts signed off in Phase 2", the chatbot's SQL conversion logic might fail to interpret "signed off" as "approved", leading to incorrect results.

Limitation:

SQL-based RAG struggles with flexible query interpretation, as NLP-based SQL conversion models are prone to misinterpretation.

4. **Scalability and Performance Issues**

   Challenge:

   As the volume of data grows, SQL queries become slower and more resource-intensive. Complex queries involving joins across multiple tables require increased processing time, impacting chatbot response speed[17].

   Example:

   A large enterprise stores thousands of process-related documents, including project timelines, regulatory compliance reports, and quality assurance phases.

   A user requests:

   "Retrieve all process artifacts related to software testing across different projects."

   In an SQL-based RAG system, this would involve:

   - Performing a multi-table join across projects, phases, artifacts, and QA_records.
   - Filtering records based on process type = 'Software Testing'.
   - Sorting results for relevance.

   This introduces significant latency, especially if the database contains millions of rows.

   Limitation:

   SQL's relational model struggles with massive datasets, making query execution slower as data grows.

5. **Schema dependency**

   Challenge:

   SQL databases require schema-dependent queries, meaning they cannot dynamically adapt to new types of data without altering the schema.

   Example:

   In a product development project, a new regulatory phase is introduced that requires tracking additional compliance artifacts.

   In an SQL-based system:

   - A new table and schema modifications are required to store this data.
   - Existing queries must be updated to accommodate the new phase.

Due to these challenges, Elasticsearch, a distributed search and analytics engine, was explored as an alternative.

3.4 Elasticsearch Indexing Mechanism

Elasticsearch employs a unique data storage mechanism called the inverted index, which enables lightning-fast search capabilities even when dealing with vast amounts of data. Unlike traditional relational databases that store data in structured tables with rows and columns, Elasticsearch organizes and indexes data in a way that prioritizes search efficiency over traditional transactional consistency.

What is an Inverted Index?

An inverted index is a data structure that maps terms (words) to the documents in which they appear. This is fundamentally different from how SQL databases store data in a tabular format, where each row represents an entry and retrieval relies on table scans or indexed lookups[24].

How It Works:

- Instead of storing documents sequentially, Elasticsearch breaks documents into individual terms and maintains a mapping of which documents contain which terms.
- This structure enables Elasticsearch to quickly locate and retrieve documents that match a search query, without having to scan entire datasets.
- The key advantage is that searching happens in a pre-indexed format, making retrieval orders of magnitude faster than conventional full-table scans in SQL databases.

Forward Index vs. Inverted Index

DocID	geo-scopeID (Forward Index)
1	Europe
2	Europe
3	France
4	England
5	Portugal
6	Quebec
7	Europe
8	Spain

Table 4: Forward Index

geo-scopeID (Inverted Index)	DocID
Europe	1, 2, 7
France	3
England	4
Portugal	5
Quebec	6
Spain	8

Table 5: Inverted Index

- The Forward Index stores document IDs mapped to locations (Table 4).
- The Inverted Index stores locations mapped to multiple document IDs (Table 5), making it easy to retrieve documents by location.
- The Documents Table contains document IDs and their respective texts (Table 6a).
- The Terms Table lists the unique words found in the documents (Table 6b).
- The Index Table stores which documents contain which terms, allowing quick full-text search (Table 6c).

DocId	Docs
1	To be or not to be
2	To be right
3	Not to be left

(a) Documents Table

TermId	Term
1	be
2	left
3	not
4	or
5	right
6	to

(b) Terms Table

Terms	Docs
1 (be)	1:2 [2,6], 2:1 [2], 3:1 [3]
2 (left)	3:1 [4]
3 (not)	1:1 [4], 3:1 [1]
4 (or)	1:1 [3]
5 (right)	2:1 [3]
6 (to)	1:2 [1,5], 2:1 [1], 3:1 [2]

(c) Index Table

Table 6: Document, Terms, and Index Representation

Understanding the Index Table

The table consists of two columns:

1. **Terms:** This lists individual words (tokens) extracted from the documents.
2. **Docs:** This shows which document(s) contain the term and at what positions within those documents.

Each entry in the "Docs" column follows this format:

- DocID:Position [Occurrences]
  - DocID refers to the document in which the term appears.
  - Position indicates the index position of the term in the document.
  - Occurrences is the count of times the term appears in that document.

Example: The term "be" (1) in Docs: 1:2 [2,6], 2:1 [2], 3:1 [3]

- Appears in:
  - **Doc 1:** Position 2 and 6, occurs twice
  - **Doc 2:** Position 1, occurs once [2]
  - **Doc 3:** Position 1, occurs once [3]
- The word "be" appears in positions 2 and 6 in Document 1.

3.5 Comparison: Elasticsearch vs. SQL Databases

Table 7 provides a comparative overview of SQL databases (RDBMS) and Elasticsearch, highlighting their structural differences, data relationships, flexibility, and use case suitability.

Aspect	SQL Database (RDBMS)	Elasticsearch (NoSQL, Document-Oriented)
Structure	Stores data in structured tables with predefined schemas.	Stores data as JSON documents in a schema-less format.
Data Relationships	Uses foreign keys and joins to relate data across multiple tables.	Uses nested documents and denormalization, reducing the need for joins.
Flexibility	Rigid structure; schema changes require migrations.	Highly flexible; new fields can be added dynamically.
Use Case Suitability	Best for structured data, transactions, and business applications.	Best for text search, analytics, and unstructured data retrieval.

Table 7: Comparison of SQL and Elasticsearch Databases

While SQL databases excel in handling structured data and transactional operations, Elasticsearch is optimized for full-text search, analytics, and handling unstructured or semi-structured data. The choice between the two depends on the specific application requirements, with SQL being ideal for relational data management and Elasticsearch offering superior search and retrieval capabilities.

To further illustrate the differences between SQL databases and Elasticsearch, Table 8, Table 9 and Table 10 demonstrates how data is stored in each system. In a SQL database, data is organized into structured tables with predefined schemas, whereas in Elasticsearch, data is stored as JSON documents in a flexible, schema-less format.

ProjectID	Name	StartDate	EndDate
2001	Website Redesign	2024-01-01	2024-06-30
2002	ERP Implementation	2024-03-01	2024-09-30

Table 8: Projects Table

MilestoneID	ProjectID	Name	DueDate
3001	2001	Wireframe Completion	2024-02-15
3002	2002	Requirements Finalized	2024-04-10

Table 9: Milestones Table

ArtifactID	ProjectID	Type	Location
4001	2001	Design Docs	/docs/website_wireframes.pdf
4002	2002	Requirement	/docs/erp_requirements.xlsx

Table 10: Artifacts Table

Elasticsearch Storage (JSON Document Format)

Instead of storing data across multiple tables and performing JOIN queries, Elasticsearch stores all project-related data in a single JSON document, enabling fast and efficient retrieval.

```
{
```

```
  "project": {
    "id": 2001,
    "name": "Website Redesign",
    "start_date": "2024-01-01",
    "end_date": "2024-06-30"
  },
  "milestones": [
    {
      "milestone_id": 3001,
      "name": "Wireframe Completion",
      "due_date": "2024-02-15"
    }
  ],
  "artifacts": [
    {
      "artifact_id": 4001,
      "type": "Design Docs",
      "location": "/docs/website_wireframes.pdf"
    }
  ]
}
```

Example Query Performance

Table 11 compares the query execution and performance of SQL databases and Elasticsearch, focusing on aspects such as query language, keyword matching, joins, aggregations, and vector search capabilities. While SQL databases rely on structured queries and relational operations, Elasticsearch is optimized for full-text search, real-time analytics, and vector-based retrieval. This comparison highlights how each system processes queries differently and their suitability for various use cases.

Aspect	SQL Database	Elasticsearch
Query Language	Uses SQL (Structured Query Language) for precise, schema-based queries.	Uses DSL (Domain-Specific Language), supporting full-text search and vector search.
Keyword Matching	Relies on exact keyword matching using LIKE or FULL-TEXT search.	Uses fuzzy matching, NLP-based search, and tokenization for better contextual understanding.
Joins	Queries require JOINs across multiple tables, slowing performance.	Eliminates joins by storing all related data together in documents.
Aggregation and Analytics	Requires GROUP BY and HAVING clauses, leading to slow performance on large datasets.	Provides real-time aggregations using inverted indices and shards.
Vector Search	Not natively supported.	Supports vector similarity search for NLP-based retrieval.

Table 11: Comparison of Query Execution and Performance Between SQL Databases and Elasticsearch

Query Performance in Retrieval-Augmented Generation (RAG) Chatbots

To highlight the differences in query performance between SQL-based retrieval and Elasticsearch-based retrieval in Retrieval-Augmented Generation (RAG) chatbots, the following example demonstrates how each system processes a user query. While SQL databases rely on structured queries and indexing, Elasticsearch utilizes full text search and relevance scoring to retrieve the most relevant information. The examples below compares how the same query is executed in both systems.

Query:

"Show me all artifacts related to the ERP Implementation project."

SQL Query (Multiple Joins Required)

```
SELECT a.ArtifactID, a.Type, a.Location
FROM Artifacts a
JOIN Projects p ON a.ProjectID = p.ProjectID
WHERE p.Name = 'ERP Implementation';
```

Elasticsearch Query

```
{
  "query": {
    "bool": {
      "must": [
        { "match": { "project.name": "ERP Implementation" } }
      ]
    }
  }
}
```

- Retrieves all relevant project data in a single query.
- No need for complex joins, making it much faster.

Scalability and Performance

Table 12 compares the scalability and performance of SQL databases and Elasticsearch in handling large-scale chatbot queries. While SQL databases are optimized for transactional workloads, Elasticsearch excels in distributed search, fast retrieval, and high-speed data ingestion, making it more suitable for chatbots processing millions of queries.
Due to the limitations of SQL databases in handling large-scale knowledge retrieval, Elasticsearch was chosen as the backbone for the chatbot developed in this thesis. SQL databases rely on structured queries and exact keyword matching, making them slower and less efficient for unstructured or fuzzy searches. In contrast, Elasticsearch offers full-text search, fuzzy matching, NLP-based retrieval, and vector search, ensuring faster and more context-aware responses.Additionally, SQL databases scale vertically, requiring more powerful hardware, whereas Elasticsearch's distributed architecture allows for horizontal scaling, enabling efficient handling of millions of chatbot queries. SQL also depends on JOINs for relational queries, which slows performance, while Elasticsearch eliminates joins by storing related data as nested documents,

Aspect	SQL Database	Elasticsearch
Scalability	Vertical scaling (adding more RAM/CPU to a single machine).	Horizontal scaling (distributed search across multiple nodes).
Read Performance	Slower for complex queries involving multiple tables.	Faster for large-scale searches due to inverted index-based retrieval.
Write Performance	Optimized for transactional workloads (OLTP).	Optimized for high-speed data ingestion.
Data Replication	Replication supported, but slow for large datasets.	Built-in replication and shard-based distribution.

Table 12: Comparison of Scalability and Performance Between SQL Databases and Elasticsearch

significantly improving query speed and flexibility.Furthermore, real-time analytics and high-speed data ingestion make Elasticsearch better suited for dynamic chatbot interactions. Unlike SQL, which lacks native support for vector similarity search, Elasticsearch enables AI-driven, semantic search for intelligent chatbot responses. Given these advantages in query performance, scalability, real-time analytics, and AI integration, Elasticsearch was selected as the superior solution for powering chatbot queries, ensuring speed, accuracy, and efficiency in knowledge retrieval.

4 Data Preparation

Building on the previous chapter, which justified the transition to Elasticsearch and outlined the chatbot's architecture, this chapter focuses on the data preparation process. Effective data preparation is essential for optimizing the chatbot's performance, ensuring that the data is properly structured and indexed to enable accurate and efficient retrieval.

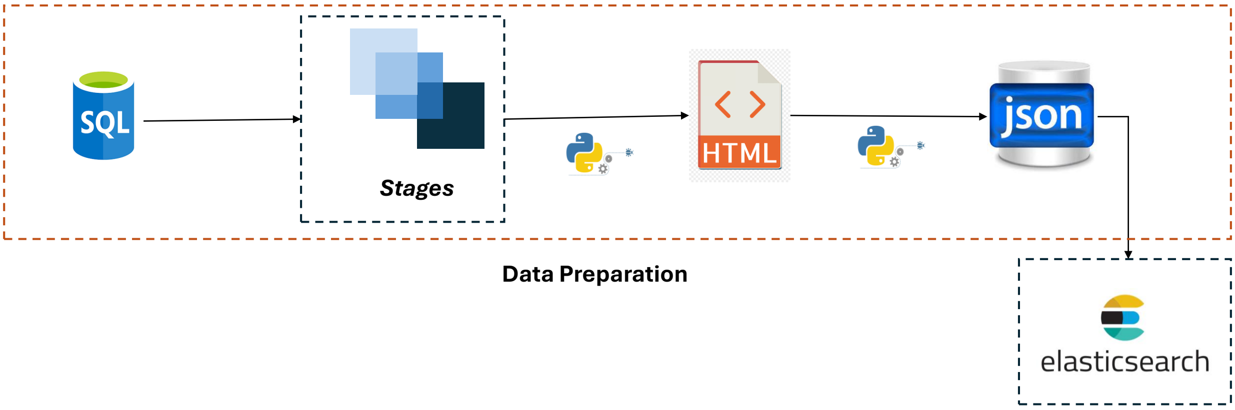

Figure 4: Data preparation Workflow

This chapter explores different methods of migrating data from SQL databases to Elasticsearch, the challenges faced during the migration particularly within the Bosch network and alternative approaches to streamline data integration. One such approach involves web crawling the Stages website to extract relevant data when direct database migration is impractical (Figure 4). A detailed breakdown of the web scraping implementation is provided, covering the extraction of structured HTML content and conversion into JSON format for indexing. Finally, the chapter discusses the data ingestion process, outlining how the extracted data is structured and stored in Elasticsearch, including sharding and distributed indexing to optimize search performance.

4.1 Different Approaches for Migrating Data from SQL to Elasticsearch

Depending on the source, format, and use case, data can be migrated into Elasticsearch from SQL databases or other sources using different techniques. This section explores the major approaches for adding data from a SQL database into Elasticsearch, including direct ingestion, pipeline-based methods, and external tools.

4.1.1 Using Logstash (ETL Pipeline Approach)

Logstash is a powerful tool for ETL (Extract, Transform, Load) operations. It uses a JDBC input plugin that enables direct data fetching from SQL databases.

How It Works:

- Establishes a connection with the SQL database.
- Queries the database at specified intervals (supports scheduling).
- Transforms data if required (filtering, renaming fields, etc.).
- Sends structured JSON documents to Elasticsearch for indexing.

Challenges:

- Performance issues with large datasets.
- Limited flexibility in complex transformations.
- Requires JDBC drivers for each database type.

4.1.2 Using Python (Custom Scripts)

Python provides fine-grained control over data extraction and transformation using libraries like pandas, psycopg2, pymysql, and Elasticsearch-py.

How It Works:

- Connects to the SQL database and executes queries.
- Fetches data in batches for efficient memory usage.
- Converts SQL data into Elasticsearch-compatible JSON.
- Indexes documents into ES using Elasticsearch API.

Challenges:

- Requires scripting knowledge.
- Manual execution unless scheduled with cron jobs or workflow automation tools.

4.1.3 Using Elasticsearch SQL Connector (JDBC River)

JDBC River was an official Elasticsearch plugin for SQL integration (now deprecated). An alternative is using JDBC Importer, which enables SQL-to-ES integration.

How It Works:

- Queries SQL data using a JDBC connection.
- Streams results directly into Elasticsearch.
- Can be scheduled to fetch updates periodically.

Challenges:

- Deprecated official support.
- Requires maintaining external tools like JDBC Importer.

4.1.4 Using Bulk API for Large Data Sets

Elasticsearch Bulk API allows batch indexing of documents, making it the best approach for migrating millions of records efficiently.

How It Works:

- Export SQL data to JSON format in bulk API structure.
- Use the Elasticsearch _bulk API to index data in large chunks.
- Ensures efficient memory and performance handling.

Challenges:

- Requires proper data formatting.
- Not ideal for incremental updates.

4.2 Challenges in Data Migration from SQL to Elasticsearch within the Bosch Network

In the Bosch network, Elasticsearch is located in the Security Layer 2 (SL2) , while the Stages database resides in the Security Layer 4 (SL4) (Figure 5). Due to this segregation of security layers, establishing a direct connection between these two systems is not feasible without lifting security restrictions and modifying firewall settings.

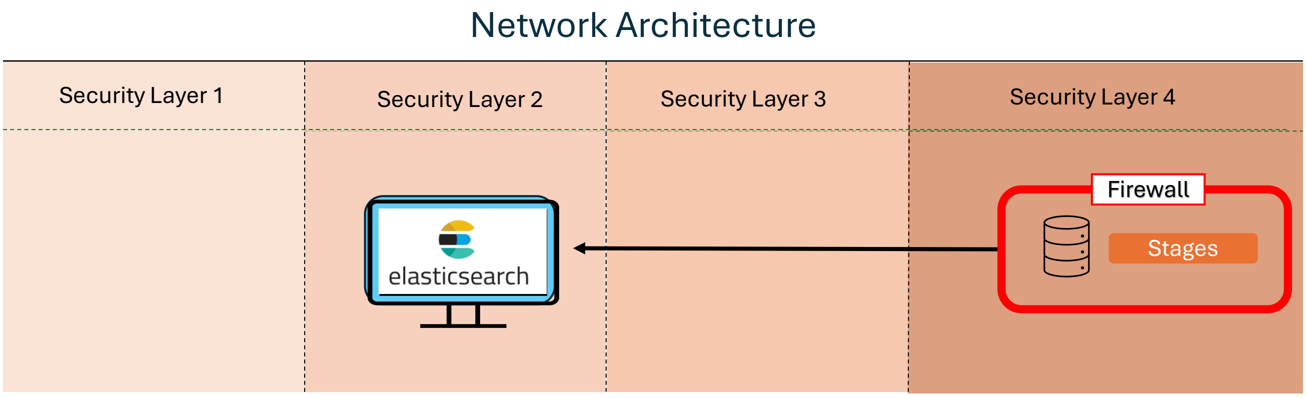

Figure 5: Bosch Network Architecture

The process of obtaining the necessary approvals and implementing these security modifications typically requires two to four months, making it impractical for immediate data migration. Consequently, standard migration methods, such as direct SQL-to-Elasticsearch pipelines, were not applicable in this case. Alternative solutions had to be explored to facilitate data transfer while adhering to Bosch's security policies.

4.2.1 Alternative Approach: Web Crawling Stages Website for Data Addition

Given the security restrictions within the Bosch network, where Elasticsearch is confined to Security Layer 2 (SL2) and the Stages database resides in Security Layer 4 (SL4), establishing a direct connection for data migration was not feasible. This led to an exploration of alternative methods, and web crawling emerged as a potential solution.

Evaluating the Inbuilt Elasticsearch Web Crawler

Initially, the built-in Elasticsearch web crawler was considered as an alternative. This approach is particularly convenient for open-source and publicly accessible websites, as it allows seamless data extraction and indexing without requiring direct database access.

To assess its viability, a test case was conducted using the BSH homepage, where the crawler successfully extracted and indexed content. This demonstrated that Elasticsearch's web crawler is effective for structured and semi-structured data from websites, making it a promising solution.

Challenges with Internal Data Crawling

Despite the successful test on an open-source website, applying the same approach to internal data proved to be infeasible due to security restrictions. Unlike publicly accessible websites, internal systems in Bosch are protected by multiple layers of authentication, firewalls, and access controls. As a result:

- The built-in Elasticsearch web crawler could not establish a connection to crawl and index internal pages.
- Security protocols prevented automated web scrapers from interacting with restricted content, limiting access to only authenticated users demanding an SSO/SSL certificate.
- Even with valid credentials, network security policies restricted automated tools from making repeated requests to extract structured data.

Thus, while web crawling was a viable approach for external sources, it was not a viable solution for internal Bosch data.

Exploring a Local Elasticsearch Web Crawler

Another alternative was considered: deploying the Elasticsearch web crawler on a local machine within the same network segment as the Stages database. This would have allowed:

- Crawling of internal web pages without violating firewall rules.
- Controlled indexing of extracted content into Elasticsearch.

However, this approach was also restricted due to Bosch's security policies, as:

- Elasticsearch could not be installed on local machines due to strict IT governance.
- Local execution of web crawlers was blocked for security reasons, preventing unauthorized data extraction.
- Network segmentation policies prevented direct data transfer between local and server environments.

4.3 Web Crawling Using Selenium for Data Extraction in a Restricted Network Environment

Due to network segmentation and firewall restrictions, conventional methods like JDBC, Logstash, and API-based integrations were blocked. While Elasticsearch's built-in web crawler was considered, internal security policies restricted its use for internal sites. To bypass these limitations while maintaining security compliance, Selenium WebDriver was employed to automate the extraction of structured HTML content from the Stages platform.

This section details the browser automation methodology using Selenium, covering its implementation, functionality, and significance in this thesis. It explains how Selenium automates web interactions, extracts data when direct database access is restricted, and ensures efficient migration of structured content to Elasticsearch. The implementation includes setting up Selenium, configuring drivers, handling dynamic elements, and executing automated scripts. This approach not only overcomes security barriers but also streamlines data collection, preserving document structure and hierarchical relationships.

