UNIVERSITY OF THESSALY

SCHOOL OF ENGINEERING

DEPARTMENT OF ELECTRICAL AND COMPUTER ENGINEERING

Enhancing Public Sector Transparency: Retrieval-Augmented Generation with Large Language Models on Diavgeia Greek Government Data.

Diploma Thesis

Kollimenos Lampros

Antoniou Georgios

Supervisor: Elias Houstis

September 2024

UNIVERSITY OF THESSALY

SCHOOL OF ENGINEERING

DEPARTMENT OF ELECTRICAL AND COMPUTER ENGINEERING

Enhancing Public Sector Transparency: Retrieval-Augmented Generation with Large Language Models on Diavgeia Greek Government Data.

Diploma Thesis

Kollimenos Lampros

Antoniou Georgios

Supervisor: Elias Houstis

September 2024

ΠΑΝΕΠΙΣΤΗΜΙΟ ΘΕΣΣΑΛΙΑΣ

ΠΟΛΥΤΕΧΝΙΚΗ ΣΧΟΛΗ

ΤΜΗΜΑ ΗΛΕΚΤΡΟΛΟΓΩΝ ΜΗΧΑΝΙΚΩΝ ΚΑΙ ΜΗΧΑΝΙΚΩΝ ΥΠΟΛΟΓΙΣΤΩΝ

Ενίσχυση της Διαφάνειας του Δημόσιου τομέα: Retrieval-Augmented Generation με Large Language Models στα δεδομένα της ελληνικής κυβέρνησης της Διαύγειας.

Διπλωματική Εργασία

Κολλημένος Λάμπρος

Αντωνίου Γεώργιος

Επιβλέπων: Ηλίας Χούστης

Σεπτέμβριος 2024

Approved by the Examination Committee:

Supervisor **Elias Houstis**
Emeritus Professor, Department of Electrical and Computer Engineering, University of Thessaly

Member **Emmanouil Vavalis**
Professor, Electrical and Computer Engineering, University of Thessaly

Member **Konstantinos Skianis**
Assistant Professor, Department of Computer Science and Engineering, University of Ioannina

Acknowledgements

We would like to express our sincere gratitude to the individuals who have been instrumental in the completion of this thesis.

First and foremost, we extend our deepest thanks to our supervisor, Emeritus Professor Elias Houstis of the Department of Electrical and Computer Engineering at the University of Thessaly. His dedication and expertise have greatly enriched our understanding of the subject matter. His extensive knowledge and constructive critiques have significantly enhanced the quality of our work.

We are also profoundly grateful to Assistant Professor Konstantinos Skianis of the Department of Computer Science and Engineering, University of Ioannina, a member of our thesis committee, for his thoughtful contributions and expert advice.His invaluable everyday guidance, unwavering support, and insightful feedback have been crucial throughout this research journey.

Our thanks also go to Professor Emmanouil Vavalis, whose suggestions have been vital in shaping our research. His keen insights and encouragement have been greatly appreciated.

We would also like to acknowledge the exceptional collaboration between us. Working together has been a rewarding experience, and our combined efforts and complementary skills have greatly contributed to the successful completion of this thesis.

Lastly, we would like to acknowledge the support of our families, whose patience and encouragement have sustained us through the challenges of this project.

Thank you all for your invaluable contributions and support.

DISCLAIMER ON ACADEMIC ETHICS AND INTELLECTUAL PROPERTY RIGHTS

«Being fully aware of the implications of copyright laws, I expressly state that this diploma thesis, as well as the electronic files and source codes developed or modified in the course of this thesis, are solely the product of my personal work and do not infringe any rights of intellectual property, personality and personal data of third parties, do not contain work / contributions of third parties for which the permission of the authors / beneficiaries is required and are not a product of partial or complete plagiarism, while the sources used are limited to the bibliographic references only and meet the rules of scientific citing. The points where I have used ideas, text, files and / or sources of other authors are clearly mentioned in the text with the appropriate citation and the relevant complete reference is included in the bibliographic references section. I also declare that the results of the work have not been used to obtain another degree. I fully, individually and personally undertake all legal and administrative consequences that may arise in the event that it is proven, in the course of time, that this thesis or part of it does not belong to me because it is a product of plagiarism».