4.3.1 Detailed Breakdown of the Implementation

This section provides pre-requirements and a step-by-step breakdown of the workflow involved in implementing Selenium-based browser automation for data extraction. The process ensures structured data retrieval from the Stages platform, converting it into a format suitable for ingestion into Elasticsearch while adhering to security constraints. For more detailed information and reference, please refer to APPENDIX I. Below is a generalized workflow detailing each stage of the implementation:

Requirements for Browser Automation Using Selenium

To successfully implement browser automation for data extraction, several technical, environmental, and operational requirements must be met. These requirements can be categorized into the following sections:

- Python 3.x (latest stable version recommended)
- Selenium WebDriver for browser automation
- Additional libraries for handling URLs, files, and logs:
  - hashlib
  - os
  - time

Web Browser and WebDriver:

Since Selenium interacts with web pages through a browser, selecting an appropriate browser along with its corresponding WebDriver is essential for smooth automation. The supported browsers include Google Chrome (preferred for its best compatibility and extensive support), Mozilla Firefox, and Microsoft Edge. Ensuring that the correct WebDriver version matches the installed browser version is crucial to prevent compatibility issues and ensure seamless execution of automation scripts.

1. Defining the Data Extraction Scope

Before initiating the extraction process, it was essential to define the scope of the data to be collected. This included:

- Identifying the target webpages containing the required information.
- Establishing URL filtering criteria to avoid unnecessary or irrelevant pages by fixing a base URL, so that all the other URLs containing the main URL is only extracted.
- Determining the depth of recursion for following links to additional content.
- Deciding on file storage formats to ensure compatibility with later processing.

These steps ensured that only relevant and structured data was collected efficiently.

2. Setting Up Browser Automation

To automate webpage navigation and data retrieval, a browser automation tool was utilized. The key steps involved:

- Configuring the browser driver to interact with webpages dynamically.

- Optimizing browser settings by disabling pop-ups, notifications, and unwanted elements.
- Loading web pages automatically using predefined URLs.
- Handling authentication if required, ensuring that restricted pages could be accessed.

By using an automated browser, the system could mimic human interactions and retrieve dynamically loaded content that traditional scraping methods could not.

3. Extracting Web Page Content

Once a page was loaded, its content was extracted using the following steps:

- Retrieving the entire HTML structure of the page.
- Handling JavaScript-rendered elements by waiting for full page load.
- Extracting text-based information, such as headings, paragraphs, and structured data.
- Identifying and storing hyperlinks for further exploration.
- Capturing PDF document links for later processing.

This step ensured that complete and structured data was retrieved, maintaining its original format and hierarchy.

4. Embedding URLs in Extracted HTML for Reference in the Chatbot

In addition to extracting web pages using browser automation with Selenium, an essential step in the data processing pipeline was the preservation of source URLs. This was achieved by embedding the original URL within the extracted HTML files, allowing the chatbot to reference the actual web pages when retrieving information.

Purpose of Storing URLs in Extracted HTML Since the Stages platform contains dynamic process-related data, it was critical to retain the original source of each extracted document. This enables:

- **Traceability:** Users interacting with the chatbot can access the actual web page corresponding to the retrieved content.
- **Data Validation:** The chatbot can provide a direct reference for users to verify the extracted information.
- **Navigation Support:** Instead of relying solely on indexed content, users can open the original page for further context.

To achieve this, each extracted HTML file was modified to include a canonical reference to the original URL.

5. Recursively Following Links for Comprehensive Data Collection

To expand the data coverage, the automation tool recursively followed links within each page:

- Extracting all internal links and filtering them based on predefined criteria.
- Navigating to unvisited pages and repeating the extraction process.

- Avoiding duplicate processing by tracking visited URLs.
- Limiting the recursion depth to prevent unnecessary navigation.

This approach allowed for systematic exploration of content while maintaining efficiency and control.

6. Handling Special Cases and Error Management

During execution, various challenges were encountered, requiring specific solutions:

- **Dynamically loaded content:** Introduced delays to ensure elements were fully rendered before extraction.
- **Interactive buttons and pop-ups:** Simulated user interactions to access hidden content.
- **Broken or restricted links:** Implemented exception handling to bypass inaccessible pages.
- **Session timeouts:** Managed authentication persistence for seamless navigation.

By incorporating error handling mechanisms, the process became robust and resilient to unexpected issues. These optimizations aimed to make the data extraction pipeline more scalable and efficient.

4.4 Indexing Extracted HTML Files into Elasticsearch Bulk API JSON Format

After extracting HTML files using browser automation, the next step involves converting them into a structured JSON format suitable for efficient indexing in Elasticsearch. This section outlines the conversion process, the required JSON structure, and how it integrates with Elasticsearch's Bulk API for optimal performance.

The Python code used for this conversion process has been included in APPENDIX II for reference. The code extracts key elements from the HTML files, such as title, meta tags, body content, headings, and links, and formats them into a structured JSON format. It also includes error handling and data cleaning steps to ensure that the extracted content is consistent and properly formatted for Elasticsearch indexing. The code leverages Python's BeautifulSoup and JSON libraries to parse HTML and generate the final JSON output.

4.4.1 Importance of Converting HTML to JSON for Elasticsearch

Elasticsearch requires structured JSON documents for indexing and searching. Since raw HTML contains unstructured data, it must be processed and transformed into a structured format to enable efficient querying and retrieval. The primary reasons for this conversion include:

- **Improved Searchability:** HTML content needs to be broken down into distinct searchable fields such as titles, headings, and paragraphs.
- **Metadata Preservation:** Extracting key metadata like URLs, timestamps, and document hierarchy enhances content organization and indexing.
- **Optimized Query Performance:** A well-structured JSON format ensures efficient data retrieval and ranking in Elasticsearch.

4.4.2 Converting HTML to JSON for Elasticsearch Ingestion

The process of converting HTML content into a structured JSON format involves extracting relevant information from web pages and organizing it into a search-friendly structure suitable for Elasticsearch. This transformation ensures that essential metadata, textual content, and structural elements of the webpage are preserved for efficient retrieval and indexing.

1. Extracting Relevant Data from HTML When processing an HTML file, the script extracts key elements, including:

- **Document ID:** Derived from the file name, ensuring a unique identifier for each document.
- **URL Information:** Extracted from the canonical URL, including:
  - Host (domain name)
  - Path (full URL path)
  - Subdirectory levels (first, second, and third-level directories for better categorization)
  - Scheme (HTTP or HTTPS)
- **Metadata Extraction:**
  - Title: Extracted from the `<title>` tag.
  - Meta Description & Keywords: Pulled from `<meta>` tags to provide additional context for search.
- **Body Content:** The main textual content of the webpage is extracted for full-text search.
- **Headings & Links:**
  - Headings (h1, h2, h3, etc.) provide a structured summary of the page.
  - Hyperlinks are collected to analyze internal and external references.
- **Last Crawled Timestamp:** A timestamp (last_crawled_at) is added to indicate when the page was last processed.

4.4.3 Formatting Data for Elasticsearch

Once extracted, the structured data is formatted into a JSON document, making it compatible with Elasticsearch indexing. Each document follows a well-structured format, ensuring consistency and searchability. To efficiently ingest multiple HTML documents into Elasticsearch, the script follows a bulk indexing approach:

1. Prepare JSON documents in bulk format.
2. Index metadata is added before each document to specify the Elasticsearch index (html_data).
3. The documents are written into a single JSON file, where each entry is serialized and written line by line, ensuring compatibility with Elasticsearch's bulk ingestion API.

The Bulk indexing format in Elasticsearch follows a bulk API structure where each document is defined by an index metadata line followed by its corresponding content data.

Example Structure for Bulk Indexing:

```
{ "index": { "_index": "my-index", "_id": "1" } }
{ "field1": "value1", "field2": "value2" }
{ "index": { "_index": "my-index", "_id": "2" } }
{ "field1": "value3", "field2": "value4" }
```

4.4.4 Analysis of Extracted JSON Data for Elasticsearch Bulk Indexing

As part of the web crawling and data extraction process, a total of 2,060 structured JSON documents were created for Stages website and indexed into Elasticsearch. Each JSON entry corresponds to a unique webpage from the Stages platform, containing structured metadata, content, and navigational references. Following is an analysis of the extracted data and its significance for further use in search, retrieval, and chatbot integration.

Following is an example of a single JSON index for created for Elasticsearch:


```
{"index": {"_index": "html_data", "_type": "_doc"}}
{"id": "https___stages.bshg.com_stages___workspace_672__vv_process_activity
__P0gB4Leh_FW1lP-W-Gx7kw",
"url": "https://stages.bshg.com/stages/#/workspace/672/_vv/process/activity/
_P0gB4Leh_FW1lP-W-Gx7kw",
"canonical_url": "https://stages.bshg.com/stages/#/workspace/672/_vv/process
/activity/_P0gB4Leh_FW1lP-W-Gx7kw",
"url_host": "stages.bshg.com",
"url_path": "/stages/#/workspace/672/_vv/process/activity
/_P0gB4Leh_
FW1lP-W-Gx7kw",
"url_path_dir1": "/stages",
"url_path_dir2": "/#",
"url_path_dir3": "/workspace",
"url_port": "80",
"url_scheme": "https",
"title": "Set up problem solving team to process the
quality issue - GDE-ED e/dPDP",
"meta_description": "N/A",
"meta_keywords": "N/A",
"body_content": "Set up problem solving team to process the quality issue
- GDE-ED e/dPDP\nGDE-ED e/dPDP\nProcess Models\nProblem
solving\nProcess\nCollect input at ED level\nActivity\nPS Method
and Sponsor defined\nDecision\nInform about decision\nActivity\nSet up
problem solving team to process the quality issue\nActivity\nDocument the
quality issue with TRC and MRC\nActivity
\nRegular reporting of quality issues\nActivity\nImplement findings within
organization\nActivity\nUser\nHelp\nLog out\nProcess\nProcesses and
Activities\nSupporting Processes and Applied Methods\nProblem Resolution
Management\nProblem solving\nedPDP 3.2.0\nSet up problem solving team to
```

process the quality issue\nFlow\nTable\nGrid\nZoom\n100%\nProblem solving\nProblem solving\nSet up problem solving team to process the quality issue\nSet up problem solving team\nto process the quality issue\nInform about decision\nInform about decision\nCollect input at ED level\nCollect input at ED level\nPS Method and Sponsor defined\nPS Method and Sponsor...\nRegular reporting of quality issues\nRegular reporting of quality\nissues\nImplement findings within organization\nImplement findings within\norganization\nDocument the quality issue with TRC and MRC\nDocument the quality issue\nwith TRC and MRC\nDescription\nThe department head / sponsor announces officially a task force leader for the problem solving process and informs GQM-EDL that a quality issue is processed.\nBased on decision matrix on Problem Solving method, 8D leader / PSS leader / project leader call meeting for starting of project , he has to use 8D methodology to document the overall results. He also\ninvite potential team members as a part of problem solving team.\nRoles\nResponsible\n3\n8D Project Leader\nProblem Solving- Sponsor\nQuality Engineer (Monitoring)\nPhases\nPhases\n2\nE6 Produce Pilot Series\nSeries Production\nActivity\nNone\nWork Products\nInputs\n1\n8D leader and sponsor\nOutputs\n2\nPSS Sharepoint\n8D Problem Solving\nexisting or initiated\nFind details in\nNone\nGuidance\nNone",
"links": ["#/workspace/672/_vv/process/activity/_8qmDUDTDFLWTauYRBmYwYw",
"https://doc.stages.digital/doku.php?id=710:start",
"https://stages.bshg.com/stages/#/workspace/672/_vv/process/activity/
_YsD3MLq8_FW1lP-W-Gx7kw",
"#/workspace/672/_vv/process/role/_LYTdwJHHGAyR3rLBkP_XOw",
"#/workspace/672/_vv/process/artifact/__iFPgFK2My-nWdjROODtWSA",
"#/workspace/672/_vv/process/artifact/_sfv4cFUQ_Fq1lP-W-Gx7kw"],
"headings": ["GDE-ED e/dPDP", "Process Models", "Problem solving", "Process",
"Collect input at ED level", "Activity", "PS Method and Sponsor
defined", "Decision",
"Inform about decision", "Activity", "Set up problem solving team to
process the quality issue",
"Activity", "Document the quality issue with TRC and MRC", "Activity",
"Regular reporting of quality issues", "Activity", "Implement findings
within organization",
"Activity", "User", "Help", "Log out", "Set up problem solving team to
process the quality issue",
"Flow", "Table", "Grid", "Zoom", "100%", "Description", "Roles", "Responsible",
"8D Project Leader", "Problem Solving- Sponsor", "Quality Engineer (Monitoring)",
"Phases", "Phases", "E6 Produce Pilot Series", "Series Production", "Activity",
"Work Products", "Inputs", "8D leader and sponsor", "Outputs", "PSS Sharepoint",
"8D Problem Solving", "Find details in", "Guidance"],
"last_crawled_at": "2024-12-09T14:08:32.127395"}

Overview of Extracted Data Each JSON document represents a single webpage, preserving its text content, metadata, structure, and relationships. The extracted data enables:

- Efficient search and retrieval through Elasticsearch queries.
- Structured organization of technical documents and process models.
- Link-based navigation to related pages and external references.
- Support for chatbot responses by associating content with its original context.

4.4.5 Key Data Elements and Their Relevance

Metadata Extraction

In Table 13, The metadata fields provide essential contextual information about each document.

Field	Description	Purpose
`"id"`	Unique identifier based on URL	Ensures each document is uniquely indexed
`"url"`	Original webpage URL	Provides a direct reference for retrieval
`"canonical_url"`	Standardized URL format	Helps avoid duplicate indexing
`"url_host"`	Domain of the extracted page	Useful for filtering by source
`"url_path"`	Path component of the URL	Allows structured navigation within the website
`"url_scheme"`	Protocol used (e.g., HTTP, HTTPS)	Helps manage site security and accessibility
`"url_port"`	Port used in the request (default 80 for HTTP)	Useful for network and security checks
`"title"`	Webpage title	Used for search ranking and document identification
`"meta_description"`	Short description from meta tags	Useful for displaying previews in chatbot responses

Table 13: Metadata Fields and Their Purpose

Use Case:

- Full-text search on titles and descriptions allows quick document retrieval.
- URL components can be used for domain-specific filtering and grouping.
- Canonical URLs prevent duplicate indexing of similar content.

Content Extraction

In Table 14,The body content field captures the main textual content of the page, including headings, instructions, and descriptions.

Field	Description	Purpose
"body_content"	Extracted raw text from the webpage	Enables full-text search and chatbot responses
"headings"	List of extracted headings (H1, H2, H3, etc.)	Preserves document structure for hierarchical search

Table 14: Content Extraction Fields

Use Case:

- **Chatbot integration:** Allows searching within structured content for context-aware responses.
- **Topic classification:** Headings help categorize documents for better search filtering.
- **Process documentation:** Important for retrieving step-by-step procedures.

Process and Workflow Information

The JSON captures workflow-related data(Table 15), which is crucial for process documentation and automation.

Field	Description	Purpose
"roles"	List of user roles related to the process	Helps in user-specific document filtering
"phases"	Development phases extracted from the document	Supports phase-based document retrieval
"work_products"	Inputs and outputs associated with the process	Enables workflow analysis and optimization

Table 15: Workflow-Related Fields

Use Case:

- Process tracking: Users can retrieve documents related to specific workflow stages.
- Role-based search: A chatbot can filter results by user role (e.g., engineer, project manager).
- Work product documentation: Helps in tracking process inputs and expected outputs.

Internal and External Links

Extracting hyperlinks allows for relationship mapping between documents (Table 16).

Field	Description	Purpose
"links"	List of extracted hyperlinks from the page	Helps in navigational search and related document suggestions
"pdf_links"	Links to downloadable PDFs referenced in the document	Enables extraction of technical documents

Table 16: Link-Related Fields

Use Case:

- Network of related documents: Useful for chatbot-suggested related content.
- Navigational search: Improves content discovery across linked documents.
- PDF extraction: Supports retrieving technical guides and reference materials.

Timestamp and Crawling Information

The timestamp allows tracking when the content was last updated (Table 17).

Field	Description	Purpose
`"last_crawled_at"`	Timestamp of data extraction	Helps in version control and data freshness checks

Table 17: Timestamp Information

Use Case:

- Keeping content up to date: Helps identify stale or outdated documents.
- Indexing new documents periodically: Enables scheduled content refresh

The data provides a rich foundation for chatbot integration, workflow documentation, and process optimization. With further enhancements like semantic search and periodic updates, this approach significantly improves knowledge retrieval and user interaction.

4.5 Data Ingestion (JSON) into Elastic Search

This section discuss the process of bulk ingestion using Python and Elasticsearch's REST API, focusing on efficiently loading 2,060 indexes from a JSON file. Each index corresponds to a single web page with a unique URL, including metadata such as the page title, body content, and timestamp (Figure 6).

Elasticsearch allows users to ingest and index data through multiple approaches, depending on dataset size and ingestion efficiency. Two primary methods include:

1. **Direct Upload** – Uploading a JSON file by dragging and dropping it into Kibana or Elasticsearch UI.
2. **Bulk Ingestion** – Using Python and Elasticsearch's REST API to efficiently index large datasets.

This section provides an overview of both methods, considering the scenario where 2,060 indexes from a JSON file (each representing a unique web page) need to be ingested.

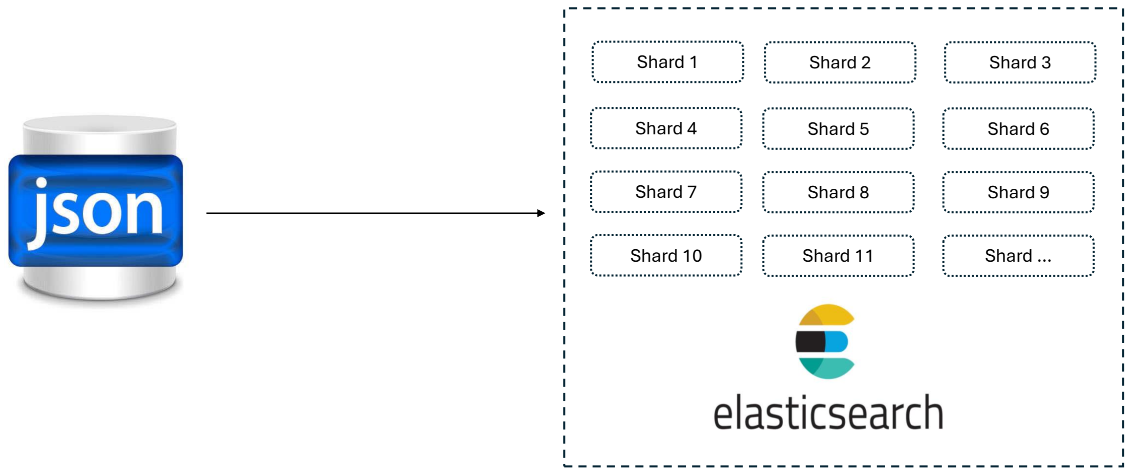

Figure 6: Data Ingestion into Elastic Search

Direct Upload (Drag and Drop in Kibana UI)

Elasticsearch provides a drag-and-drop upload feature within Kibana, allowing users to upload structured JSON, CSV, and other supported formats.

- **Dataset Size:** Suitable for small datasets ($\leq$100MB).
- **Use Case:** When users prefer a no-code solution for quick ingestion into Elasticsearch.
- **Requirements:** Data should be well-structured and formatted according to Elasticsearch expectations.

Bulk Ingestion Using Python and REST API

Bulk ingestion via the Elasticsearch Bulk API enables large-scale indexing by processing multiple documents in batches.

Applicability

- **Dataset Size:** Recommended for large datasets exceeding 100MB.
- **Use Case:** When handling thousands of documents that need to be indexed efficiently.
- **Flexibility:** Allows full control over index mappings, transformations, and data processing.

4.5.1 Re-Indexing in Elasticsearch

Re-indexing is a process in which data is copied from an existing index (source index) to a newly configured index (destination index). This operation is necessary when modifying mappings, adjusting vector dimensions, optimizing performance, or restructuring data to enhance search efficiency. Since dense vector dimensions play a crucial role in similarity searches, re-indexing allows for an increase in dimensionality, enhancing retrieval accuracy and overall search quality.

Modifying Dense Vector Dimensions

Dense vector fields are essential for semantic search, image retrieval, and AI-driven recommendations. Adjusting the dims parameter in Elasticsearch improves accuracy at the cost of increased storage and computation.

The following configuration increases the dense vector dimensions from 256 (default) to 512, 1024, or 2048 based on computational needs.

Create a New Index with Updated Vector Dimensions

```
PUT new_index_name
{
  "mappings": {
    "properties": {
      "your_dense_vector_field": {
        "type": "dense_vector",
        "dims": 512  // Adjust as required (512, 1024, or 2048)
      },
      "other_field": {
        "type": "text"
      }
    }
  }
}
```

Re-Index Data from the Old Index to the New One

```
POST _reindex
{
  "source": {
    "index": "old_index_name"
  },
  "dest": {
    "index": "new_index_name"
  }
}
```

5 Implementation

After successfully preparing the data and ingesting it into Elasticsearch, the next step is to develop the chatbot capable of retrieving relevant information from the database and responding to user queries. This section outlines the process of chatbot development, including the system architecture, interaction with Elasticsearch, integration with the custom Language Model (LLM), and overall system flow.

5.1 Front-End Design of the Chatbot

The chatbot's front-end interface is built using HTML, CSS, and JavaScript, ensuring a responsive, user-friendly, and visually appealing experience. The design focuses on simplicity and accessibility, allowing users to interact with the chatbot seamlessly. Key considerations include intuitive navigation, clear typography, and real-time interaction capabilities to enhance usability.

The Front-end design in Figure 7 is intended solely for testing purposes. This prototype serves as a foundation for functionality validation, while further development options and enhancements are explored in detail in later sections.

The code for the frontend design of the chatbot has also been included in APPENDIX III for reference. It covers the layout, user interface elements, and the logic for handling user queries and displaying chatbot responses. The code is designed using HTML, CSS, and JavaScript, integrated with a Flask backend to facilitate seamless communication with the chatbot's processing engine.

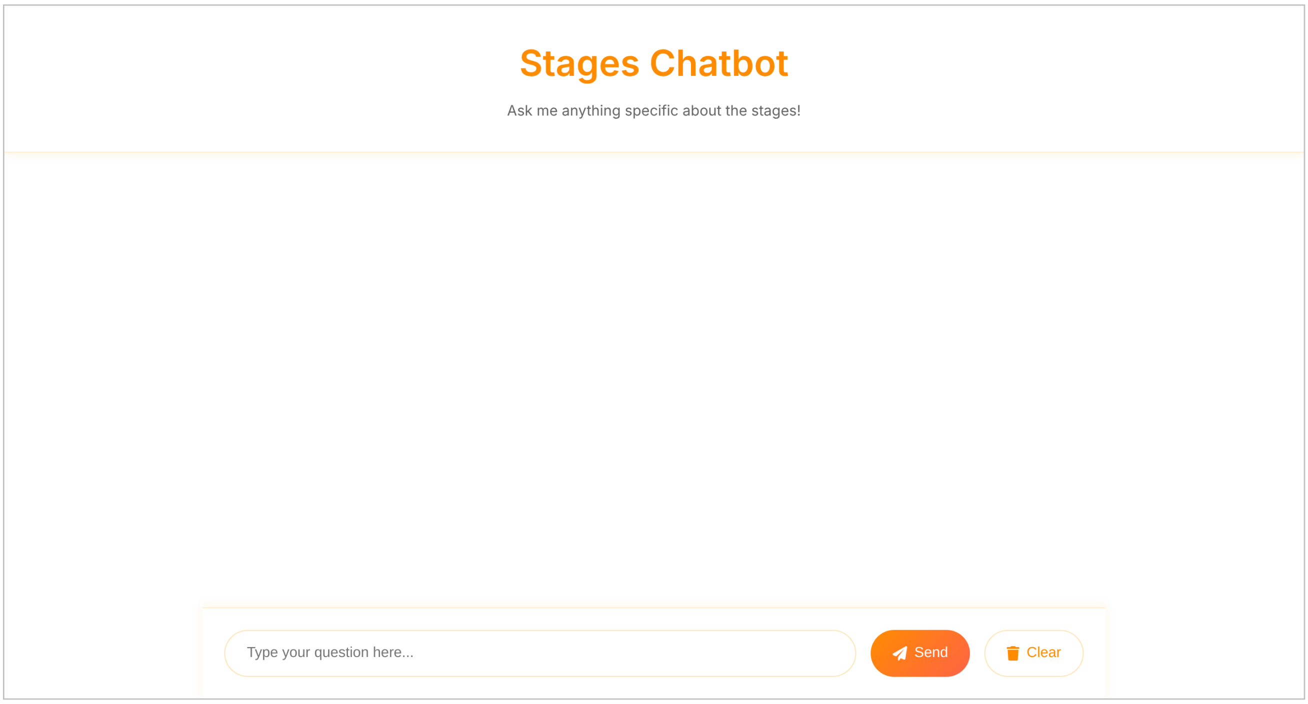

Figure 7: Chatbot Test UI

5.1.1 Layout and Structure

The chatbot UI consists of three main sections:

1. **Header:** Displays the chatbot's name and tagline.

2. **Chat Container:** Shows the conversation history.
3. **Input Area:** Allows users to type queries and submit them.

The header remains fixed at the top and dynamically minimizes when a user starts interacting, maximizing screen space for conversations.

Color Scheme and Branding: The interface follows the BSH colour priority, incorporating shades of orange (#ff8c00) and white (#fff) with subtle shadows for a clean and modern look. The chatbot messages have a contrasting background to differentiate between user and bot responses.

Chat Interface

- Messages are structured with headers and icons to visually separate bot and user responses.
- Animated fade-in effects make messages appear smoothly.
- Bullet points and strong/italic formatting enhance readability.

Interactive Features

- **Clear Button:** Resets the conversation.
- **Send Button:** Submits user queries.
- **Enter Key Support:** Allows users to send messages by pressing Enter.
- **Loading Indicator:** Displays a "Thinking..." message while waiting for a response.

Responsive Design

- On smaller screens, buttons stack vertically for easy mobile interaction.
- The chat messages adapt dynamically, ensuring a seamless user experience.

The overall design ensures that the chatbot remains user-friendly, professional, and visually aligned with BSH's branding guidelines, making it an effective proof-of-concept for the chatbot interface.

5.2 Backend Architecture of the Chatbot

The chatbot's backend is built using Flask, a lightweight web framework, and integrates with Elasticsearch and an LLM (Large Language Model) API to process user queries. The backend is designed to efficiently handle requests, retrieve relevant information, and generate responses (Figure 8).

The Python code for the chatbot development has been added to APPENDIX IV for reference. The code has been screened to remove company credentials and sensitive information to comply with corporate data protection and privacy policies. It includes the logic for query handling, Elasticsearch integration, and response generation using the LLM API.

5.2.1 Flask-based Web Application

Flask is a lightweight web framework that plays a crucial role in developing the chatbot by handling user interactions, managing API requests, and integrating backend services. In chatbot development, Flask primarily serves as the bridge between the user and the chatbot logic, facilitating smooth communication. Here are the key roles Flask plays:

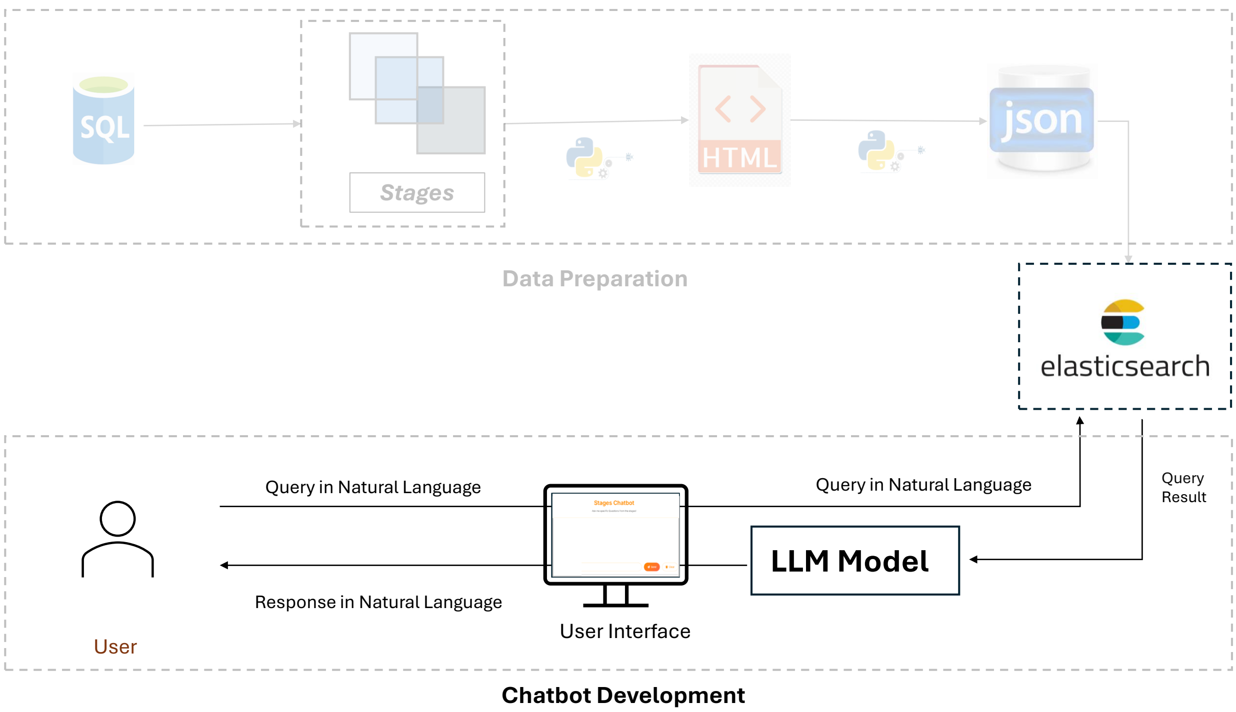

Figure 8: Chatbot Development architecture

Handling HTTP Requests and Responses

- Flask processes user queries via API endpoints and sends appropriate responses.
- Example: A user sends a question through the chatbot interface, and Flask routes it to the appropriate function.

Routing and API Management

- Flask allows defining multiple routes to serve different functionalities of the chatbot.
- Example: The chatbot has separate endpoints for user queries (`/chat`), resetting conversations (`/reset`), and retrieving chat history.

Integrating with Backend (LLM, Elasticsearch, etc.)

- Flask facilitates integration with external services such as databases, AI models, and search engines.
- Example: A chatbot can use Flask to fetch data from an Elasticsearch database to provide answers.

Rendering Web Interfaces

- Flask can serve HTML pages where users interact with the chatbot.
- Example: Displaying a chatbot UI using Flask's `render_template` function.

Session Management for Contextual Conversations

- Flask can maintain a session for users, allowing the chatbot to remember past interactions.
- Example: Maintaining a conversation history.

5.3 Chatbot Workflow

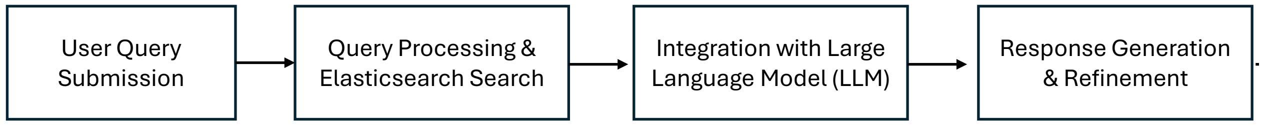

Figure 9: Chatbot Workflow

The chatbot follows a structured flow to ensure accurate and meaningful responses(Figure 9). The process can be broken down into the following steps:

1. **User Query Submission:**
   - The user enters a query into the chatbot's interface.
   - The query is sent as a request to the Flask-based backend.
2. **Query Processing & Elasticsearch Search:**
   - The chatbot processes the query and sends it to Elasticsearch.
   - Elasticsearch searches its indexed data and retrieves relevant documents.
   - The most relevant results as per the size specified (e.g., top 3 matches) are extracted and returned.
3. **Integration with Large Language Model (LLM):**
   - The retrieved Elasticsearch results and the original user query are combined into a structured prompt.
   - This prompt is sent to the LLM API for further processing.
   - The LLM compares the query with the search results and refines the response to ensure meaningfulness and coherence.
4. **Response Generation & Refinement:**
   - The LLM generates an answer based on the given query and contextual information from Elasticsearch.
   - The response is structured to be user-friendly and informative as mentioned in the propmpt.
   - References (Web page URLs) are included for further validation.
5. **Delivery to the User:**
   - The chatbot sends the generated response back to the user interface.
   - The user sees the final answer along with relevant references (if available).

5.4 Step-by-Step Chatbot Processing

Once a user submits a query through the front-end interface, the chatbot follows a structured workflow to process the request, retrieve relevant information, and generate a meaningful response. The process involves multiple stages, including query intake, data retrieval, response generation, and final output delivery.

The following sections provide a detailed breakdown of each step, starting with how the chatbot receives and processes the user query before retrieving relevant information from Elasticsearch.

Required Libraries

The script begins by importing essential modules:

- `os`: Handles file paths and environment variables.
- `Flask`: Manages the web server and handles HTTP requests.
- `Elasticsearch`: Provides an interface to query the Elasticsearch index.
- `requests`: Sends HTTP requests to the LLM API.
- `dotenv`: Loads environment variables from a .env file.
- `argparse`: Parses command-line arguments for dynamic

5.4.1 Query Intake and Initial Processing

Once a user submits a query through the front-end interface, the chatbot's Flask-based backend processes the request using the `/chat` endpoint. This step is crucial as it ensures the query is correctly received, analyzed, and routed for further processing.

Step 1: Receiving and Extracting the User Query

The chatbot listens for POST requests at the `/chat` endpoint, where it extracts the query from the JSON payload

- A variable (`user_query`) holds the input provided by the user.
- If no query is received, the chatbot will return an appropriate response.

Step 2: Handling Greetings

To make the chatbot more interactive, it first checks if the query is a simple greeting (e.g., "Hello", "Hi", "Hey" etc.). If the input matches one of these predefined keywords, the chatbot responds with a friendly message:

- The query is converted to lowercase and whitespace is removed to ensure accurate matching.
- If the input is a greeting, the chatbot immediately returns a pre-defined response without further processing.

Step 3: Preparing for Further Processing

If the query is not a simple greeting, it proceeds to fetch relevant information from Elasticsearch. This step marks the transition to the retrieval phase, where the chatbot searches for relevant knowledge before generating a response.

This structured intake process ensures that all user queries are processed efficiently, allowing the chatbot to differentiate between casual interactions and information requests before proceeding to data retrieval.

Installing and Initializing Elasticsearch in Python

Before Processing queries, Elasticsearch must be initialized in Python. The official elasticsearch library allows to connect and interact with an Elasticsearch cluster.

5.4.2 Querying Elasticsearch for Relevant Results

The chatbot queries Elasticsearch to find documents that match the user's input. The multi_match query searches across multiple fields using text relevance scoring.

```
def get_elasticsearch_results(query):
    """
    Retrieves relevant documents from Elasticsearch based on user query.
    """
    es_query = {
        "query": {
            "multi_match": {
                "query": query,  # User input query
                "fields": ["body_content", "title", "headings"],
                "type": "best_fields",  # Ranking method
                "tie_breaker": 0.3
            }
        },
        "size": 3  # Retrieve top 3 most relevant results
    }

    # Execute the search query in Elasticsearch index "chatbot_data"
    result = es_client.search(index="chatbot_data", body=es_query)

    # Extract only relevant hits (documents that matched the query)
    return result["hits"]["hits"]
```

Where,

- `query`: The user's input is processed and searched across multiple fields.
- `multi_match`: Searches for matches across multiple text fields.
- `fields`: Specifies the fields to search in (body_content, title, headings).
- `type`: best_fields: Uses the field with the best relevance score to rank documents.
- `tie_breaker`: Helps balance relevance scores from multiple fields.
- `size`: Limits the number of results returned (in this case, "size": 3).

5.4.3 Elasticsearch Rank Search Results Mechanism

Elasticsearch ranks search results using the BM25 (Best Matching 25) algorithm. BM25 assigns a relevance score to each document based on:

1. Term Frequency (TF): How often the query term appears in a document.
2. Inverse Document Frequency (IDF): How rare a term is across all documents.
3. Field Length Normalization: Penalizes long documents where the term appears frequently.

Example: Ranking Search Results

Consider a Elastic search index with three documents (Table 18):

Document ID	Title	Body Content
1	"Chatbot Introduction"	"A chatbot is an AI-driven tool that helps users with queries."
2	"Chatbot Capabilities"	"A chatbot can process natural language and retrieve information."
3	"Elasticsearch Basics"	"Elasticsearch helps in searching text efficiently."

Table 18: Example documents in Elasticsearch index

User Query: "Chatbot AI"

When the chatbot searches for "Chatbot AI", Elasticsearch calculates a relevance score for each documen)t (Table 19):

Document ID	Term Matches	Score (BM25)
1	"Chatbot" (Title) + "AI" (Body)	1.8
2	"Chatbot" (Body)	1.4
3	"Elasticsearch" (Body)	0.6

Table 19: Search results for query "chatbot AI"

- Document 1 ranks highest because the term "Chatbot" appears in the title (which carries more weight).
- Document 2 ranks second since "Chatbot" appears in the body.
- Document 3 ranks lowest because it contains no matching terms.

Thus, Elasticsearch returns:

```
[
    {"_id": "1", "_score": 1.8, "_source": {"title": "Chatbot Introduction",
    "body_content": "..."}},
    {"_id": "2", "_score": 1.4, "_source": {"title": "Chatbot Capabilities",
    "body_content": "..."}}
]
```

5.5 Communication with the LLM API

The chatbot leverages a Large Language Model (LLM) API to generate intelligent responses based on user queries. The LLM processes natural language inputs, understands context, and produces human-like responses. The chatbot acts as an interface between users and the LLM, ensuring a smooth and structured conversation flow.

5.5.1 Integration of Internal and External LLMs in a Chatbot

When integrating Large Language Models (LLMs) into a chatbot, the approach varies depending on whether the LLM is internally hosted (self-hosted or private API) or a External/ public API (such as OpenAI's ChatGPT). In the case of an internal LLM, the system requires dedicated infrastructure and direct access control to manage data privacy and performance optimization. Conversely, when utilizing a public LLM API, the chatbot relies on external service providers for processing, benefiting from readily available models but often facing latency, API rate limits, and data security considerations.

Internal LLMs: Custom Class for API Communication

For self-hosted or private LLMs, a custom class is required to handle communication. Internal models often require:

- Authentication and API key handling
- Session management to maintain context across multiple queries
- Custom request formatting to align with the internal LLM's API structure
- Error handling to manage API timeouts or failures

How It Works

- A custom Python class is implemented to handle interactions.
- The class stores the API endpoint, session keys, and model configurations.
- If the session expires, the class resets and retries the request.

Example Workflow

1. User query is passed to the custom LLM class.
2. The class formats the request and sends it to the internal LLM API.
3. If a session is required, it is stored and updated in subsequent calls.
4. The LLM processes the request and returns a structured response.

This approach ensures full control over API behavior, session handling, and data privacy when working with on-premises or restricted access LLMs.

General LLMs: Simple API Calls for Chat Completion

Publicly available LLMs like OpenAI's ChatGPT, Anthropic Claude, or Google Gemini provide pre-built API functions for chatbot integration. These services:

- Require minimal setup (just an API key and endpoint)
- Offer built-in session/context management
- Provide structured and optimized responses without additional configuration

How It Works

- A single API call (e.g., chat/completions) is made with a user prompt.
- The API automatically handles context, token limits, and response formatting.
- The chatbot directly retrieves the AI-generated response and returns it to the user.

Example Workflow

1. User submits a query.
2. The chatbot sends the query to OpenAI's chat/completions endpoint.
3. The API returns a structured response without requiring custom session handling.

Since these models are hosted and optimized by the provider, the chatbot does not need to handle model execution, training, or maintenance.

Table 20 presents a comparison of the key differences between internal and general LLMs.

Feature	Internal LLMs (Custom API)	General LLMs (Public APIs)
Setup Complexity	Requires custom API class	Simple API call
Session Handling	Needs explicit session key management	Managed by provider
Authentication	Custom authentication mechanism	API key-based authentication
Customization	Full control over model settings	Limited customization
Privacy & Security	Fully private	Data may be processed externally

Table 20: Comparison of Internal and General LLMs

The subsequent steps are outlined following the internal LLM integration approach, ensuring full alignment with BSH's compliance policies and security standards. This approach is specifically designed to safeguard data privacy, maintain regulatory compliance, and provide operational control over AI-generated responses. By using internally hosted LLMs, including different GPT models from BSH AIM and Azure OpenAI, the system ensures that sensitive information remains within BSH's secure infrastructure, minimizing reliance on external APIs and aligning with internal governance requirements.

Additionally, the outlined steps incorporate best practices for model deployment, session management, and API security, guaranteeing that the chatbot functions efficiently, reliably, and within BSH's defined compliance boundaries.

5.6 Prompt Instructions for the LLM

The instructions provided to the LLM serve as a guideline for response generation, ensuring that the chatbot maintains a consistent, professional, and structured approach in answering user queries. Since LLMs generate text based on probabilistic models, clear instructions help prevent hallucination, improve accuracy, and maintain coherence [6].
To enhance response relevance, the user's question and the retrieved data from Elasticsearch are provided to the LLM along with the prompt. This ensures that the model generates responses based on retrieved

factual data, aligning with the chatbot's knowledge base while maintaining clarity and contextual accuracy. The instructions focus on:

- **Data-driven responses** - Ensuring answers are derived only from retrieved Elasticsearch data.
- **Professional tone and structure** - Making responses polite, readable, and logically formatted.
- **Avoiding unnecessary content** - Preventing hallucinated, speculative, or off-topic answers.