The declarants

Kollimenos Lampros and Antoniou Georgios

Diploma Thesis

Enhancing Public Sector Transparency: Retrieval-Augmented Generation with Large Language Models on Diavgeia Greek Government Data.

Kollimenos Lampros

Antoniou Georgios

Abstract

In recent years, the demand for increased transparency in public governance has become a global imperative, driving governments to implement measures that enhance the accessibility and accountability of their administrative processes. The Greek government's Diavgeia initiative, which mandates the online publication of all government decisions, represents a significant step towards transparent governance. However, the vast and unstructured nature of the data generated by Diavgeia presents challenges in information retrieval, comprehension, and utilization for citizens, researchers, and policymakers alike.

This thesis addresses these challenges by proposing a novel approach that leverages Retrieval-Augmented Generation (RAG) with Large Language Models (LLMs) to improve the accessibility and usability of Diavgeia data. The research explores the integration of advanced AI technologies, specifically the application of RAG combined with LLMs, to develop an AI-powered chatbot capable of efficiently retrieving and generating relevant information from the extensive Diavgeia dataset.

The research's findings suggest that integrating RAG with LLMs can significantly improve the usability of government transparency platforms like Diavgeia, thereby fostering greater citizen engagement and facilitating informed decision-making.

Keywords:

Public Sector Transparency, Retrieval-Augmented Generation, Large Language Models, Diavgeia, Information Retrieval, Government Data

Διπλωματική Εργασία

Ενίσχυση της Διαφάνειας του Δημόσιου τομέα: Retrieval-Augmented Generation με Large Language Models στα δεδομένα της ελληνικής κυβέρνησης της Διαύγειας.

Κολλημένος Λάμπρος

Αντωνίου Γεώργιος

Περίληψη

Τα τελευταία χρόνια, η απαίτηση για αυξημένη διαφάνεια στη δημόσια διακυβέρνηση έχει γίνει παγκόσμια επιταγή, οδηγώντας τις κυβερνήσεις να εφαρμόσουν μέτρα που ενισχύουν την προσβασιμότητα και τη λογοδοσία των διοικητικών τους διαδικασιών. Η πρωτοβουλία «Διαύγεια» της ελληνικής κυβέρνησης, η οποία επιβάλλει την ηλεκτρονική δημοσίευση όλων των κυβερνητικών αποφάσεων, αποτελεί ένα σημαντικό βήμα προς τη διαφανή διακυβέρνηση. Ωστόσο, ο τεράστιος και αδόμητος χαρακτήρας των δεδομένων που παράγονται από τη Διαύγεια παρουσιάζει προκλήσεις όσον αφορά την ανάκτηση, την κατανόηση και τη χρήση πληροφοριών για τους πολίτες, τους ερευνητές και τους υπεύθυνους χάραξης πολιτικής.

Η παρούσα διατριβή αντιμετωπίζει αυτές τις προκλήσεις προτείνοντας μια νέα προσέγγιση που αξιοποιεί Retrieval-Augmented Generation (RAG) με Large Language Models (LLM) για τη βελτίωση της προσβασιμότητας και της χρηστικότητας των δεδομένων της Διαύγειας. Η έρευνα διερευνά την ενσωμάτωση προηγμένων τεχνολογιών τεχνητής νοημοσύνης, και συγκεκριμένα την εφαρμογή της RAG σε συνδυασμό με LLMs, για την ανάπτυξη ενός chatbot με τεχνητή νοημοσύνη, το οποίο είναι ικανό να ανακτά και να παράγει αποτελεσματικά σχετικές πληροφορίες από το εκτεταμένο σύνολο δεδομένων της Διαύγειας.

Τα ευρήματα αυτής της έρευνας υποδηλώνουν ότι η ενσωμάτωση της RAG με LLMs μπορεί να βελτιώσει σημαντικά τη χρηστικότητα των κυβερνητικών πλατφορμών διαφάνειας όπως η Διαύγεια, προωθώντας έτσι τη μεγαλύτερη εμπλοκή των πολιτών και διευκολύνοντας τη λήψη τεκμηριωμένων αποφάσεων.