Key Elements of the Instructions

The instructions provided to the LLM cover multiple aspects to control how the chatbot processes and delivers responses.

a) Context Awareness

- The LLM is explicitly instructed to only generate responses based on the provided data to avoid hallucination from the LLM.
- It must not hallucinate or invent information that is not present in the retrieved Elasticsearch results.

b) Professional and Formal Tone

- The LLM is instructed to maintain a formal and professional tone.
- The response must not include unnecessary greetings, emojis, or informal language.
- Example: Instead of saying "Hey there! Here's your answer...", the chatbot directly start with the response.

c) Structured Formatting

- The response should be structured using headings, bullet points, or numbered lists.
- The chatbot should keep answers concise and focused, avoiding unnecessary repetition or fluff.

d) Consistency with Original Data

- The LLM is instructed to use the same keywords and terminology found in the original source data.
- This ensures consistency between the chatbot's response and the user's expectations.

e) Referencing

- The chatbot should reference data sources where applicable.
- If multiple sources exist, the LLM should display only the most relevant canonical URL.

f) Handling Missing Information

- If the requested information is not found in the provided data, the LLM must clearly state that it is not available rather than making assumptions.

Example of Instructions Given to the LLM

For better clarity of the code a separate function in python `get_instructions()` defines these instructions:

```
def get_instructions():
    """
    Returns the static instructions for the LLM prompt.
    :return: A string containing the instructions.
    """
    return """Instructions for processing this data and question:
Data:
{context}

Please follow these rules:

- Display only one canonical URL of the context where the majority of the data
is used to answer the question.
- Be a professional Bot who greets and treats others politely, but don't greet
before and after the answer, and don't use smileys.
- Answer ONLY from the given data, DO NOT hallucinate.
- Use the same keywords found in the provided data whenever possible.
- Carefully review the unstructured data.
- Identify the key points, facts, or themes relevant to the question.

### Structure Your Response
- Provide your answer in a well-structured, easy-to-read format.
- Use headings, bullet points, or numbered lists if necessary.
- Keep each section focused and concise.

### Professional Tone and Clarity
- Write in a clear, formal, and professional tone.
- Use correct grammar, spelling, and punctuation.
- Avoid unnecessary jargon and explain specialized terms when needed.
### Cite or Reference Data
- Provide only one relevant canonical URL if available.

### Final Answer Format
- Begin with a short introduction or summary.
- Follow with the main body (organized under headings or bullet points).
- Conclude with a concise summary or final recommendation.
- If the requested information is not found in the data, clearly state
that it is not found.
```

```
Question:
{question}
"""
```

This ensures that every response is accurate, relying strictly on retrieved knowledge, well-structured, making it easy to read and follow, professional, avoiding informal language or unnecessary greetings, and error-free, maintaining grammatical correctness and clarity.

Example of How Instructions Shape the Response

Input Query : "What is the description of the process for setting up a problem-solving team to process the quality issue?"

Retrieved Data from Elasticsearch :

> *Description\ nThe department head / sponsor announces officially a task force leader for the problem solving process and informs GQM-EDL that a quality issue is processed.\ nBased on decision matrix on Problem Solving method, 8D leader / PSS leader / project leader call meeting for starting of project , he has to use 8D methodology to document the overall results. He also\ ninvite potential team members as a part of problem solving team.*

LLM Response (Following Instructions) :

Key Stages in the Stages Process

The description of the process is as follows:

1. **Announcement of Task Force Leader:**
   - The department head or sponsor officially announces a task force leader for the problem-solving process.
   - The sponsor informs **GQM-EDL** that a quality issue is being processed.
2. **Initiation of Problem-Solving Process:**
   - Based on the decision matrix on the Problem Solving method, the **8D leader**, **PSS leader**, or **project leader** calls a meeting to initiate the project.
3. **Documentation:**
   - The leader must use the **8D methodology** to document the overall results.
4. **Team Formation:**
   - The leader invites potential team members to be part of the problem-solving team.

The structured approach outlined in this chapter provides a comprehensive roadmap for developing an intelligent chatbot by integrating Elasticsearch for knowledge retrieval, web crawling for data extraction, and large language models (LLMs) for response generation. Each step, from data migration strategies to the development of an NLP-driven chatbot, was carefully designed to address the challenges of retrieving structured and unstructured information efficiently within a restricted network environment.

Through the migration of SQL-based data into Elasticsearch, the system gains the advantage of flexible indexing, full-text search, and real-time query handling, which are critical for chatbot interactions. The browser automation approach using Selenium overcomes security constraints, ensuring data accessibility even when direct database integration is not feasible. Additionally, the chatbot development phase integrates frontend, backend, and prompt engineering, allowing seamless user interaction with the knowledge base

With the Implementation now established, the next section will focus on testing and evaluation, assessing how well the system performs in real-world queries, retrieval accuracy and processing efficiency by Calculating BLUEscore and BERTscore. The results will offer insights into the chatbot's strengths, limitations, and areas for improvement, ensuring its effectiveness in enterprise environments.

6 Testing and Results

The development of the chatbot aimed at providing efficient and contextually relevant responses to user queries by integrating Elasticsearch for information retrieval and a Large Language Model (LLM) API for response generation. The chatbot successfully processed queries, retrieved relevant documents, and generated structured responses in real-time.

During the testing phase of the chatbot, several key functionalities were observed and documented. However, due to compliance restrictions, only a limited number of images could be provided as supporting evidence.

One of the fundamental capabilities demonstrated by the chatbot was its ability to recognize user interactions and respond with a friendly and contextually appropriate greeting. This feature enhances user engagement and provides a welcoming interface, making interactions feel more natural and intuitive.

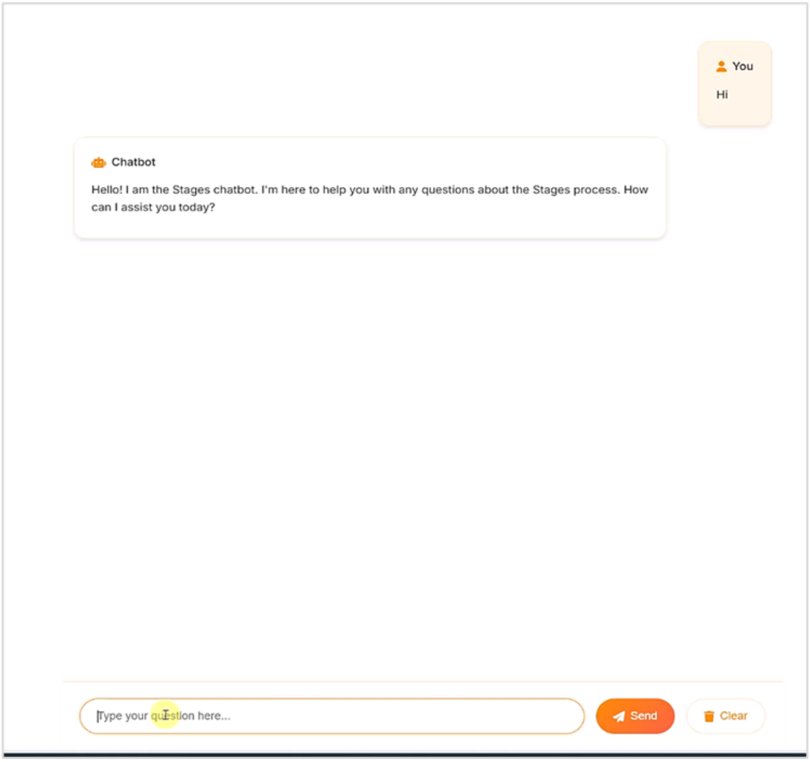

Figure 10: Greeting Response from the chatbot

Additionally, the chatbot was tested with a series of basic definition-type queries, where users asked for explanations of specific terms and concepts. The chatbot successfully retrieved relevant information and provided concise yet comprehensive responses based on the indexed data sources. This functionality ensures that users can quickly access fundamental knowledge without needing to navigate through extensive documents manually.

These initial observations highlight the chatbot's effectiveness in handling simple queries and creating a user-friendly experience. While only a limited number of screenshots could be included in the thesis due to data compliance restrictions, the documented testing outcomes provide strong evidence of the chatbot's ability to facilitate efficient information retrieval and interaction.

6.1 Chatbot Response Showcase

The following examples demonstrate the chatbot's ability to generate clear, structured, and informative responses based on user queries. The chatbot is designed to adapt its response style based on the type of question asked:

- For general queries, it provides concise, well-structured answers with bullet points where necessary.
- For definition-type questions, it offers detailed explanations, ensuring clarity and completeness.
- For each response, three reference links are included, allowing users to trace the retrieved information back to its original source. These links are clickable, directing users to the relevant web pages. This functionality is achieved by separately adding webpage URLs to the JSON files, ensuring that the chatbot not only provides accurate responses but also maintains transparency by linking to source documents. The following examples illustrate these capabilities in real-world interactions.

Additionally, screenshots of the original source web pages are included to demonstrate how the chatbot accurately retrieves and presents relevant data. These references were successfully integrated into the chatbot by embedding webpage URLs into the JSON files, ensuring both transparency and traceability in the information retrieval process.

In the Figure 11, the definition of "Core_Competences" is clearly displayed on the Stages web-UI, indicating that it serves as the primary source of information for the chatbot. This suggests that the chatbot relies on the structured data presented on the Stages platform to generate its responses.

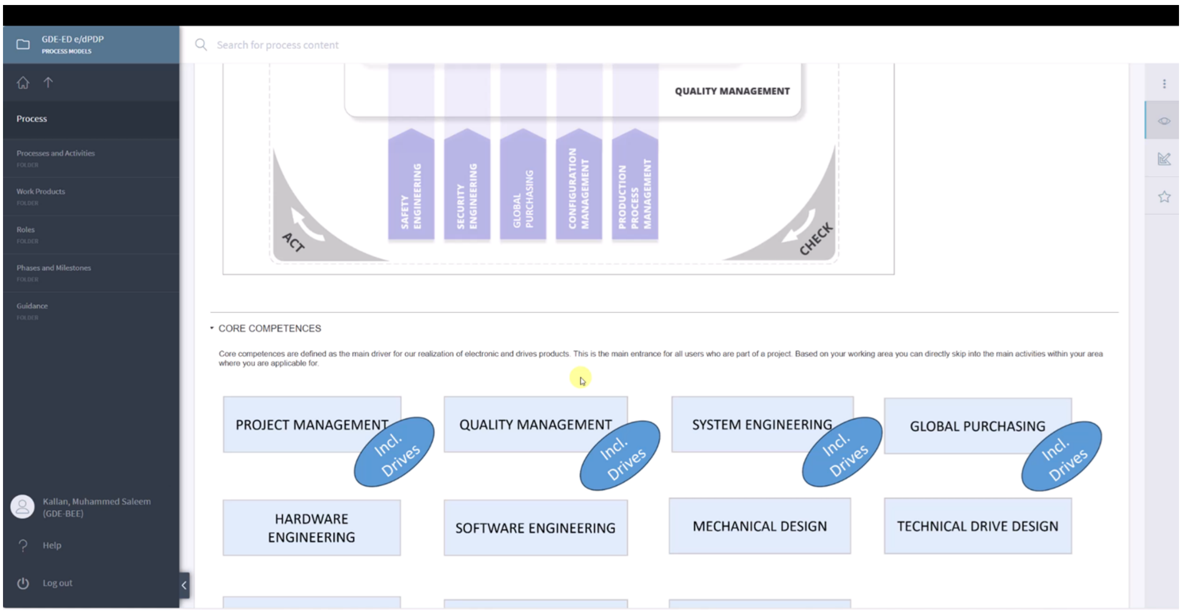

Figure 11: Definition of Core Competences in STAGES (Source of Response)

Figure 12 demonstrates how the chatbot processes and responds to a user query. The figure highlights the chatbot's ability to extract relevant information from the defined "Core_Competences" and deliver a coherent response based on that data. This reflects the chatbot's capability to retrieve and process information from the underlying data source effectively.Figure 10 Shows the source taken from the reference URLs in Figure 11.

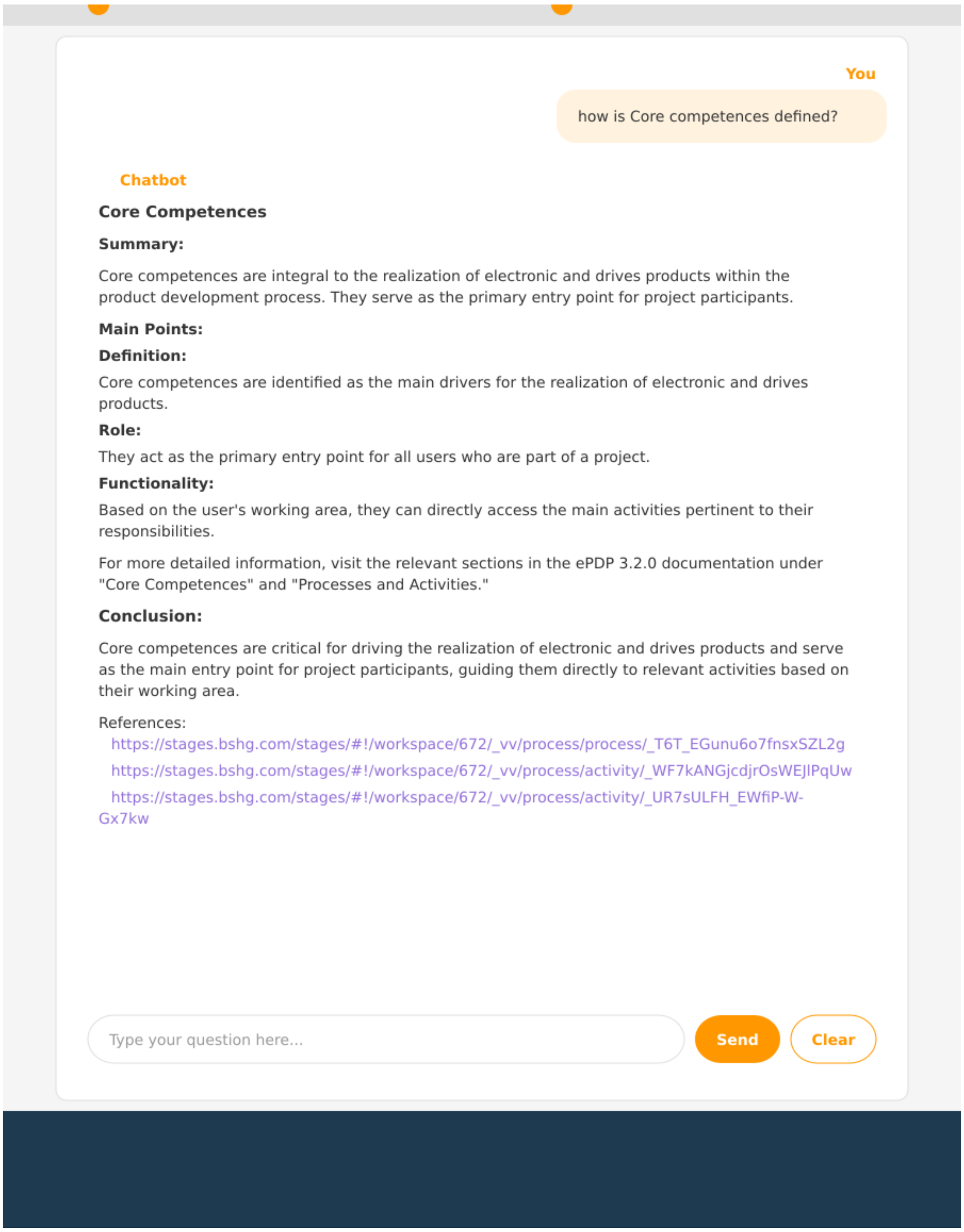

Figure 12: Chatbot Response for the query ”How is core competences defined?”

Figure 13 presents the chatbot’s response to the query ”What is RACI format.” The response is structured and explained clearly using bullet points, making it easy to understand.

Figure 14 shows the source of the information, where the data is organized in a tabular format. This demonstrates the chatbot’s ability to convert structured tabular data into a more user-friendly, summarized format when delivering responses.

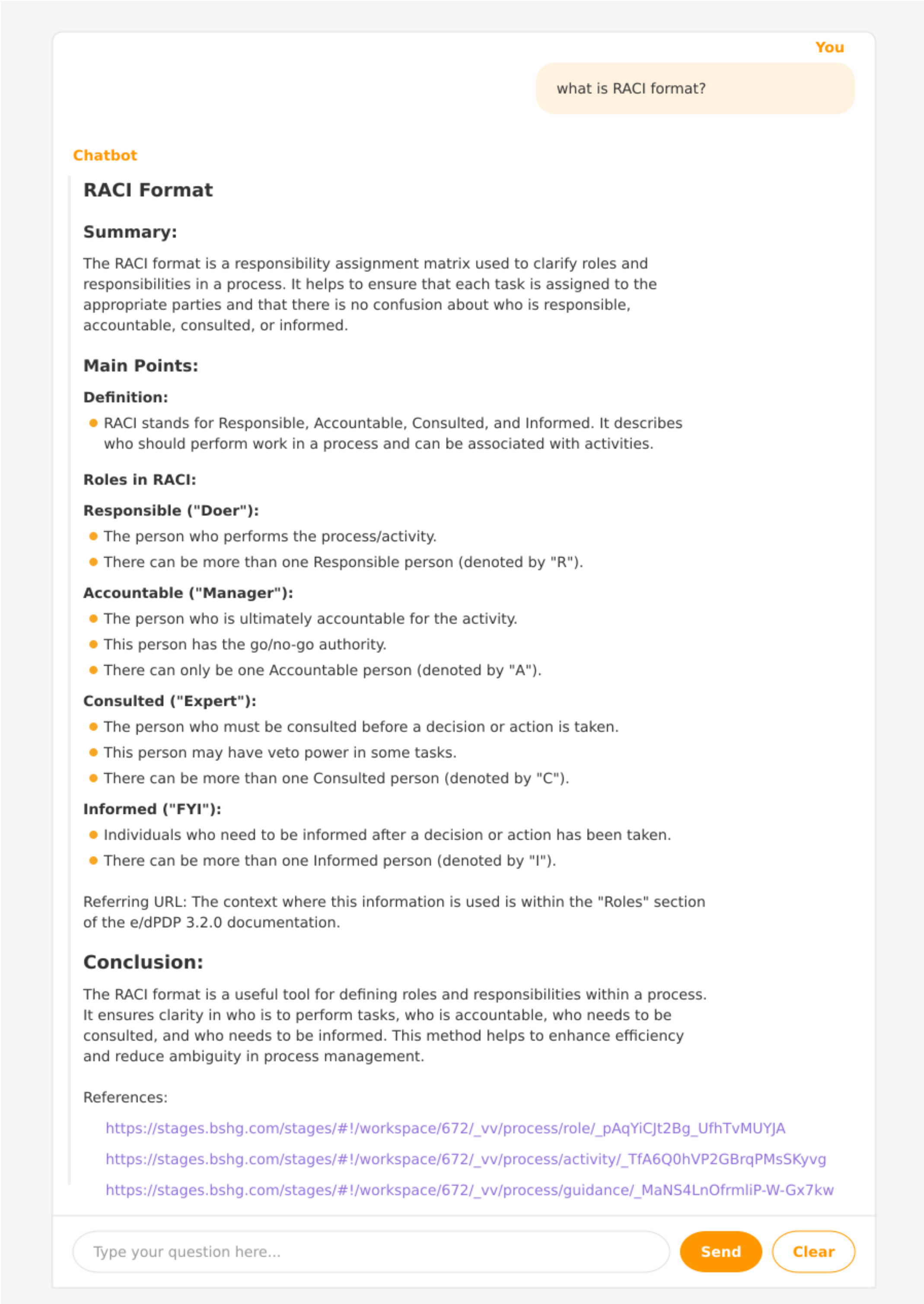

Figure 13: Chatbot Response for the query ”What is RACI format?

As the testing progressed, the chatbot was evaluated with more advanced and indirect queries to assess its ability to handle complex user inputs (Figure 15). Unlike basic definition-type questions, these queries required the chatbot to synthesize information from multiple sources while maintaining coherence and contextual accuracy in its responses.

During these interactions, the chatbot effectively retrieved data from multiple indexed web pages, leveraging Elasticsearch’s top-ranked results. In cases where a single source did not contain the complete answer, the chatbot combined information from different references, ensuring that the response remained meaningful and contextually relevant. Despite integrating data from multiple sources, the chatbot maintained logical

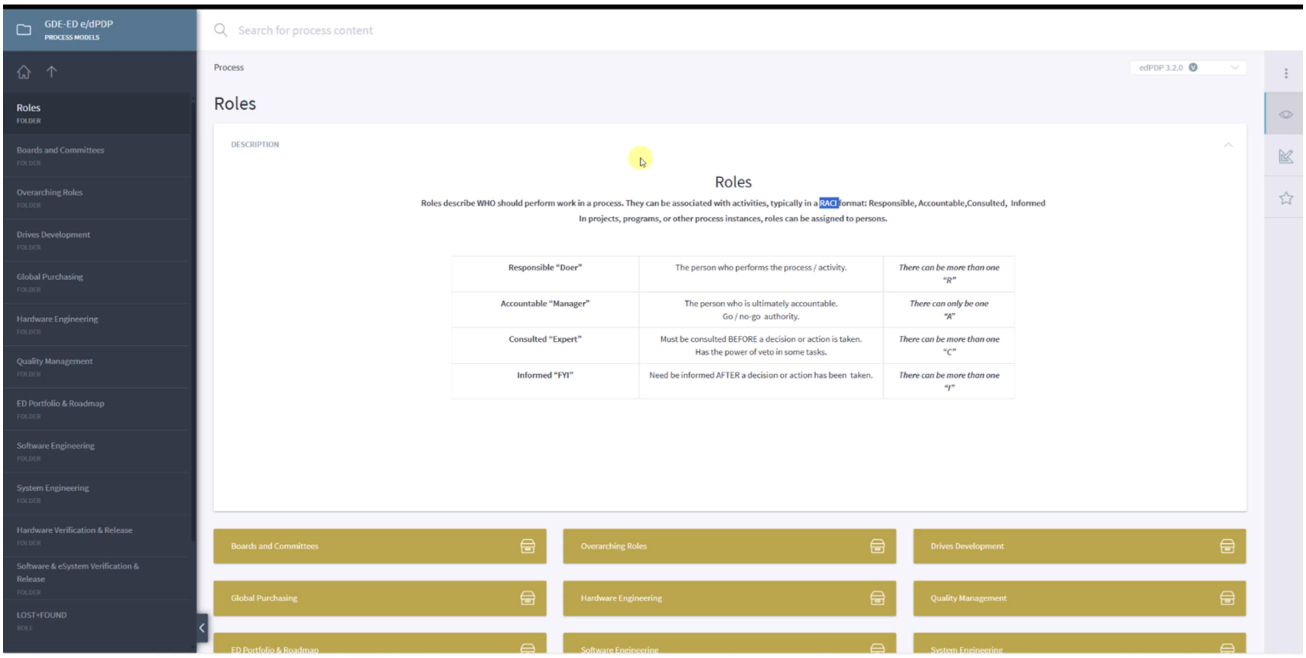

Figure 14: RACI Format Documented in STAGES (Source of Respones)

flow and coherence, allowing users to understand the synthesized information without confusion.

This capability is particularly valuable for scenarios where answers are not explicitly available in a single document but instead require aggregation from various knowledge bases. The chatbot's ability to dynamically merge and present information in a structured and comprehensible manner highlights its practical application in real-world decision-making processes.

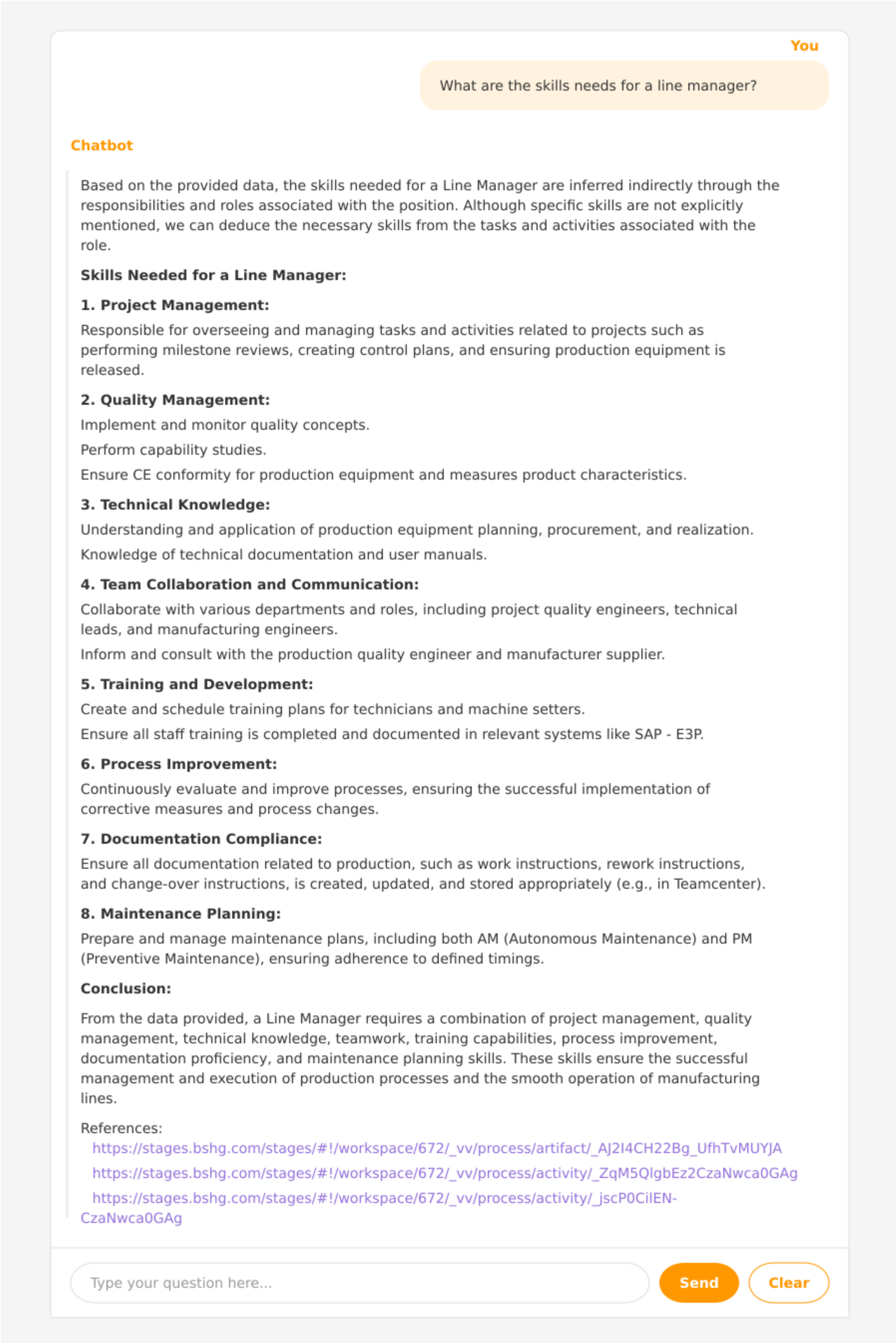

Figure 15: Chatbot Response for the query for the skills needed for a Line Manager

6.2 Test Setup and User Involvement

The chatbot was tested with 42 users, including process engineers and managers At BSH, over a designated period. During this phase, the chatbot was deployed in an internal environment where users could query it for relevant process-related information. The log data from the LLM API was collected, but individual user identification was not possible. The analysis primarily focused on the overall accuracy of responses rather than tracking specific user interactions.

6.2.1 Data Collection and Evaluation Methodology

To evaluate the performance of the chatbot models, a dataset consisting of 350 queries was collected and tested on four different models and Manually:

- BSH AIM GPT-3
- BSH AIM GPT-4
- Azure OpenAI GPT-3
- Azure OpenAI GPT-4

The responses generated by the models were compared against a reference dataset obtained from Elasticsearch (top 3 search results). The reference dataset served as the ground truth for evaluating the quality of the generated responses.

6.2.2 Evaluation Criteria

The evaluation of the chatbot responses focused on two key aspects:

- Coherence – Measures the logical flow and consistency of the generated responses.
- Relevance – Measures how accurately the response addresses the user's query.

Two widely used NLP metrics were employed to assess coherence and relevance:

1. BLEU Score – Evaluates the surface-level similarity of the chatbot's response to the reference response based on n-gram overlap.
2. BERTScore – Measures the semantic similarity between the chatbot's response and the reference response using contextual embeddings from the BERT language model.

6.2.3 BLEU Score Calculation

According to Papineni et al. [20], the BLEU score was computed using the formula:

$$B = \exp\left(\min\left(0, 1 - \frac{R}{C}\right)\right) \prod_{n=1}^{N} \left(p_n^{w_n}\right) \tag{4}$$

Where:

- R = Length of the reference response
- C = Length of the candidate (generated) response
- p_n = Precision of n-grams (number of matching n-grams / total n-grams)

- w_n = Weight assigned to each n-gram level (commonly 0.25 for BLEU-4)

A brevity penalty was applied if the candidate response was shorter than the reference response to discourage overly short answers that might artificially inflate precision. BLEU scores were calculated for n-gram levels from 1 to 4 using the nltk library in Python.

6.2.4 BERTScore Calculation

BERTScore was computed using cosine similarity between the contextual embeddings of the generated and reference responses[25]. The BERTScore formula is:

$$\text{BERTScore} = \frac{1}{|S|} \sum_{x \in S} \max_{y \in T} \text{cosine}(x, y) \tag{5}$$

Where:

- S = Set of token embeddings from the generated response
- T = Set of token embeddings from the reference response
- cosine(x, y) = Cosine similarity between two token embeddings

The bert-score Python package was used with the bert-base-uncased model. The F1-score from BERTScore was recorded as the final score for semantic similarity.

6.2.5 Human Evaluation

In addition to BLEU and BERTScore, a human evaluation was conducted to validate the automated results. Responses were rated by two independent reviewers for:

- Relevance – How well the response addresses the user's query (Scale: 1–5)
- Coherence – Logical flow and consistency of the response (Scale: 1–5)

6.3 Results

6.3.1 BLEU and BERTScore Performance

The performance of each model was evaluated using BLEU and BERTScore. Table 21 summarizes the results:

Model	BLEU Score	BERTScore (F1)	Human Rating (Relevance)	Human Rating (Coherence)	Accuracy Rate
BSH AIM GPT-3	0.61	0.84	3.5	3.6	70.5%
BSH AIM GPT-4	0.68	0.88	4.2	4.5	79.1%
Azure OpenAI GPT-3	0.58	0.82	3.4	3.5	68.0%
Azure OpenAI GPT-4	0.65	0.86	4.0	4.3	74.0%

Table 21: Performance comparison of different models

6.3.2 Analysis of Results

1. BSH AIM GPT-4 demonstrated the highest performance across all metrics:
   - Highest BLEU Score (0.68) → More accurate word-level matching
   - Highest BERTScore (0.88) → Strong semantic similarity with the reference response
   - Best human rating for coherence and relevance → Consistently structured and meaningful responses
2. Azure OpenAI GPT-4 showed competitive results but slightly lower than BSH AIM GPT-4:
   - Lower BLEU Score (0.65) indicates slightly less accurate surface-level matching.
   - Lower BERTScore (0.86) reflects reduced semantic alignment.
3. BSH AIM GPT-3 outperformed Azure OpenAI GPT-3:
   - Higher BLEU (0.61 vs 0.58) → More precise matching
   - Higher BERTScore (0.84 vs 0.82) → Better semantic similarity
4. GPT-4 vs GPT-3 Comparison:
   - GPT-4 models consistently outperformed GPT-3 models, reflecting improved language understanding and context handling.
   - The improvement in BERTScore for GPT-4 models indicates better semantic understanding rather than simple pattern matching.

6.3.3 Human Evaluation vs Automated Scores

The human evaluation results align closely with BLEU and BERTScore values:

- Higher BLEU and BERTScore values correspond to higher human ratings for relevance and coherence.
- GPT-4 models showed greater consistency and logical flow, as reflected in higher human ratings.

The results confirm that GPT-4 models (both BSH AIM and Azure OpenAI) outperform GPT-3 models in both accuracy and semantic similarity. The higher BLEU and BERTScore values for GPT-4 models demonstrate that they can generate both more accurate and more natural responses.

Furthermore, the BSH AIM models generally performed better than Azure OpenAI models. This suggests that the internal training and data compliance processes at BSH have led to better alignment with domain-specific content.

The correlation between BLEU, BERTScore, and human evaluation indicates that BLEU captures surface-level accuracy, while BERTScore reflects deeper semantic understanding. The combined use of both metrics, along with human feedback, provides a comprehensive evaluation of chatbot performance.

6.4 User Feedback and Evaluation

To assess the performance and user satisfaction of the chatbot, a survey was conducted among 42 BSH employees who regularly interact with the system. The survey aimed to evaluate key aspects such as usability, response accuracy, relevance of answers, and overall user experience. The detailed list of survey questions can be found in APPENDIX VII.

The feedback from the survey indicated that the chatbot was well-received by most users. A majority of respondents reported that the chatbot was easy to use and intuitive, with 85% of participants rating the user interface as user-friendly. The chatbot's ability to understand and accurately respond to queries was also highly rated, with 78% of users stating that they received the correct information on the first attempt.

Furthermore, 82% of respondents found the chatbot's response time to be fast and 77% mentioned that the chatbot was able to handle complex queries effectively. About 74% of users appreciated the chatbot's ability to suggest relevant follow-up questions and provide contextually accurate answers.

However, some respondents suggested minor improvements, such as enhancing the chatbot's ability to interpret ambiguous queries and refining the handling of multi-step questions. About 15% of users reported difficulties in getting precise answers for complex or highly technical queries. This feedback has been considered for future updates and improvements.

Overall, the survey results demonstrate that the chatbot has significantly improved information retrieval efficiency and user experience at BSH. The positive response reflects the successful integration of the chatbot into the existing knowledge management system and its potential for further development and scalability.

6.5 Limitations

1. **Data Ingestion and Retrieval Approach**

   The data ingestion for the chatbot was carried out using web scraping techniques, as direct migration or database copying required multiple security checks and firewall clearance, leading to significant time delays of approximately 2 to 4 months. Although web scraping provided an efficient alternative for data retrieval, it also introduced certain limitations:

   - **Potential Data Gaps:** Due to the nature of web scraping, some information may not have been captured, leading to occasional gaps in the dataset.
   - **Frequent Data Updates:** Unlike direct database connections, which allow for incremental updates, web scraping requires complete re-scraping each time new data is needed, as it is difficult to track which sections have changed.
   - **URL Referencing:** One of the advantages of this method was the ability to provide direct reference URLs in chatbot responses. Constructing these URLs from a database would have been more complex due to frequent content updates.

   The testing confirmed that while web scraping was a viable solution for initial deployment, it is not the most sustainable method for long-term updates. A direct database connection, once established, would allow for real-time updates, improving accuracy and reducing data redundancy.

2. **Limitations in Extracting Non-Textual Data**

Since these elements were embedded as non-textual content (e.g., images, SVGs, or graphical representations) rather than standard text, they were not scrapable using the Selenium-based web crawling approach. As a result, key visual data, such as process flows, decision trees, and illustrated explanations, were missing from the chatbot's knowledge base.

However, links related to such images were successfully extracted, allowing users to access the original web pages where these visual elements are present. While the chatbot itself cannot display or interpret the missing images directly, the inclusion of these reference links ensures that users can navigate to the source material when needed.

To further improve completeness, future enhancements could include optical character recognition (OCR) for extracting text from images or embedding direct links to images and diagrams within the chatbot's responses.

3. **Accuracy of Retrieved References**

   The chatbot was designed to retrieve and present three reference URLs at the end of each response. These URLs were intended to help users verify the source of information and establish the credibility of the chatbot's answers. The reference retrieval process followed these steps:

   (a) **Elasticsearch Indexing:** Each web page from the Stages platform was stored as a separate shard within Elasticsearch.

   (b) **Top-3 Shard Selection:** When a user posed a question, the system retrieved the three most relevant shards and sent them to the LLM for response generation.

   (c) **Reference URLs:** The URLs corresponding to these three shards were displayed at the end of the chatbot's response.

   Findings:

   - In most cases, the chatbot successfully retrieved URLs that contained relevant data, making it easier for users to cross-check the information.
   - However, there were instances where the reference links did not directly correspond to the response provided by the chatbot.
   - This issue occurred because, out of the three retrieved shards, only one or two might contain actual useful data, while the third shard did not contribute any meaningful content.
   - Identifying which specific shard(s) contributed to the final response remains challenging, as the LLM itself determines how to construct the answer using the given data.
   - A prompt was added to instruct the LLM to explicitly mention the shard names used, but the LLM does not yet provide this information consistently. Future refinements may be necessary to enhance reference accuracy.

4. **Frontend Prototype and User Interface Observations**

   The frontend design of the chatbot was developed solely as a proof-of-concept and is expected to be redesigned in the future for improved usability and performance. The testing process revealed several areas for enhancement in the user interface:

   Current Capabilities of the Frontend:

- A simple chat interface allowing users to enter questions and receive and clear responses.
- Display of retrieved reference URLs for verification.

Potential Future Enhancements:

(a) **LLM Model Switching:** Users could be provided with an option to switch between different LLM models.

(b) **LLM Version Selection:** A feature allowing users to choose between different model versions, such as GPT-3.5 or GPT-4or higher models based on their preference or requirement.

(c) **Custom Prompting:** Users could input custom prompts to modify the chatbot's behavior, allowing for more personalized interactions.

(d) **Session Management:**
   - The ability to start a new chat session separate from the previous one.
   - Improved conversation history tracking.

These enhancements would contribute to a more dynamic and user-friendly experience, enabling greater control over chatbot interactions.

6.6 Future Potential and Discussions

Addressing the limitations identified in this research, several improvements can enhance the chatbot's scalability, efficiency, and accuracy, making it a more robust knowledge management system.

1. **Optimizing Data Ingestion and Retrieval**
   - Secure Database Connectivity: Direct database integration will enable real-time updates, reducing the need for frequent web scraping.
   - Hybrid Data Ingestion: Combining web scraping with structured database access will improve data accuracy and update efficiency.
   - Automated Change Detection: Implementing differential scraping will minimize redundant processing and optimize resource usage.

2. **Extracting and Incorporating Non-Textual Data**
   - Optical Character Recognition (OCR): Enables text extraction from images to retain valuable information.
   - Embedding Image Links with Context: Displays relevant images alongside explanatory text for better comprehension.
   - Multimodal AI Integration: Future versions can process both text and visual data for a richer user experience.

3. **Enhancing Retrieval Accuracy in RAG**
   - Shard-Level Contribution Tracking: Improves reference accuracy by ranking retrieved sources based on relevance.
   - Metadata Tagging: Enhances contextual understanding for better document retrieval.
   - Fine-Tuned LLM for Attribution: Optimizes responses by explicitly linking sources to generated content.

4. **Strengthening Frontend and User Experience**
    - Interactive Model Selection: Allows users to choose between different AI models based on performance and cost.The code for the frontend and backend of the chatbot with multiple LLM selection capability has been added in APPENDIX V and APPENDIX VI for reference. The implementation supports switching between Azure OpenAI and BSH AIM as two different models, enabling flexible and dynamic model selection based on performance and user preferences.
    - Session Management: Retains conversation history for multi-step queries.
    - Role-Based UI Customization: Tailors responses based on user needs (e.g., engineers, project managers).

Scalability and Industry-Wide Application

- Cross-Domain Adaptability: Can be extended to legal, medical, and financial knowledge bases.
- Cloud-Native Scalability: Containerized deployment (e.g., Kubernetes) ensures seamless scaling.
- Enterprise Data Integration: Future versions can connect with ERP and CRM systems for real-time insights.
- Reducing AI Provider Dependence: Using open-source models (e.g., Llama, Falcon) ensures long-term sustainability.

6.7 Applicability of the Chatbot

The primary advantage of the chatbot developed in this thesis is its universality and adaptability across different industries and organizational structures. The methodology demonstrated in this study establishes a flexible, scalable, and secure approach for integrating large language models (LLMs) with Elasticsearch, providing significant value in various scenarios.

1. **Universal LLM Integration with Elasticsearch**

    One of the key findings of this thesis is that any LLM, regardless of the company it belongs to, can be successfully connected with Elasticsearch. While Elasticsearch provides built-in support for widely used LLM APIs such as ChatGPT, Azure OpenAI, and Gemini, this study proves that custom or proprietary LLMs can also be seamlessly integrated.

    This capability is crucial for organizations that:

    - Have concerns about data security when using third-party LLMs.
    - Prefer to use in-house or trusted AI models instead of commercial APIs.
    - Require a private and controlled RAG (Retrieval-Augmented Generation) setup to protect sensitive information.

    Without this flexibility, companies relying solely on external LLM providers would face potential security risks, as retrieved data from Elasticsearch is processed externally by third-party LLMs. However, the findings from this study eliminate these security concerns by enabling trusted, internal LLMs to handle RAG tasks, breaking the traditional limitations of Elasticsearch's built-in integrations.

2. **Broad Database Compatibility and Simplified Data Migration**

   Another major advantage of the proposed approach is its compatibility with a wide variety of databases, from legacy systems to modern cloud-based solutions. Elasticsearch natively supports multiple database connections, making data migration seamless depending on the security policies of individual companies.

   This flexibility makes the proposed method highly applicable to:

   - Corporations transitioning from older databases to modern data storage and retrieval systems.
   - Organizations with stringent data protection policies that restrict direct external connections but still require advanced search and retrieval capabilities.
   - Enterprises managing diverse datasets across multiple sources, ensuring smooth integration without disrupting existing workflows.

   Thus, the approach presented in this study is not limited to specific sectors or database architectures. It can be adopted across industries, providing companies with a future-proof solution for data accessibility and retrieval.

3. **Application for All Sized Businesses**

   The method of web scraping for data preparation, as employed in this study, provides an alternative data ingestion strategy for organizations that lack a structured database. Many small and medium-sized enterprises (SMEs) do not maintain well-organized databases but still require efficient knowledge retrieval systems.

   This study shows that:

   - SMEs without a structured database can still implement a functional chatbot-driven information retrieval system using web scraping.
   - Web scraping enables data collection from existing websites, even when direct database access is unavailable.
   - This method is cost-effective for smaller organizations that cannot invest in database migration or complex IT infrastructure.