Λέξεις-κλειδιά:

Διαφάνεια του δημόσιου τομέα, Διαύγεια, Ανάκτηση πληροφοριών, Κυβερνητικά δεδομένα

Table of contents

Acknowledgements **ix**

Abstract **xii**

Περίληψη **xiii**

Table of contents **xv**

List of figures **xxi**

List of tables **xxiii**

Abbreviations **xxv**

1 Introduction **1**

1.1 Problem 1

1.2 Technological Landscape 2

1.3 Thesis overview 4

2 Literature Review **7**

2.1 AI Assistants 7

2.1.1 Historical Development 7

2.1.2 Current Trends 8

2.2 Retrieval-Augmented Generation (RAG) 8

2.2.1 Concept and Mechanism 8

2.2.2 Advantages and Challenges 9

2.2.3 Applications and Case Studies 10

2.3 Large Language Models (LLMs) 10

2.3.1 Development and Capabilities 10
2.3.2 Applications and Case Studies 11
2.4 Chatbots in Government Services 11
2.4.1 Existing Applications 11
2.4.2 Case Studies 12
2.5 Diavgeia: Transparency and Accountability in the Greek Government . . . 12
2.5.1 Overview of Diavgeia 12
2.5.2 Challenges and Limitations in Current E-Government Systems . . . 13
2.5.3 Potential for AI Integration 13
2.6 Summary 14

3 OpenData API and Data 15
3.1 Diavgeia OpenData API 15
3.1.1 Comprehensive Repository of Governmental Decisions 15
3.1.2 Data Structure and Accessibility 16
3.1.3 Implications for Transparency and Accountability 16
3.2 API Description 17
3.3 Data Format 17
3.3.1 Decision Format 17
3.3.2 Organizations Struct Format 18
3.3.3 Units Struct Format 19
3.3.4 Signers Struct Format 20
3.3.5 Example of a JSON Response 21
3.4 Exploring Key Functions in OpenData Diavgeia API 23
3.4.1 get_simple_search_results Function 23
3.4.2 get_decision_type Function 24
3.4.3 get_unit Function 24
3.4.4 get_signer Function 25
3.4.5 get_organization Function 25
3.4.6 _get_resource 25
3.5 Summary 26