   By using this approach, small companies can leverage AI-powered search functionalities without undergoing extensive database restructuring, making knowledge retrieval accessible to businesses of all sizes.

   Since Elasticsearch's connectivity spans across both traditional and modern database systems, this approach is highly adaptable to varied security and data governance policies across industries.

7 Conclusion

Summary of Findings and Contributions

This thesis set out to address the challenge of retrieving information from BSH's complex Stages project management platform by developing an AI-powered chatbot. The research successfully built a conversational interface that integrates an Elasticsearch backend with BSH's internal LLM (BSH AIM GPT-4) to interpret natural language queries and retrieve relevant project data. The resulting chatbot markedly improves information retrieval efficiency and accessibility for Stages users. It enables project engineers and managers to obtain answers instantly using everyday language, eliminating the need to manually navigate through multiple layers of the Stages interface. Key contributions of this work include demonstrating that natural language search can replace rigid query syntax, providing faster data retrieval and reduced manual effort, while ensuring standardized answers across the enterprise. Together, these improvements streamline project management processes by saving time, aiding new employee onboarding, and ensuring that everyone accesses information in a uniform manner.

Limitations

While the implementation proved effective, there are certain limitations to acknowledge in its current form. These limitations highlight areas where the system or methodology faced challenges and where future improvements are needed:

- **Natural Language Query Handling:** Although the chatbot understands many user queries, integrating truly open-ended natural language questions remains challenging. Some complex or ambiguous queries may not be interpreted correctly by the LLM, leading to less precise answers. This indicates a need for ongoing refinement of the language model and query parsing techniques to cover a broader range of user expressions and project-specific terminology.
- **Scalability and Performance:** The current solution has not yet been tested at a large enterprise scale with many simultaneous users or extremely large data volumes. As usage grows, response times could be affected if the underlying infrastructure (Elasticsearch, LLM API, etc.) is not scaled appropriately. Moreover, the data ingestion method via web scraping, while suitable for a proof-of-concept, is not ideal for real-time updates – new or changed project data require re-crawling the entire site, which could become inefficient at scale. Thus, there are potential scalability issues in both the application's throughput and the data update process that need to be addressed for long-term, enterprise-wide deployment.
- **Reliance on Structured Project Data:** The chatbot's knowledge is confined to the structured data extracted from the Stages platform. Any information not captured in the structured format (or not present in the Stages database) is beyond the chatbot's reach. For instance, graphical elements like process flowcharts or embedded images were not fully ingested, meaning the chatbot cannot directly answer questions about those visuals. This reliance on structured textual data means the system may miss context that is available only in non-textual or unstructured forms. It also implies that the quality of the chatbot's responses is highly dependent on the completeness and accuracy of the underlying project data; inconsistencies or gaps in the source data will be reflected in the answers.

Future Research Directions

The encouraging results of this project open several avenues for future research and enhancements. Building on the current framework, researchers and practitioners can explore improvements that make the chatbot more powerful, versatile, and broadly applicable beyond the immediate context of project management:

- **Enhanced AI Capabilities:** Future work can focus on upgrading the chatbot's language understanding and generation abilities. This might include integrating more advanced or domain-specific LLMs, fine-tuning the model on project management terminology, or allowing dynamic model selection (e.g., choosing between different AI models or versions for better performance). Additionally, enabling the chatbot to maintain richer context over multi-turn conversations (improved session management) would make interactions more natural, allowing users to ask follow-up questions without starting from scratch. Another promising direction is to incorporate multimodal AI capabilities – for example, processing visual data or diagrams related to projects – so the chatbot can interpret charts or images and include them in its responses. These AI enhancements would collectively make the chatbot an even more intelligent assistant, capable of handling complex inquiries with greater accuracy and depth.
- **Broader Data Integration:** To overcome the limitations of data sourcing and scalability, future implementations should integrate a wider range of data sources. A logical next step is connecting the chatbot directly to enterprise databases and tools (e.g. a direct SQL database link or APIs to the Stages backend) to allow real-time data retrieval and eliminate the need for repetitive web scraping. This would ensure that answers are always based on the latest information and drastically reduce maintenance overhead. In addition, integrating other enterprise systems such as ERP or CRM platforms could provide the chatbot with a more holistic view of project-related data (for instance, linking project tasks with resource or financial information). Embracing a hybrid data ingestion approach – combining structured database queries with selective web crawling or even document repositories – would make the system more robust and scalable. Ensuring compliance with security and privacy will be key here, but the architecture demonstrated in this thesis (which successfully used an internal LLM with Elasticsearch) indicates that secure, flexible integration is feasible.
- **Applications Beyond Project Management:** Perhaps one of the most exciting future directions is adapting and applying this AI chatbot approach to domains beyond the Stages platform. The core solution of an LLM-driven assistant over an Elasticsearch index is domain-agnostic and can serve as a blueprint for enterprise knowledge management in various fields. For example, other departments at BSH or similar enterprises could deploy chatbots to navigate HR policies, customer support knowledge bases, technical manuals, or financial data, using natural language queries just as this project has done for project management. The methodology is highly adaptable across industries, from legal and medical knowledge bases to technical support databases. Small and medium-sized enterprises could also benefit, as demonstrated by this study's web-scraping approach – even without sophisticated databases, they can create AI assistants on top of existing web content to improve information access. In summary, the principles and architecture developed here form a future-proof foundation for conversational AI applications in many other contexts, promising to streamline information retrieval well beyond the project management sphere.

Closing Remarks

In conclusion, the development of the Stages Platform chatbot has fulfilled the research objectives with an accuracy rate of 79.1% and addressed the initial problem statement by providing an intuitive solution to navigate complex project data. The chatbot not only proved the feasibility of integrating conversational

AI into enterprise project management systems but also demonstrated tangible benefits in efficiency, consistency, and user empowerment. By bridging the gap between intricate data structures and user-friendly access, this research underscores the significance of AI-driven chatbots as tools for enhancing organizational knowledge management. The findings and framework laid out in this thesis serve as a strong foundation for continued innovation, heralding a future where enterprise information no matter how complex is only a natural language query away.

References

[1] Eleni Adamopoulou and Lefteris Moussiades. Chatbots: History, technology, and applications. *Machine Learning with Applications*, 2:100006, 2020.

[2] Hannah Bast, Björn Buchhold, and Elmar Haussmann. Semantic search on text and knowledge bases. *Foundations and Trends® in Information Retrieval*, 10(2-3):119–271, 2016.

[3] Tom Brown, Benjamin Mann, Nick Ryder, Melanie Subbiah, Jared D Kaplan, Prafulla Dhariwal, Arvind Neelakantan, Pranav Shyam, Girish Sastry, Amanda Askell, Sandhini Agarwal, Ariel Herbert-Voss, Gretchen Krueger, Tom Henighan, Rewon Child, Aditya Ramesh, Daniel Ziegler, Jeffrey Wu, Clemens Winter, Chris Hesse, Mark Chen, Eric Sigler, Mateusz Litwin, Scott Gray, Benjamin Chess, Jack Clark, Christopher Berner, Sam McCandlish, Alec Radford, Ilya Sutskever, and Dario Amodei. Language models are few-shot learners. In H. Larochelle, M. Ranzato, R. Hadsell, M.F. Balcan, and H. Lin, editors, *Advances in Neural Information Processing Systems*, volume 33, pages 1877–1901. Curran Associates, Inc., 2020.

[4] Tom B. Brown, Benjamin Mann, Nick Ryder, Melanie Subbiah, Jared Kaplan, Prafulla Dhariwal, Arvind Neelakantan, Pranav Shyam, Girish Sastry, Amanda Askell, Sandhini Agarwal, Ariel Herbert-Voss, Gretchen Krueger, Tom Henighan, Rewon Child, Aditya Ramesh, Daniel M. Ziegler, Jeffrey Wu, Clemens Winter, Christopher Hesse, Mark Chen, Eric Sigler, Mateusz Litwin, Scott Gray, Benjamin Chess, Jack Clark, Christopher Berner, Sam McCandlish, Alec Radford, Ilya Sutskever, and Dario Amodei. Language models are few-shot learners, 2020.

[5] Gobinda G Chowdhury. *Introduction to modern information retrieval*. Facet publishing, 2010.

[6] Yunfan Gao, Yun Xiong, Xinyu Gao, Kangxiang Jia, Jinliu Pan, Yuxi Bi, Yi Dai, Jiawei Sun, Meng Wang, and Haofen Wang. Retrieval-augmented generation for large language models: A survey, 2024.

[7] Clinton Gormley and Zachary Tong. *Elasticsearch: the definitive guide: a distributed real-time search and analytics engine*. " O'Reilly Media, Inc.", 2015.

[8] Shailja Gupta, Rajesh Ranjan, and Surya Narayan Singh. A comprehensive survey of retrieval-augmented generation (rag): Evolution, current landscape and future directions, 2024.

[9] Jan L Harrington. *Relational database design clearly explained*. Morgan Kaufmann, 2002.

[10] Matthew Hoy. Alexa, siri, cortana, and more: An introduction to voice assistants. *Medical Reference Services Quarterly*, 37:81–88, 01 2018.

[11] Daniel Jurafsky and James H. Martin. *Speech and Language Processing: An Introduction to Natural Language Processing, Computational Linguistics, and Speech Recognition with Language Models*. 3rd edition, 2025. Online manuscript released January 12, 2025.

[12] Darshita Kalyani and Devarshi Mehta. Paper on searching and indexing using elasticsearch. *Int. J. Eng. Comput. Sci*, 6(6):21824–21829, 2017.

[13] Madhusudhan Konda. *Elasticsearch in action*. Simon and Schuster, 2024.

[14] Patrick Lewis, Ethan Perez, Aleksandra Piktus, Fabio Petroni, Vladimir Karpukhin, Naman Goyal, Heinrich Küttler, Mike Lewis, Wen-tau Yih, Tim Rocktäschel, Sebastian Riedel, and Douwe Kiela. Retrieval-augmented generation for knowledge-intensive nlp tasks. In H. Larochelle, M. Ranzato, R. Hadsell, M.F. Balcan, and H. Lin, editors, *Advances in Neural Information Processing Systems*, volume 33, pages 9459–9474. Curran Associates, Inc., 2020.

[15] Patrick Lewis, Ethan Perez, Aleksandra Piktus, Fabio Petroni, Vladimir Karpukhin, Naman Goyal, Heinrich Küttler, Mike Lewis, Wen tau Yih, Tim Rocktäschel, Sebastian Riedel, and Douwe Kiela. Retrieval-augmented generation for knowledge-intensive nlp tasks, 2021.

[16] Mengyi Liu and Jianqiu Xu. Nli4db: A systematic review of natural language interfaces for databases. *arXiv preprint arXiv:2503.02435*, 2025.

[17] Ali Mohammadjafari, Anthony S Maida, and Raju Gottumukkala. From natural language to sql: Review of llm-based text-to-sql systems. *arXiv preprint arXiv:2410.01066*, 2024.

[18] Mohammad Nuruzzaman and Omar Khadeer Hussain. A survey on chatbot implementation in customer service industry through deep neural networks. In *2018 IEEE 15th International Conference on e-Business Engineering (ICEBE)*, pages 54–61, 2018.

[19] Joshua Ofoeda, Richard Boateng, and John Effah. Application programming interface (api) research: A review of the past to inform the future. *International Journal of Enterprise Information Systems (IJEIS)*, 15(3):76–95, 2019.

[20] Kishore Papineni, Salim Roukos, Todd Ward, and Wei-Jing Zhu. Bleu: a method for automatic evaluation of machine translation. In *Proceedings of the 40th annual meeting of the Association for Computational Linguistics*, pages 311–318, 2002.

[21] Stephen Robertson, Hugo Zaragoza, et al. The probabilistic relevance framework: Bm25 and beyond. *Foundations and Trends® in Information Retrieval*, 3(4):333–389, 2009.

[22] Xiaoyu Shen, Svitlana Vakulenko, Marco del Tredici, Gianni Barlacchi, Bill Byrne, and Adrià de Gispert. Low-resource dense retrieval for open-domain question answering: A comprehensive survey, 2022.

[23] Saku-Matti Syrjä. Retrieval-augmented generation utilizing sql database case: web sport statistics application. 2024.

[24] He Wang, Chengying Chi, Xiumei Zhang, and Yunyun Zhan. Inverted index construction algorithms for large-scale data. *IAENG International Journal of Computer Science*, 49(4), 2022.

[25] Tianyi Zhang, Varsha Kishore, Felix Wu, Kilian Q Weinberger, and Yoav Artzi. Bertscore: Evaluating text generation with bert. *arXiv preprint arXiv:1904.09675*, 2019.

[26] Xiaohu Zhu, Qian Li, Lizhen Cui, and Yongkang Liu. Large language model enhanced text-to-sql generation: A survey, 2024.

Appendices

I Code - Web Scraping

```
import os
import time
from selenium import webdriver
from selenium.webdriver.chrome.service import Service
from selenium.webdriver.common.by import By
import hashlib

# Function to check if a URL should be processed
def should_process_url(url, prefix):
    """
    Checks if a URL should be processed based on the prefix.
    :param url: The URL to check.
    :param prefix: The required prefix for valid URLs.
    :return: True if the URL starts with the prefix, False otherwise.
    """
    return isinstance(url, str) and url.startswith(prefix)

# Function to generate a safe filename
def get_safe_filename(url):
    """
    Generates a sanitized, safe filename for a given URL.
    :param url: The URL to convert into a filename.
    :return: A sanitized filename string.
    """
    name = url.split('/')[-1]  # Extract the last part of the URL
    if not name:  # Generate a hash-based name if no name exists
        name = hashlib.md5(url.encode()).hexdigest()
    return ''.join(c for c in name if c.isalnum() or c in '._-')[:50]  # Sanitize the
       filename

# Function to save a webpage locally
def save_page_locally(url, output_dir, driver):
    """
    Saves the HTML content of a webpage locally.
    :param url: The URL to save.
    :param output_dir: The directory where the file will be saved.
    :param driver: The Selenium WebDriver instance.
    :return: True if the page was saved successfully, False otherwise.
    """
    try:
        print(f"Saving page: {url}")
        driver.get(url)  # Navigate to the URL
        time.sleep(5)  # Wait for the page to load

        # Save the HTML content
        html_content = driver.page_source

        # Add the canonical URL in a <link> tag inside the <head> section
        canonical_tag = f'<link rel="canonical" href="{url}">\n'
        if "<head>" in html_content:
```

```
            html_content_with_canonical = html_content.replace("<head>", f"<head>\n{
                canonical_tag}", 1)
        else:
            html_content_with_canonical = f"<head>\n{canonical_tag}</head>\n" +
                html_content

        # Generate a safe filename
        safe_filename = url.replace(":", "_").replace("/", "_").replace("?", "_").
            replace("#", "_") + ".html"
        file_path = os.path.join(output_dir, safe_filename)

        # Write the HTML content with the canonical URL to the file
        with open(file_path, "w", encoding="utf-8") as f:
            f.write(html_content_with_canonical)

        print(f"Saved HTML with canonical URL: {file_path}")
        return True

    except Exception as e:
        print(f"Error saving page {url}: {str(e)}")
        return False

# Function to get all links from the current page
def get_all_links(driver, pdf_links):
    """
    Extracts all links and PDF links from the current page.
    :param driver: The Selenium WebDriver instance.
    :param pdf_links: A set to store discovered PDF links.
    :return: A set of non-PDF links from the page.
    """
    links = set()
    elements = driver.find_elements(By.XPATH, "//a | //*[@onclick] | //*[@role='button
        ']")

    for element in elements:
        try:
            href = element.get_attribute("href")
            if href:
                if href.lower().endswith('.pdf'):
                    pdf_links.add(href)  # Store PDF links
                else:
                    links.add(href)
                continue

            # For clickable elements without href, simulate a click
            element.click()
            time.sleep(1)

            current_url = driver.current_url
            if not current_url.lower().endswith('.pdf'):
                links.add(current_url)
            driver.back()  # Navigate back
            time.sleep(1)

        except:
```

```
            continue  # Ignore elements that cannot be clicked

    return links

# Recursive function to scrape process pages
def scrape_process_pages(url, driver, output_dir, process_url_prefix, visited_links,
    pdf_links, depth=0, max_depth=10):
    """
    Recursively processes pages, saves their content, and follows their links.
    :param url: The URL to process.
    :param driver: The Selenium WebDriver instance.
    :param output_dir: Directory to save pages.
    :param process_url_prefix: The URL prefix to filter links.
    :param visited_links: A set of already visited links.
    :param pdf_links: A set to store discovered PDF links.
    :param depth: Current recursion depth.
    :param max_depth: Maximum allowed recursion depth.
    """
    if not should_process_url(url, process_url_prefix) or url in visited_links or
        depth > max_depth:
        return  # Stop processing if the URL is invalid or already visited

    print(f"\nProcessing page {len(visited_links) + 1}: {url}")
    visited_links.add(url)

    try:
        save_page_locally(url, output_dir, driver)  # Save the current page
        links = get_all_links(driver, pdf_links)  # Extract links from the page
        print(f"Found {len(links)} links on {url}")

        for link in links:  # Recursively process links
            if link not in visited_links:
                scrape_process_pages(link, driver, output_dir, process_url_prefix,
                    visited_links, pdf_links, depth + 1, max_depth)

    except Exception as e:
        print(f"Error processing {url}: {str(e)}")

# Main execution
if __name__ == "__main__":
    # Configuration parameters - REPLACE THESE WITH YOUR OWN VALUES
    main_url = "https://example.com/main-page"  # Starting URL for scraping
    process_url_prefix = "https://example.com/"  # URL prefix to limit scraping scope
    output_dir = "output_html_folder"  # Directory to save scraped pages
    chrome_driver_path = "path/to/chromedriver"  # Path to chromedriver executable

    pdf_links_file = os.path.join(output_dir, "pdf_links.txt")
    visited_links = set()
    pdf_links = set()

    os.makedirs(output_dir, exist_ok=True)  # Ensure output directory exists

    # Set up Selenium WebDriver
    options = webdriver.ChromeOptions()
    options.add_argument("--start-maximized")
```

```
    options.add_argument("--disable-popup-blocking")
    options.add_argument("--disable-notifications")
    service = Service(chrome_driver_path)
    driver = webdriver.Chrome(service=service, options=options)

    try:
        # Start scraping
        scrape_process_pages(main_url, driver, output_dir, process_url_prefix,
            visited_links, pdf_links)

        # Save processed URLs and PDF links
        with open(os.path.join(output_dir, "processed_urls.txt"), "w", encoding="utf-8
            ") as f:
            for url in visited_links:
                f.write(f"{url}\n")

        with open(pdf_links_file, "w", encoding="utf-8") as f:
            for url in pdf_links:
                f.write(f"{url}\n")

        print(f"\nTotal pages processed: {len(visited_links)}")
        print(f"PDF links found: {len(pdf_links)}")
        print(f"Pages saved to: {output_dir}")

    finally:
        driver.quit()  # Ensure WebDriver is closed
```

II Code - Conversion of html to JSON

```
import os
import json
from bs4 import BeautifulSoup
from datetime import datetime

def process_html_file(file_path):
    """
    Extracts data from a single HTML file and organizes it into a structured document.
    :param file_path: Path to the HTML file.
    :return: A dictionary containing extracted data or None if an error occurs.
    """
    try:
        with open(file_path, 'r', encoding='utf-8') as file:
            content = file.read()

        # Parse HTML content
        soup = BeautifulSoup(content, 'lxml')

        # Extract URL information from canonical tag
        canonical_tag = soup.find('link', rel='canonical')
        canonical_url = canonical_tag['href'] if canonical_tag and canonical_tag.get('
            href') else "N/A"

        # Extract metadata and content
        title = soup.title.string if soup.title else "N/A"
```

```python
        meta_description = soup.find('meta', attrs={'name': 'description'})
        meta_keywords = soup.find('meta', attrs={'name': 'keywords'})

        # Extract links
        links = [link.get('href') for link in soup.find_all('a') if link.get('href')]

        # Extract headings
        headings = [tag.text.strip() for tag in soup.find_all(['h1', 'h2', 'h3', 'h4',
             'h5', 'h6'])]

        # Extract all text content
        body_content = soup.get_text(strip=True, separator="\n")

        # Prepare document structure
        document = {
            "id": os.path.splitext(os.path.basename(file_path))[0],  # Unique ID based
                 on filename
            "url": canonical_url,
            "canonical_url": canonical_url,
            "url_host": canonical_url.split('/')[2] if "://" in canonical_url else "N/
                A",
            "url_path": "/" + "/".join(canonical_url.split('/')[3:]) if "://" in
                canonical_url else "N/A",
            "url_path_dir1": "/" + canonical_url.split('/')[3] if "://" in
                canonical_url and len(canonical_url.split('/')) > 3 else "N/A",
            "url_path_dir2": "/" + canonical_url.split('/')[4] if "://" in
                canonical_url and len(canonical_url.split('/')) > 4 else "N/A",
            "url_path_dir3": "/" + canonical_url.split('/')[5] if "://" in
                canonical_url and len(canonical_url.split('/')) > 5 else "N/A",
            "url_port": "80",  # Adjust if port information is available
            "url_scheme": canonical_url.split(':')[0] if "://" in canonical_url else "
                N/A",
            "title": title,
            "meta_description": meta_description['content'] if meta_description else "
                N/A",
            "meta_keywords": meta_keywords['content'] if meta_keywords else "N/A",
            "body_content": body_content,
            "links": links,
            "headings": headings,
            "last_crawled_at": datetime.utcnow().isoformat()
        }

        return document

    except Exception as e:
        print(f"Error processing file {file_path}: {e}")
        return None

if __name__ == "__main__":
    """
    Main entry point of the script. Processes HTML files, extracts content,
    and saves the data into a JSON file.
    """
    # Input and output paths
    input_folder = "INPUT_HTML_FOLDER_PATH"  # Replace with your HTML files directory
```

```python
output_file = "OUTPUT_JSON_FILE_PATH"    # Replace with your output file path
index_name = "YOUR_INDEX_NAME"           # Replace with your index name

# List to store all documents
documents = []

# Process all HTML files in the input folder
for filename in os.listdir(input_folder):
    if filename.endswith(".html"):
        file_path = os.path.join(input_folder, filename)
        print(f"Processing file: {filename}")

        # Process the HTML file
        document = process_html_file(file_path)

        if document:
            documents.append(document)

# Prepare the bulk JSON format for Elasticsearch
bulk_data = []
for doc in documents:
    bulk_data.append({"index": {"_index": index_name, "_type": "_doc"}})  # Index
       metadata
    bulk_data.append(doc)

# Save the data to a single JSON file
with open(output_file, 'w', encoding='utf-8') as json_file:
    for entry in bulk_data:
        json.dump(entry, json_file, ensure_ascii=False)
        json_file.write('\n')

print(f"\nAll data has been combined and saved to {output_file}")
```