4 Data Crawling and Preprocess **27**

4.1 Crawler Implementation 27

4.1.1 Setup 27

4.1.2 Credentials and Logging 28

4.1.3 Fetching Data 28

4.1.4 Purpose of `time.sleep()` 29

4.1.5 Handling Multiple Pages of API Responses 30

4.1.6 Saving Data 31

4.2 Data Indexing Using Elasticsearch 33

4.2.1 Introduction to Elasticsearch 33

4.2.2 Setting Up Elasticsearch 35

4.2.3 Indexing Data in Elasticsearch 35

4.3 Conclusion 37

5 Architecture **39**

5.1 Retrieval-Augmented Generation (RAG) 40

5.1.1 Overview of RAG approach 40

5.1.2 RAG Architecture 41

5.1.3 Large Language Models (LLM) 42

5.1.4 RAG Advantages 43

5.1.5 Integration and Workflow 43

5.1.6 Search Concepts in Elasticsearch 44

5.2 The `search` Function in Elasticsearch Python Client 44

5.2.1 Importing the Client 44

5.2.2 Function Signature 45

5.2.3 Parameters 45

5.2.4 Return Value 45

5.2.5 Example Usage 46

5.2.6 Error Handling 48

5.3 Our ChatBot Approach with RAG 48

5.3.1 Elasticsearch Integration 48

5.3.2 Integrating OpenAI for Contextual Responses 50

5.3.3 Conversation History 52

5.4 Summary 57

6 System Architecture and Implementation 59

6.1 Overview of System Architecture 59

6.2 Backend Services 60

6.2.1 Flask Application 60

6.2.2 Dockerfile for Flask Application 60

6.2.3 Docker Compose for Multi-Container Setup 63

6.3 Flask Application Logic 67

6.3.1 run.py 67

6.4 Frontend Development 69

6.4.1 HTML Code 69

6.4.2 CSS Code 70

6.4.3 JavaScript Code 72

6.4.4 Overall interface 72

6.5 Summary 73

7 Evaluation, Testing, and Results 75

7.1 Evaluation Methodology 75

7.2 Evaluation Script for Structured Data 76

7.2.1 Custom Queries and Ground Truth 77

7.2.2 Results and Analysis 77

7.3 Evaluation for Random Queries 79

7.3.1 The Need for Automated Evaluation 79

7.3.2 Code Components and Their Functions 80

7.3.3 Results and Analysis 83

7.4 History service Evaluation 87

7.5 Conclusion 88

8 Additional Work 91

8.1 Implementing Streaming Responses 91

8.1.1 Introduction to Streaming Responses 91

8.1.2 Technical Implementation of Streaming in DiaygeiaBot 92

8.1.3 Benefits of Streaming Responses 94

8.1.4 Challenges and Considerations 94
8.2 Conclusion 94

Bibliography **97**

List of figures

4.1 Important info of a diavgeia document. 32
4.2 Data of a stored text file 33

5.1 This is our RAG approach 41

6.1 Indexer through Kibana service 66
6.2 Chatbot's interface 73

7.1 Output for an evaluation pair. 85
7.2 Average BertScores calculated by our script. 85
7.3 Precison Bertscore. 86
7.4 Recall BertScore. 86
7.5 F1 BertScore. 87
7.6 History service usage. 88

List of tables

3.1 Description of Decision Fields 18
3.2 Description of Organization Fields 19
3.3 Description of Unit Fields 20
3.4 Description of Signer Fields 21

Abbreviations

API	Application Programming Interface
RAG	Retrieval-Augmented Generation
LLM	Large Language Model
AI	Artificial Intelligence
JSON	JavaScript Object Notation
UID	Unique Identifier
HTTP	Hypertext Transfer Protocol
CSS	Cascading Style Sheets
HTML	Hypertext Markup Language
IDE	Integrated Development Environment
NLP	Natural Language Processing
BERT	Bidirectional Encoder Representations from Transformers
AWS	Amazon Web Services
GPT	Generative Pre-trained Transformer
ADA	Αριθμός Διαδικτυακής Ανάρτησης (Web Posting Number)
F1	Harmonic Mean of Precision and Recall

Chapter 1

Introduction

This chapter explores the challenges associated with the Diavgeia initiative and examines how emerging technologies can address these challenges. It sets the foundation for understanding the need for advanced information retrieval systems. It introduces the potential of LLMs and RAG techniques in improving the accessibility and utility of public sector data. Through this exploration, the chapter aims to highlight how these technologies can enhance transparency and accountability in governance.

1.1 Problem

In recent years, the demand for increased transparency in the public sector has emerged as a critical concern for governments and citizens worldwide. Transparency is not only a cornerstone of democratic governance but also a vital mechanism for fostering trust between the government and the public. In the context of Greece, the government's Diavgeia initiative, launched in 2010, represents a pioneering effort to enhance transparency by mandating the publication of all government decisions, contracts, and administrative acts online. This initiative has generated an extensive and growing repository of publicly accessible documents, intended to provide citizens, journalists, researchers, and other stakeholders with insight into the workings of the state.

However, while the Diavgeia initiative has significantly advanced the cause of transparency, it also presents new challenges. The vast volume of data generated—spanning multiple years, diverse government agencies, and a wide range of topics—has led to difficulties in information retrieval, comprehension, and utilization. The data is often unstructured, scat-

tered across different formats, and challenging to search effectively. This complexity can hinder the ability of citizens to find relevant information, understand government activities, and hold public officials accountable. Furthermore, researchers and policy analysts face significant obstacles in extracting actionable insights from this data, limiting the overall impact of the Diavgeia initiative.

The problem is further compounded by the limitations of traditional information retrieval systems, which struggle to process and present such large and diverse datasets meaningfully. Despite the availability of search engines and basic filtering tools, users often encounter difficulties in accessing the specific information they need, leading to an information overload. This situation underscores the need for more sophisticated methods to enhance the usability and accessibility of the Diavgeia dataset, ensuring that the promise of transparency is fully realized.

1.2 Technological Landscape

Addressing the challenges associated with accessing and utilizing the extensive data generated by government transparency initiatives, such as Diavgeia, has been facilitated by advancements in artificial intelligence (AI) and natural language processing (NLP). Among the most prominent solutions are Large Language Models (LLMs), which leverage vast amounts of text data to understand and generate human-like language. These models are grounded in the theory of NLP, an area of artificial intelligence that studies how language is understood by machines.

The goal of the specialised field of natural language processing (NLP) within artificial intelligence (AI) is to make it possible for computers to comprehend and interpret human language. By employing techniques such as tokenization, parsing, and entity recognition, NLP bridges the discrepancy between machine cognition and human communication. Advanced models in NLP use deep learning to analyze and generate text with high accuracy, facilitating complex tasks like summarization, translation, and information retrieval. This capability is crucial for managing large datasets and extracting meaningful insights, particularly from extensive public sector documents.

At the forefront of these technologies are models like OpenAI's GPT-4 and Google's BERT. GPT-4, the fourth iteration of OpenAI's Generative Pre-trained Transformer series,

builds upon its predecessors by incorporating a more extensive and diverse dataset, allowing it to generate more coherent and contextually relevant text. GPT-4 employs a transformer architecture that excels in understanding context, generating creative content, and answering complex queries with a high degree of accuracy (OpenAI, 2023). This model's ability, with its 175 billion parameters, to perform tasks such as summarization, translation, and information retrieval makes it particularly useful for managing large volumes of public sector documents, extracting pertinent details, and presenting them in an accessible format.[1]

Similarly, Google's BERT (Bidirectional Encoder Representations from Transformers) represents a significant advancement in NLP. Introduced in 2018, BERT utilizes a bidirectional approach to understand the context of words in a sentence, rather than processing them sequentially. This enables BERT to grasp nuances and ambiguities in language more effectively, enhancing its performance in tasks like question answering and language inference (Devlin et al., 2018). BERT's architecture allows it to excel in understanding the context and intent behind queries, making it an invaluable tool for parsing and analyzing extensive data sets generated by transparency initiatives.[2]

One of the most promising techniques for leveraging LLMs in this context is Retrieval-Augmented Generation (RAG). RAG combines the strengths of retrieval-based and generative models, allowing for more precise and context-aware responses. In a RAG system, the model first retrieves relevant documents or snippets from a large dataset using traditional search methods. It then uses these retrieved documents as context to generate a more accurate and detailed response. This approach is particularly effective for handling the unstructured and diverse nature of government data found in the Diavgeia initiative, as it allows the model to focus on specific, relevant information while still drawing on the broader dataset. By integrating RAG with LLMs, researchers and developers can create powerful tools that significantly enhance the accessibility of government documents, making it easier for users to find and understand the information they need.

Beyond LLMs and RAG, the availability of public APIs and open data platforms is crucial for transparency and innovation in the public sector. APIs, or Application Programming Interfaces, are tools that allow different software applications to communicate with each other by providing a set of rules and protocols for accessing specific data or services. The Greek government, through the Diavgeia initiative, has made a vast amount of data publicly available via APIs. These APIs give developers structured access to government decisions, con-

tracts, and other official documents, enabling the creation of custom applications and tools. By leveraging these APIs alongside LLMs and other AI technologies, it is possible to develop comprehensive systems that not only retrieve and display government data but also analyze trends, generate summaries, and offer insights to enhance public debate and inform policy-making.

Moreover, the integration of cloud computing platforms such as Amazon Web Services (AWS), Google Cloud, and Microsoft Azure further enhances the scalability and accessibility of these solutions. These platforms offer robust infrastructure for deploying LLMs and processing large datasets, ensuring that the solutions can handle the growing volume of government data without compromising performance. Additionally, cloud-based deployment enables real-time updates and access to the latest government publications, ensuring that the public always has access to the most current information. Together, these technologies form a powerful toolkit for addressing the challenges of public sector transparency, enabling more effective oversight, accountability, and engagement in democratic processes.

1.3 Thesis overview

This thesis addresses this challenge by exploring the application of cutting-edge artificial intelligence (AI) technologies, specifically Large Language Models (LLMs), to the problem of information retrieval and generation from the Diavgeia dataset. By leveraging Retrieval-Augmented Generation (RAG) techniques, which combine the strengths of retrieval-based systems and generative AI models, this research aims to develop innovative solutions that can significantly improve the way government data is accessed and utilized. The ultimate goal is to enhance the transparency and accountability of the Greek public sector by making government data more accessible, understandable, and actionable for all stakeholders.