Listing 1: HTML Data Extraction and Processing Script

III Code - Frontend Design

```html
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8"/>
    <meta name="viewport" content="width=device-width, initial-scale=1.0"/>
    <title>Stages Chatbot</title>
    <link href="https://cdnjs.cloudflare.com/ajax/libs/font-awesome/6.0.0/css/all.min.
       css" rel="stylesheet"/>
    <style>
        @import url('https://fonts.googleapis.com/css2?family=Inter:wght@400;500;600&
           display=swap');

        * {
            margin: 0;
            padding: 0;
            box-sizing: border-box;
        }
```

```css
body {
    font-family: 'Inter', sans-serif;
    background: #fff;
    color: #333;
    line-height: 1.6;
    height: 100vh;
    display: flex;
    flex-direction: column;
    overflow: hidden;
}

.header {
    text-align: center;
    padding: 2rem 1rem;
    background: rgba(255, 255, 255, 0.95);
    backdrop-filter: blur(10px);
    border-bottom: 1px solid rgba(255, 165, 0, 0.2);
    position: fixed;
    width: 100%;
    top: 0;
    z-index: 1000;
    box-shadow: 0 2px 10px rgba(255, 165, 0, 0.1);
    transition: all 0.3s ease;
}

.header.minimized {
    padding: 0.5rem;
    transform: translateY(-100%);
}

h1 {
    font-size: 2.5rem;
    color: #ff8c00;
    margin-bottom: 0.5rem;
    font-weight: 600;
    transition: all 0.3s ease;
}

.subtitle {
    color: #666;
    font-size: 1rem;
    transition: all 0.3s ease;
}

.chat-container {
    flex: 1;
    max-width: 1000px;
    margin: 7rem auto 0;
    padding: 1rem;
    width: 95%;
    display: flex;
    flex-direction: column;
    gap: 1rem;
    overflow-y: auto;
```

```css
    height: calc(100vh - 180px);
}

.chat-container.header-minimized {
    margin-top: 3rem;
}

.message {
    padding: 1.5rem;
    border-radius: 1rem;
    max-width: 85%;
    animation: fadeIn 0.5s ease-in-out;
    box-shadow: 0 4px 6px rgba(0, 0, 0, 0.1);
}

@keyframes fadeIn {
    from { opacity: 0; transform: translateY(20px); }
    to { opacity: 1; transform: translateY(0); }
}

.user {
    background: rgba(255, 165, 0, 0.1);
    border: 1px solid rgba(255, 165, 0, 0.2);
    align-self: flex-end;
    margin-left: 15%;
}

.bot {
    background: #fff;
    border: 1px solid rgba(255, 165, 0, 0.2);
    align-self: flex-start;
    margin-right: 15%;
}

.message-header {
    display: flex;
    align-items: center;
    margin-bottom: 1rem;
    font-weight: 500;
    color: #333;
}

.message-header i {
    margin-right: 0.5rem;
    color: #ff8c00;
}

.message-content {
    line-height: 1.6;
    color: #333;
}

.content-line {
    margin: 8px 0;
}
```

```css
.list-item {
    display: flex;
    align-items: flex-start;
    margin: 8px 0;
    padding-left: 20px;
}

.bullet {
    color: #ff8c00;
    margin-right: 10px;
    margin-left: -20px;
}

/* Bold/strong text style */
.message-content strong {
    font-weight: 600;
    color: #333;
}

/* Italic/em text style */
.message-content em {
    font-style: italic;
    color: #333;
}

.input-container {
    background: rgba(255, 255, 255, 0.95);
    backdrop-filter: blur(10px);
    border-top: 1px solid rgba(255, 165, 0, 0.2);
    padding: 1.5rem;
    position: sticky;
    bottom: 0;
    display: flex;
    gap: 1rem;
    max-width: 1000px;
    margin: 0 auto;
    width: 95%;
    box-shadow: 0 -2px 10px rgba(255, 165, 0, 0.1);
}

#queryInput {
    flex: 1;
    padding: 1rem 1.5rem;
    border: 1px solid rgba(255, 165, 0, 0.3);
    border-radius: 2rem;
    font-size: 1rem;
    transition: all 0.3s ease;
}

#queryInput:focus {
    outline: none;
    border-color: #ff8c00;
    box-shadow: 0 0 0 3px rgba(255, 140, 0, 0.1);
}
```

```
        button {
            padding: 1rem 1.5rem;
            border: none;
            border-radius: 2rem;
            font-size: 1rem;
            cursor: pointer;
            transition: all 0.3s ease;
            display: flex;
            align-items: center;
            gap: 0.5rem;
        }

        .submit-btn {
            background: linear-gradient(135deg, #ff8c00 0%, #ff6347 100%);
            color: white;
        }

        .clear-btn {
            background: rgba(255, 255, 255, 0.9);
            color: #ff8c00;
            border: 1px solid rgba(255, 165, 0, 0.3);
        }

        button:hover {
            transform: translateY(-2px);
            box-shadow: 0 4px 12px rgba(255, 140, 0, 0.2);
        }

        @media (max-width: 768px) {
            .message {
                max-width: 90%;
            }

            .input-container {
                flex-direction: column;
                gap: 0.5rem;
            }

            button {
                width: 100%;
                justify-content: center;
            }
        }
    </style>
</head>
<body>
    <div class="header">
        <h1>Stages Chatbot</h1>
        <p class="subtitle">Ask me !</p>
    </div>

    <div class="chat-container"></div>

    <div class="input-container">
```

```
    <input type="text" id="queryInput" placeholder="Type your question here...">
    <button class="submit-btn" onclick="sendQuery()">
        <i class="fas fa-paper-plane"></i>
        Send
    </button>
    <button class="clear-btn" onclick="clearChat()">
        <i class="fas fa-trash"></i>
        Clear
    </button>
</div>

<script>
    const header = document.querySelector('.header');
    const chatContainer = document.querySelector('.chat-container');

    function minimizeHeader() {
        header.classList.add('minimized');
        chatContainer.classList.add('header-minimized');
    }

    function formatMessage(text, references = []) {
        function formatContent(content) {
            // 1. Trim overall text to remove trailing/leading whitespace
            content = content.trim();

            // 2. Convert **text** to <strong>text</strong>
            content = content.replace(/\*\*(.*?)\*\*/g, '<strong>$1</strong>');

            // 3. Convert *text* to <em>text</em>
            content = content.replace(/\*(.*?)\*/g, '<em>$1</em>');

            // 4. Split by new lines and handle bullet points or normal lines
            let lines = content.split('\n');
            let formattedLines = lines.map(line => {
                line = line.trim();
                if (!line) return '';

                // If it starts with "- ", create a bullet list item
                if (line.startsWith('- ')) {
                    return `<div class="list-item"><span class="bullet">   </span>
                        ${line.substring(2)}</div>`;
                }

                // Otherwise, wrap in a content-line div
                return `<div class="content-line">${line}</div>`;
            });

            return formattedLines.join('');
        }

        let formattedContent = formatContent(text);

        let html = `
            <div class="message-header">
                <i class="fas fa-robot"></i> Chatbot
```

```
        </div>
        <div class="message-content">
            ${formattedContent}
        </div>`;

    // Only add references if they exist and are not empty
    if (references && references.length > 0 && references.some(ref => ref.trim
        () !== '')) {
        html += `
            <div class="references-container">
                <div class="references-title">References:</div>
                ${references
                    .filter(ref => ref && ref.trim() !== '')
                    .map(ref => `
                        <div class="reference-item">
                            <a href="${ref}" target="_blank">
                                <i class="fas fa-link"></i>
                                ${ref}
                            </a>
                        </div>
                    `)
                    .join('')}
            </div>`;
    }

    return html;
}

async function sendQuery() {
    const userQuery = document.getElementById("queryInput").value;
    if (!userQuery.trim()) return;

    // Minimize header on first message
    minimizeHeader();

    // Show user's query in the chat
    const userMessage = document.createElement("div");
    userMessage.className = "message user";
    userMessage.innerHTML = `
        <div class="message-header">
            <i class="fas fa-user"></i> You
        </div>
        <div class="message-content">
            <div class="content-line">${userQuery}</div>
        </div>`;
    chatContainer.appendChild(userMessage);

    // Clear the input
    document.getElementById("queryInput").value = "";

    // Show a "Thinking..." bot message
    const loadingMessage = document.createElement("div");
    loadingMessage.className = "message bot";
    loadingMessage.innerHTML = `
        <div class="message-header">
```

```
            <i class="fas fa-robot"></i> Chatbot
        </div>
        <div class="message-content">
            <div class="content-line">
                <i class="fas fa-spinner fa-spin"></i> Thinking...
            </div>
        </div>`;
    chatContainer.appendChild(loadingMessage);
    loadingMessage.scrollIntoView({ behavior: 'smooth' });

    // Send query to server
    try {
        const response = await fetch("/chat", {
            method: "POST",
            headers: { "Content-Type": "application/json" },
            body: JSON.stringify({ query: userQuery })
        });

        const result = await response.json();
        chatContainer.removeChild(loadingMessage);

        // Show bot's response
        const botMessage = document.createElement("div");
        botMessage.className = "message bot";
        botMessage.innerHTML = formatMessage(
            result.answer || "I didn't quite get that.",
            result.references
        );
        chatContainer.appendChild(botMessage);
        botMessage.scrollIntoView({ behavior: 'smooth' });

    } catch (error) {
        chatContainer.removeChild(loadingMessage);

        const errorMessage = document.createElement("div");
        errorMessage.className = "message bot";
        errorMessage.innerHTML = formatMessage(
            "Error: Unable to fetch response. Please try again later."
        );
        chatContainer.appendChild(errorMessage);
        errorMessage.scrollIntoView({ behavior: 'smooth' });
    }
}

function clearChat() {
    chatContainer.innerHTML = "";
    header.classList.remove('minimized');
    chatContainer.classList.remove('header-minimized');
}

document.addEventListener("DOMContentLoaded", () => {
    const queryInput = document.getElementById("queryInput");
    queryInput.addEventListener("keydown", (event) => {
        if (event.key === "Enter") {
            event.preventDefault();
```

```
                    sendQuery();
                }
            });
        });
    </script>
</body>
</html>
```

This chatbot provides a user-friendly interface with a responsive design that adapts to different screen sizes. It feat

IV Code - Chatbot Development

```
# Importing necessary Flask modules for creating a web application,
# handling incoming requests, rendering HTML templates, and sending JSON responses
from flask import Flask, request, render_template, jsonify

# Importing the Elasticsearch client from the official Elasticsearch package
from elasticsearch import Elasticsearch

# Importing RequestsHttpNode which is used by Elasticsearch for making HTTP
    connections
from elastic_transport import RequestsHttpNode

# The requests library is used for making HTTP requests, such as sending the prompt to
     the custom LLM service
import requests

# The Retry utility is used to set up retry strategies for HTTP requests
from urllib3.util.retry import Retry

# The HTTPAdapter is used to mount custom configurations (like retries) onto a
    requests session
from requests.adapters import HTTPAdapter

# The dotenv library is used to load environment variables from a .env file
from dotenv import load_dotenv

# The argparse library is used for creating command-line interfaces to pass in
    arguments
import argparse

# The os module is used for environment variables and file path manipulations
import os

# Create a Flask application object. The 'app' variable will be used to register
    routes and run the server.
app = Flask(__name__)


class CustomLLM:
    """
    Custom LLM client that handles communication with your LLM endpoint.
    """
```

```python
# The __init__ method initializes the CustomLLM with the given endpoint, API key, and model ID.
# It also initializes a session_key, which may be used to maintain conversational context with the LLM.
def __init__(self, api_endpoint, api_key, api_model_id):
    self.endpoint = api_endpoint
    self.api_key = api_key
    self.model_id = api_model_id
    self.session_key = None

# This method takes a prompt string, optionally allows a retry mechanism (default True),
# and sends the prompt to the custom LLM service. It returns the LLM response as a string.
def generate_completion(self, prompt, retry=True):
    """
    Generates a completion using the custom LLM service.
    """
    # Convert the prompt to lowercase and strip whitespace.
    # If the user typed in a typical greeting, respond with a canned greeting message.
    lower_prompt = prompt.lower().strip()
    if lower_prompt in ['hello', 'hi', 'hey', 'greetings']:
        return "Hello! I am the Stages chatbot. I'm here to help you with any questions about the Stages. How can I assist you today?"

    # Build the JSON body to send to the LLM service, including the prompt, the API key, the session key (if any), and the model ID.
    json_body = {
        "api_key": self.api_key,
        "user_message": prompt,
        "session_key": self.session_key if self.session_key else "",
        "ai_model_id": self.model_id,
    }

    # Print out the session key being used, useful for debugging purposes.
    print(f"Sending request with session key: {self.session_key}")

    # Make a POST request to the endpoint with the JSON body and appropriate headers.
    response = requests.post(
        self.endpoint,
        json=json_body,
        headers={"Content-Type": "application/json"}
    )

    # If the status code is 200 (OK), parse the JSON response.
    # Update the session key if the service returned one. Return the response content.
    if response.status_code == 200:
        result = response.json()
        # Update session key if provided
        if result.get('session_key'):
            self.session_key = result['session_key']
            print(f"Received/Updated session key: {self.session_key}")
```

```
            return result.get('content', '')

        # If a 500 error occurs and retry is True, we consider it might be a session
           error.
        # We reset the session key and retry once more.
        elif response.status_code == 500 and retry:
            print("Session error encountered, retrying with new session...")
            self.session_key = None
            return self.generate_completion(prompt, retry=False)

        # If another error occurs, print it and raise an exception.
        else:
            print(f"Error response: {response.text}")
            raise Exception(f"LLM request failed with status code {response.
                status_code}")

    # Method to reset the session key, effectively resetting the conversation context.
    def reset_session(self):
        self.session_key = None
        print("Session key reset")


# This function builds and sends a multi_match query to Elasticsearch to find relevant
     documents.
def get_elasticsearch_results(query):
    """
    Retrieves relevant documents from Elasticsearch.
    """
    # Build a multi_match query object.
    # The 'best_fields' type merges the relevance scores from matched fields and picks
         the best combination.
    # The tie_breaker parameter allows results from multiple fields to be combined
        more effectively.
    es_query = {
        "query": {
            "multi_match": {
                "query": query,
                "fields": [
                    "body_content",
                    "title",
                    "headings",
                    "index._index",
                    "index._type"
                ],
                "type": "best_fields",
                "tie_breaker": 0.3
            }
        },
        # The 'size' key determines how many documents are returned.
        # Currently set to 3, but can be changed as per requirement.
        "size": 3
    }

    # Perform the search on the specified Elasticsearch index using the constructed
        query.
```

```
    # Note: Replace "your_index_name" with your actual Elasticsearch index name
    result = es_client.search(index=ES_INDEX_NAME, body=es_query)
    hits = result["hits"]["hits"]
    # Filter out any documents that have an empty or blank body_content.
    return [hit for hit in hits if hit["_source"].get("body_content", "").strip()]

# This function creates a structured prompt to feed to the LLM.
# It aggregates the relevant context from Elasticsearch hits and provides clear
    instructions for the LLM to answer.
from instructions import get_instructions

# Update the create_llm_prompt function
def create_llm_prompt(results, question):
    """
    Creates a structured prompt for the LLM using Elasticsearch results.
    """
    # Initialize an empty context string and a list for canonical URLs.
    context = ""
    canonical_urls = []

    # Loop through each Elasticsearch hit.
    for hit in results:
        # Get the source field name from index_source_fields dictionary based on the
            index name.
        source_field = index_source_fields.get(hit["_index"], [])[0]
        # Extract the context data (body_content) from the hit.
        hit_context = hit["_source"].get(source_field, "")
        # Append the body_content to our overall context.
        context += f"{hit_context}\n"
        # If there's a 'canonical_url', store it in canonical_urls list, provided it's
            not just a placeholder.
        canonical_url = hit["_source"].get("canonical_url", "")
        if canonical_url and canonical_url != "No URL provided":
            canonical_urls.append(canonical_url)

    # Fetch the instructions from the separate file and format them
    instructions = get_instructions().format(context=context, question=question)

    # Return the constructed prompt and the list of canonical URLs.
    return instructions, canonical_urls

# This is the root endpoint of our Flask application.
# When the home page is accessed, the session key for the LLM client is reset,
# and the index1.html template is rendered.
@app.route("/")
def home():
    llm_client.reset_session()
    return render_template("index1.html")

# The /chat endpoint is used for handling POST requests containing user queries.
@app.route("/chat", methods=["POST"])
def chat():
    # Extract the "query" field from the JSON payload of the request.
    user_query = request.json.get("query")
    try:
```

```
        # Print out which session key is being used for debugging.
        print(f"Processing query with session key: {llm_client.session_key}")

        # Lowercase and strip the query to check for simple greeting keywords.
        lower_query = user_query.lower().strip()
        if lower_query in ['hello', 'hi', 'hey', 'greetings']:
            # If the query is a greeting, return a greeting JSON response with an
                empty reference list.
            return jsonify({
                "answer": "Hello! I am the Stages chatbot. I'm here to help you with
                    any questions about the Stages process. How can I assist you today
                    ?",
                "references": []
            })

        # If it's not a greeting, proceed to get Elasticsearch results for the query.
        elasticsearch_results = get_elasticsearch_results(user_query)

        # If no results are returned, let the user know that no information could be
            found.
        if not elasticsearch_results:
            return jsonify({
                "answer": "I apologize, but I couldn't find any specific information
                    about that in my knowledge base. Could you please ask a more
                    specific question about the Stages process?",
                "references": []
            })

        # Create a prompt for the LLM using the Elasticsearch results and the user's
            query.
        prompt, canonical_urls = create_llm_prompt(elasticsearch_results, user_query)

        # Generate the LLM response.
        raw_answer = llm_client.generate_completion(prompt)

        # Return the LLM's answer along with any canonical URLs (references) in JSON
            format.
        return jsonify({
            "answer": raw_answer,
            "references": canonical_urls if canonical_urls else []
        })
    except Exception as e:
        # If something goes wrong, log the error and return a friendly error message.
        print(f"Error in chat endpoint: {str(e)}")
        return jsonify({
            "answer": "I apologize, but I'm having trouble processing your request.
                Please try again.",
            "references": []
        }), 500


# The main function is the entry point of the script when executed directly.
# It sets up and runs the Flask app, after configuring environment variables and
    command-line arguments.
def main():
```

```
    # Load environment variables from .env if available
    load_dotenv()

def environ_or_required(key):
    return (
        {'default': os.environ.get(key)} if os.environ.get(key)
        else {'required': True}
    )

def get_arg():
    # Load environment variables from .env file
    load_dotenv()
    parser = argparse.ArgumentParser(description="Run the chatbot application.")
    parser.add_argument('--es-host', type=str, help="Elasticsearch host URL.",**
        environ_or_required('ES_HOST'))
    parser.add_argument('--es-api-key', type=str, help="Elasticsearch API key.",**
        environ_or_required('ES_API_KEY'))
    parser.add_argument('--api-endpoint', type=str, help="LLM API endpoint.",**
        environ_or_required('AIM_API_ENDPOINT'))
    parser.add_argument('--api-key', type=str, help="LLM API key.",**
        environ_or_required('AIM_API_KEY'))
    parser.add_argument('--api-model-id', type=int, default=9, help="LLM model ID (
        default: 9).")
    parser.add_argument('--es-index', type=str, default=os.environ.get('ES_INDEX_NAME'
        , 'your_index_name'),
                        help="Elasticsearch index name.")

    return parser.parse_args()

if __name__ == "__main__":
    args = get_arg()

    # Set the global Elasticsearch index name
    ES_INDEX_NAME = args.es_index

    # A dictionary that maps Elasticsearch index names to the fields we want to
        extract from each document in that index.
    index_source_fields = {
        ES_INDEX_NAME: ["body_content"]  # Using the provided index name with its
            primary field
    }

    llm_client = CustomLLM(
        api_endpoint=args.api_endpoint,
        api_key=args.api_key,
        api_model_id=args.api_model_id
    )

    # Initialize the Elasticsearch client
    es_client = Elasticsearch(
        args.es_host,
        api_key=args.es_api_key
    )

    # Run the Flask app
```

```
    app.run(debug=True, host='0.0.0.0', port=5000)
```