A chatbot is a piece of software that uses voice or text interfaces to mimic human communication. It utilizes natural language processing (NLP) and machine learning algorithms to understand user queries, provide relevant responses, and perform various tasks autonomously. Chatbots can serve as virtual assistants that engage with users in real-time, offering information, support, and guidance based on the input they receive.

The main objective of the research is outlined as: to design an AI assistant bot for Diavgeia with the following capabilities of RAG and GPT models. The specific objectives of

the research include:

- Design the web crawler application for the Diavgeia data acquisition.
- Design a data indexing system for realization in the Docker environment using Elasticsearch. Docker is a platform that allows developers to package applications and their dependencies into containers, which are isolated and portable environments that ensure consistent operation across different computing environments.
- Build an AI chatbot based on RAG and GPT models that can give a proper, relevant, and correct response to the user's requests.
- These results will help measure performance and effectiveness in improving access to data imparted by the chatbot implemented within Diavgeia.

The key contribution of the research is:

- Experimentation that integrates advanced but raw NLP models closer to government data toward discovering information in the novel.
- Now, we will create a chatbot in Diavgeia to service the other governmental platforms.
- Tips on assessing the effectiveness of the government services chatbots.

The present thesis is outlined as follows:

- **Chapter 2: Literature Review** - In this chapter, we review the relevant literature on AI assistants, RAG, and LLMs and their applications across various domains.
- **Chapter 3: OpenData API and Data** - This chapter details the data collection and API integration process, focusing on the use of the Diavgeia OpenData API to support the chatbot's access to governmental decisions.
- **Chapter 4: Data Crawling and Preprocess** - This chapter describes the data crawling and preprocessing steps, including the implementation of a web crawler to extract government decisions from the Diavgeia platform and the subsequent indexing of this data into Elasticsearch for efficient querying.

- **Chapter 5: Architecture** - This chapter offers a thorough overview of the chatbot's architecture, focusing on using of Retrieval-Augmented Generation (RAG). It demonstrates how RAG enhances the system's ability to provide accurate and contextually relevant responses for the Diavgeia platform.
- **Chapter 6: System Architecture and Implementation** - This chapter provides a concise overview of the chatbot's architecture, encompassing backend, frontend, and Docker aspects.
- **Chapter 7: Evaluation, Testing, and Results** - This chapter details the evaluation methodology, testing process, and results for the DiaygeiaBot.
- **Chapter 8: Additional Work** - Testing features like streaming responses and other LLMs.

Chapter 2

Literature Review

The purpose of this literature review is to provide an overview of existing research and developments in the fields of AI assistants, Retrieval-Augmented Generation (RAG), and Large Language Models (LLMs). This review will focus on their applications in various domains, particularly in government services, to set the context for the development of our AI assistant bot for Diavgeia.

2.1 AI Assistants

AI assistants, also known as virtual assistants or chatbots, have gained significant attention due to their potential to automate and enhance human-computer interactions. Early AI assistants were rule-based, but recent advancements in machine learning have enabled the development of more sophisticated models.

2.1.1 Historical Development

The development of AI assistants can be traced back to the 1960s , when simple, rule-based systems such as ELIZA mimicked a psychotherapist by recognizing keywords and responding with pre-programmed phrases. This era marked the beginning of conversational agents designed to simulate human dialogue through pattern matching and scripted responses.

In the following decades, AI assistants evolved significantly with the integration of more advanced algorithms and natural language processing (NLP) techniques. The 1990s and early 2000s saw the emergence of assistants like Apple's Siri and Microsoft's Clippy, which utilized basic NLP and machine learning techniques to provide more dynamic and useful interactions.

2.1.2 Current Trends

Today, AI assistants leverage deep learning and large datasets to understand and generate human language more effectively. Models like Google Assistant, Amazon Alexa, and Apple Siri have become ubiquitous, utilizing sophisticated NLP algorithms and vast amounts of training data to handle a wide range of user queries.