V Code - Frontend design for Multiple LLM model selection

```
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8"/>
    <meta name="viewport" content="width=device-width, initial-scale=1.0"/>
    <title>Stages Chatbot</title>
    <link href="https://cdnjs.cloudflare.com/ajax/libs/font-awesome/6.0.0/css/all.min.
       css" rel="stylesheet"/>
    <style>
        @import url('https://fonts.googleapis.com/css2?family=Inter:wght@400;500;600&
            display=swap');

        * {
            margin: 0;
            padding: 0;
            box-sizing: border-box;
        }

        body {
            font-family: 'Inter', sans-serif;
            background: #fff;
            color: #333;
            line-height: 1.6;
            min-height: 100vh;
            display: flex;
            flex-direction: column;
            overflow-x: hidden;
        }

        .header {
            text-align: center;
            padding: 1rem;
            background: rgba(255, 255, 255, 0.95);
            backdrop-filter: blur(10px);
            border-bottom: 1px solid rgba(255, 165, 0, 0.2);
            position: fixed;
            width: 100%;
            top: 0;
            z-index: 999;
            box-shadow: 0 2px 10px rgba(255, 165, 0, 0.1);
            transition: all 0.3s ease;
            height: auto;
        }

        .header.minimized {
            transform: translateY(-100%);
        }

        h1 {
            font-size: 2rem;
```

```css
    color: #ff8c00;
    margin-bottom: 0.25rem;
    font-weight: 600;
}

.subtitle {
    color: #666;
    font-size: 0.9rem;
}

.chat-container {
    flex: 1;
    max-width: 1200px;
    margin: 4rem auto 0;
    padding: 1rem;
    width: 95%;
    display: flex;
    flex-direction: column-reverse;
    gap: 1rem;
    overflow-y: auto;
    height: calc(100vh - 8rem);
}

.message {
    padding: 1.5rem;
    border-radius: 1rem;
    max-width: 85%;
    animation: fadeIn 0.5s ease-in-out;
    box-shadow: 0 4px 6px rgba(0, 0, 0, 0.1);
}

@keyframes fadeIn {
    from { opacity: 0; transform: translateY(20px); }
    to { opacity: 1; transform: translateY(0); }
}

.user {
    background: rgba(255, 165, 0, 0.1);
    border: 1px solid rgba(255, 165, 0, 0.2);
    align-self: flex-end;
    margin-left: 15%;
}

.bot {
    background: #fff;
    border: 1px solid rgba(255, 165, 0, 0.2);
    align-self: flex-start;
    margin-right: 15%;
}

.message-header {
    display: flex;
    align-items: center;
    margin-bottom: 1rem;
    font-weight: 500;
```

```css
    color: #333;
}

.message-header i {
    margin-right: 0.5rem;
    color: #ff8c00;
}

.message-content {
    line-height: 1.6;
    color: #333;
}

.content-line {
    margin: 8px 0;
}

.list-item {
    display: flex;
    align-items: flex-start;
    margin: 8px 0;
    padding-left: 20px;
}

.bullet {
    color: #ff8c00;
    margin-right: 10px;
    margin-left: -20px;
}

.input-container {
    background: rgba(255, 255, 255, 0.95);
    backdrop-filter: blur(10px);
    border-top: 1px solid rgba(255, 165, 0, 0.2);
    padding: 1.5rem;
    position: sticky;
    bottom: 0;
    display: flex;
    gap: 1rem;
    max-width: 1000px;
    margin: 0 auto;
    width: 95%;
    box-shadow: 0 -2px 10px rgba(255, 165, 0, 0.1);
    align-items: center;
}

.model-buttons {
    display: flex;
    gap: 0.5rem;
    min-width: 220px;
}

.model-btn {
    padding: 0.5rem 1rem;
    border: 1px solid rgba(255, 165, 0, 0.3);
```

```
    border-radius: 1rem;
    font-size: 0.9rem;
    background: white;
    color: #666;
    cursor: pointer;
    transition: all 0.3s ease;
    flex: 1;
}

.model-btn.active {
    background: #ff8c00;
    color: white;
    border-color: #ff8c00;
    box-shadow: 0 2px 4px rgba(255, 140, 0, 0.2);
}

.model-btn:hover:not(.active) {
    border-color: #ff8c00;
    color: #ff8c00;
}

#queryInput {
    flex: 1;
    padding: 1rem 1.5rem;
    border: 1px solid rgba(255, 165, 0, 0.3);
    border-radius: 2rem;
    font-size: 1rem;
    transition: all 0.3s ease;
}

#queryInput:focus {
    outline: none;
    border-color: #ff8c00;
    box-shadow: 0 0 0 3px rgba(255, 140, 0, 0.1);
}

.action-buttons {
    display: flex;
    gap: 0.5rem;
}

.submit-btn, .clear-btn {
    padding: 0.75rem 1.25rem;
    border: none;
    border-radius: 1rem;
    font-size: 1rem;
    cursor: pointer;
    transition: all 0.3s ease;
    display: flex;
    align-items: center;
    gap: 0.5rem;
}

.submit-btn {
    background: linear-gradient(135deg, #ff8c00 0%, #ff6347 100%);
```

```css
    color: white;
}

.clear-btn {
    background: rgba(255, 255, 255, 0.9);
    color: #ff8c00;
    border: 1px solid rgba(255, 165, 0, 0.3);
}

button:hover {
    transform: translateY(-2px);
    box-shadow: 0 4px 12px rgba(255, 140, 0, 0.2);
}

.references-container {
    margin-top: 1.5rem;
    padding-top: 1rem;
    border-top: 1px solid rgba(255, 165, 0, 0.2);
}

.references-title {
    color: #666;
    font-weight: 500;
    margin-bottom: 0.5rem;
}

.reference-item {
    background: rgba(255, 255, 255, 0.9);
    padding: 0.75rem 1rem;
    border-radius: 0.5rem;
    margin: 0.5rem 0;
    border: 1px solid rgba(255, 165, 0, 0.2);
}

.reference-item a {
    color: #002fff;
    text-decoration: none;
    display: flex;
    align-items: center;
    gap: 0.5rem;
}

@media (max-width: 768px) {
    .input-container {
        flex-direction: column;
        gap: 0.75rem;
    }

    .model-buttons {
        width: 100%;
    }

    .action-buttons {
        width: 100%;
    }
```

```
            .submit-btn, .clear-btn {
                flex: 1;
                justify-content: center;
            }

            .message {
                max-width: 90%;
            }
        }
    </style>
</head>
<body>
    <div class="header">
        <h1>Stages Chatbot</h1>
        <p class="subtitle">Ask me anything specific about the stages!</p>
    </div>

    <div class="chat-container"></div>

    <div class="input-container">
        <div class="model-buttons">
            <button class="model-btn active" data-model="azure" onclick="selectModel(
                this)">Azure OpenAI</button>
            <button class="model-btn" data-model="aim" onclick="selectModel(this)">AIM
                </button>
        </div>
        <input type="text" id="queryInput" placeholder="Type your question here...">
        <div class="action-buttons">
            <button class="submit-btn" onclick="sendQuery()">
                <i class="fas fa-paper-plane"></i>
                Send
            </button>
            <button class="clear-btn" onclick="clearChat()">
                <i class="fas fa-trash"></i>
                Clear
            </button>
        </div>
    </div>

    <script>
        let selectedModel = 'azure';

        function selectModel(btn) {
            document.querySelectorAll('.model-btn').forEach(button => {
                button.classList.remove('active');
            });
            btn.classList.add('active');
            selectedModel = btn.dataset.model;
        }

        function minimizeHeader() {
            const header = document.querySelector('.header');
            header.classList.add('minimized');
        }
```

```
function showHeader() {
    const header = document.querySelector('.header');
    header.classList.remove('minimized');
}

function formatMessage(text, references = []) {
    function formatContent(content) {
        content = content.replace(/\*\*(.*?)\*\*/g, '<strong>$1</strong>');

        let lines = content.split('\n');
        let formattedLines = lines.map(line => {
            line = line.trim();
            if (!line) return '';

            if (line.startsWith('- ')) {
                return `<div class="list-item"><span class="bullet">   </span>
                    ${line.substring(2)}</div>`;
            }

            return `<div class="content-line">${line}</div>`;
        });

        return formattedLines.join('');
    }

    let formattedContent = formatContent(text);

    let html = `
        <div class="message-header">
            <i class="fas fa-robot"></i> Chatbot
        </div>
        <div class="message-content">
            ${formattedContent}
        </div>`;

    if (references && references.length > 0 && references.some(ref => ref &&
        ref.trim())) {
        html += `
            <div class="references-container">
                <div class="references-title">References:</div>
                <div class="reference-list">
                    ${references.filter(ref => ref && ref.trim()).map(ref => `
                        <div class="reference-item">
                            <a href="${ref}" target="_blank" rel="noopener
                                noreferrer">
                                <i class="fas fa-link"></i>
                                ${ref}
                            </a>
                        </div>
                    `).join('')}
                </div>
            </div>`;
    }
```

```
    return html;
}

async function sendQuery() {
    const userQuery = document.getElementById("queryInput").value;
    if (!userQuery.trim()) return;

    minimizeHeader();

    const chatContainer = document.querySelector(".chat-container");

    const userMessage = document.createElement("div");
    userMessage.className = "message user";
    userMessage.innerHTML = `
        <div class="message-header">
            <i class="fas fa-user"></i> You
        </div>
        <div class="message-content">
            <div class="content-line">${userQuery}</div>
        </div>`;
    chatContainer.prepend(userMessage);

    document.getElementById("queryInput").value = "";

    const loadingMessage = document.createElement("div");
    loadingMessage.className = "message bot";
    loadingMessage.innerHTML = `
        <div class="message-header">
            <i class="fas fa-robot"></i> Chatbot
        </div>
        <div class="message-content">
            <div class="content-line">
                <i class="fas fa-spinner fa-spin"></i> Thinking...
            </div>
        </div>`;
    chatContainer.prepend(loadingMessage);

    try {
        const response = await fetch("/chat", {
            method: "POST",
            headers: { "Content-Type": "application/json" },
            body: JSON.stringify({
                query: userQuery,
                model: selectedModel
            })
        });

        const result = await response.json();
        chatContainer.removeChild(loadingMessage);

        const botMessage = document.createElement("div");
        botMessage.className = "message bot";
        botMessage.innerHTML = formatMessage(result.answer || "I didn't quite
            get that.", result.references);
        chatContainer.prepend(botMessage);
```

```
                botMessage.scrollIntoView({ behavior: 'smooth' });
            } catch (error) {
                chatContainer.removeChild(loadingMessage);

                const errorMessage = document.createElement("div");
                errorMessage.className = "message bot";
                errorMessage.innerHTML = `
                    <div class="message-header">
                        <i class="fas fa-robot"></i> Chatbot
                    </div>
                    <div class="message-content">
                        <div class="content-line">
                            <i class="fas fa-exclamation-circle"></i> Error: Unable to
                                fetch response. Please try again later.
                        </div>
                    </div>`;
                chatContainer.prepend(errorMessage);
            }
        }

        function clearChat() {
            document.querySelector(".chat-container").innerHTML = "";
            showHeader();
        }

        document.addEventListener("DOMContentLoaded", () => {
            const queryInput = document.getElementById("queryInput");
            queryInput.addEventListener("keydown", (event) => {
                if (event.key === "Enter") {
                    event.preventDefault();
                    sendQuery();
                }
            });
        });
    </script>
</body>
</html>
```

VI Code - Backend for Multiple LLM model selection

```
from flask import Flask, request, render_template, jsonify
from elasticsearch import Elasticsearch
from openai import AzureOpenAI
from dotenv import load_dotenv
import os
import requests
from instructions import get_azure_prompt, get_aim_prompt  # Import instruction
    functions

app = Flask(__name__)

# AIM Client Class
class CustomLLM:
```

```python
    def __init__(self, endpoint, api_key, model_id):
        self.endpoint = endpoint
        self.api_key = api_key
        self.model_id = model_id
        self.session_key = None

    def generate_completion(self, prompt, retry=True):
        lower_prompt = prompt.lower().strip()
        if lower_prompt in ['hello', 'hi', 'hey', 'greetings']:
            return "Hello! I am the Stages chatbot. I'm here to help you with any
                questions. How can I assist you today?"

        json_body = {
            "api_key": self.api_key,
            "user_message": prompt,
            "session_key": self.session_key if self.session_key else "",
            "ai_model_id": self.model_id,
        }

        response = requests.post(
            self.endpoint,
            json=json_body,
            headers={"Content-Type": "application/json"}
        )

        if response.status_code == 200:
            result = response.json()
            if result.get('session_key'):
                self.session_key = result['session_key']
            return result.get('content', '')
        elif response.status_code == 500 and retry:
            self.session_key = None
            return self.generate_completion(prompt, retry=False)
        else:
            raise Exception(f"LLM request failed with status code {response.
                status_code}")

    def reset_session(self):
        self.session_key = None


def get_elasticsearch_results(query, es_client):
    es_query = {
        "query": {
            "multi_match": {
                "query": query,
                "fields": [
                    "body_content",
                    "title",
                    "headings",
                    "index._index",
                    "index._type"
                ],
                "type": "best_fields",
                "tie_breaker": 0.3
```

```
            }
        },
        "size": 3
    }
    result = es_client.search(index="on_prem_test", body=es_query)
    hits = result["hits"]["hits"]
    return [hit for hit in hits if hit["_source"].get("body_content", "").strip()]


def create_prompt(results, question, index_source_fields, model_type="azure"):
    """
    Creates a prompt for the LLM based on the results and the model type.
    :param results: The search results from Elasticsearch.
    :param question: The user's question.
    :param index_source_fields: Mapping of index names to source fields.
    :param model_type: The type of model to create the prompt for ("azure" or "aim").
    :return: A tuple of (prompt, canonical_urls).
    """
    context = ""
    canonical_urls = []

    for hit in results:
        source_field = index_source_fields.get(hit["_index"], [])[0]
        hit_context = hit["_source"].get(source_field, "")
        context += f"{hit_context}\n"
        canonical_url = hit["_source"].get("canonical_url", "")
        if canonical_url and canonical_url != "No URL provided":
            canonical_urls.append(canonical_url)

    # Use the appropriate instruction function based on the model type
    if model_type == "azure":
        prompt = get_azure_prompt(context)
    else:
        prompt = get_aim_prompt(context, question)

    return prompt, canonical_urls


@app.route("/")
def home():
    aim_client.reset_session()
    return render_template("index1.html")


@app.route("/chat", methods=["POST"])
def chat():
    data = request.json
    user_query = data.get("query")
    model_type = data.get("model", "azure")  # Default to azure if not specified

    try:
        lower_query = user_query.lower().strip()
        if lower_query in ['hello', 'hi', 'hey', 'greetings']:
            return jsonify({
                "answer": "Hello! I am the Stages chatbot. I'm here to help you with
```

```
                any questions. How can I assist you today?",
            "references": []
        })

    elasticsearch_results = get_elasticsearch_results(user_query, es_client)

    if not elasticsearch_results:
        return jsonify({
            "answer": "I apologize, but I couldn't find any specific information
                about that in my knowledge base. Could you please ask a more
                specific question?",
            "references": []
        })

    prompt, canonical_urls = create_prompt(elasticsearch_results, user_query,
        index_source_fields, model_type)

    if model_type == "azure":
        response = azure_client.chat.completions.create(
            model=AZURE_DEPLOYMENT_NAME,
            messages=[
                {"role": "system", "content": prompt},
                {"role": "user", "content": user_query}
            ],
            temperature=0.0,
            max_tokens=5000,
            top_p=0.95,
            frequency_penalty=0,
            presence_penalty=0,
            stop=None
        )
        answer = response.choices[0].message.content
    else:
        answer = aim_client.generate_completion(prompt)

    return jsonify({
        "answer": answer,
        "references": canonical_urls if canonical_urls else []
    })
except Exception as e:
    print(f"Error in chat endpoint: {str(e)}")
    return jsonify({
        "answer": "I apologize, but I'm having trouble processing your request.
            Please try again.",
        "references": []
    }), 500


if __name__ == "__main__":
    # Load environment variables
    load_dotenv()

    # Configuration variables
    ES_HOST = os.getenv('ES_HOST')
    ES_API_KEY = os.getenv('ES_API_KEY')
```

```
AZURE_ENDPOINT = os.getenv('AZURE_ENDPOINT')
AZURE_API_KEY = os.getenv('AZURE_API_KEY')
AZURE_API_VERSION = os.getenv('AZURE_API_VERSION')
AIM_API_ENDPOINT = os.getenv('AIM_API_ENDPOINT')
AIM_API_KEY = os.getenv('AIM_API_KEY')
AZURE_DEPLOYMENT_NAME = "gpt-35-turbo"
AIM_MODEL_ID = 9

# Initialize Elasticsearch client
es_client = Elasticsearch(
    ES_HOST,
    api_key=ES_API_KEY
)

# Initialize Azure OpenAI client
azure_client = AzureOpenAI(
    azure_endpoint=AZURE_ENDPOINT,
    api_key=AZURE_API_KEY,
    api_version=AZURE_API_VERSION
)

# Initialize AIM client
aim_client = CustomLLM(AIM_API_ENDPOINT, AIM_API_KEY, AIM_MODEL_ID)

# Elasticsearch index field mapping
index_source_fields = {
    "on_prem_test": ["body_content"]
}

# Start the Flask app
app.run(debug=True, host="0.0.0.0", port=5000)
```

VII Survey Questions

The following survey was conducted to evaluate the overall user experience, accuracy, and satisfaction of the chatbot implemented for BSH employees. The survey aimed to gather insights on the chatbot's usability, responsiveness, and the relevance of its responses.

Chatbot User Experience Survey

Thank you for participating in our chatbot evaluation survey. Your feedback will help us improve our service.

Section 1: User Experience 💻

1. How easy was it to use the chatbot?

Very Easy	Easy	Neutral	Difficult	Very Difficult
☐	☐	☐	☐	☐

2. Did the chatbot understand your queries correctly?

Yes	Most of the time	Occasionally	Rarely	No
☐	☐	☐	☐	☐

3. How would you rate the overall responsiveness of the chatbot?

Very Fast	Fast	Neutral	Slow	Very Slow
☐	☐	☐	☐	☐

4. Did you find the chatbot's interface intuitive and user-friendly?

Yes	No
☐	☐

5. How satisfied are you with the design and layout of the chatbot?

Very Satisfied	Satisfied	Neutral	Dissatisfied	Very Dis-satisfied
☐	☐	☐	☐	☐

6. Were you able to find the information you were looking for easily?

Yes	No
☐	☐

7. Did the chatbot provide enough context to help you understand the answers?

Yes	Most of the time	Occasionally	Rarely	No
☐	☐	☐	☐	☐

Section 2: Accuracy and Relevance ◎

1. How accurate were the chatbot's responses?

Very Accurate	Mostly Accurate	Neutral	Somewhat Inaccurate	Very In-accurate
☐	☐	☐	☐	☐

2. Did the chatbot give you the correct information on the first attempt?

Yes	Most of the time	Occasionally	Rarely	No
☐	☐	☐	☐	☐

3. How often did you have to rephrase your question to get the correct answer?

Never	Rarely	Occasionally	Frequently	Always
☐	☐	☐	☐	☐

4. How relevant were the chatbot's responses to your queries?

Highly Relevant	Mostly Relevant	Neutral	Somewhat Irrelevant	Completely Irrelevant
☐	☐	☐	☐	☐

5. Did the chatbot suggest helpful follow-up questions or responses?

Yes	No
☐	☐

Section 3: Satisfaction 😊

1. How satisfied are you with the chatbot's overall performance?

Very Satisfied	Satisfied	Neutral	Dissatisfied	Very Dis-satisfied
☐	☐	☐	☐	☐

2. Would you recommend the chatbot to a colleague?

Yes	No
☐	☐

3. Did the chatbot improve your efficiency or save you time compared to traditional search methods?

Yes	No
☐	☐

4. How confident are you in the chatbot's ability to handle complex queries?

Very Confident	Somewhat Confident	Neutral	Slightly Doubtful	Not Confident
☐	☐	☐	☐	☐

5. How well did the chatbot meet your expectations?

Exceeded Expectations	Met Ex-pectations	Neutral	Fell Short of Expectations	Did Not Meet Expectations
☐	☐	☐	☐	☐

Section 4: Open-Ended Feedback

1. What did you like most about the chatbot?

2. What did you find most frustrating about the chatbot?

3. What improvements would you suggest to make the chatbot more effective?

Thank you for completing this survey!