Recent advancements include the integration of context-aware and multi-turn dialogue systems, which allow AI assistants to maintain context over multiple interactions and provide more coherent and relevant responses. Additionally, the development of Transfer Learning and pre-trained models has significantly improved the accuracy and efficiency of AI assistants [3].

Furthermore, the incorporation of Retrieval-Augmented Generation (RAG) techniques has enhanced the capabilities of AI assistants by enabling them to access and utilize external knowledge sources dynamically. This approach improves the factual accuracy and contextual relevance of responses, addressing common issues such as hallucinations and outdated information in generative models [4].

Moreover, recent studies emphasize the importance of developing AI assistants that are not only intelligent but also transparent and explainable. Ensuring that users understand how and why an AI assistant arrives at a particular response is crucial for building trust and facilitating widespread adoption, especially in sensitive domains like government services [5].

2.2 Retrieval-Augmented Generation (RAG)

The RAG model is a hybrid approach that combines retrieval-based and generation-based methods to provide more accurate and contextually relevant responses.

2.2.1 Concept and Mechanism

The RAG model operates by first retrieving relevant documents or passages from a large corpus based on the user's query. This retrieval step ensures that the model has access to factual and contextually relevant information. The second step involves generating a response using a neural network, such as a transformer model, which incorporates the retrieved information to produce a coherent and contextually appropriate reply [6].

By combining retrieval and generation, RAG models leverage the strengths of both approaches. The retrieval step grounds the model's responses in real-world data, reducing the likelihood of generating incorrect or nonsensical answers. The generation step allows for more natural and flexible interactions, as the model can generate responses that are tailored to the specific context of the query.

Recent research in open-domain question answering has demonstrated the efficacy of dense passage retrieval methods. For example, Karpukhin et al. (2020) showed that dense retrieval techniques outperform traditional sparse vector space models like BM25, resulting in significantly higher passage retrieval accuracy [7]. This advancement is crucial for improving the performance of RAG models, which rely on the quality of retrieved information to generate accurate responses.

2.2.2 Advantages and Challenges

Advantages:

- **Improved Accuracy:** RAG models enhance response accuracy by grounding outputs in up-to-date and relevant information retrieved from external databases or knowledge bases [4].
- **Dynamic Knowledge Integration:** They allow for real-time integration of new information without the need for retraining the entire model, making them highly adaptable to evolving data [5].
- **Reduced Hallucinations:** Relying on retrieved factual data makes RAG models less prone to generating fabricated or misleading information, a common issue in standard generative models.

Challenges:

- **Retrieval Quality Dependence:** The overall performance heavily depends on the quality and relevance of the retrieval component. Inaccurate or irrelevant retrieval can lead to poor responses.
- **Computational Complexity:** Integrating retrieval mechanisms increases computational requirements, which can impact response time and scalability.

- **Privacy Concerns:** Accessing and utilizing external data sources raises concerns about data privacy and security, especially when handling sensitive information [5].

2.2.3 Applications and Case Studies

RAG models have been applied in various domains, including customer support, education, and healthcare. In customer support, RAG models can provide accurate and helpful responses by retrieving information from knowledge bases and FAQs. In education, these models can assist students by answering questions based on textbooks and other learning materials. In healthcare, RAG models can support medical professionals by retrieving relevant medical literature and guidelines to inform clinical decisions [8].

Case studies have demonstrated the effectiveness of RAG models in improving the accuracy and relevance of AI-generated responses. For example, a study by Lewis et al. (2020) showed that the RAG model outperformed traditional retrieval-based and generation-based models in tasks such as open-domain question answering and dialogue systems [6].

2.3 Large Language Models (LLMs)

Large Language Models, such as OpenAI's GPT, have revolutionized the field of NLP by enabling the generation of human-like text.

2.3.1 Development and Capabilities

LLMs, including the Generative Pre-trained Transformer (GPT) series developed by OpenAI, are trained on vast amounts of text data using deep learning techniques. These models utilize transformer architectures, which allow them to handle long-range dependencies in text and generate coherent and contextually appropriate responses [9].

The capabilities of LLMs have expanded significantly with each iteration. GPT-3, for instance, has 175 billion parameters, enabling it to generate highly nuanced and context-aware text. LLMs can perform a wide range of tasks, including text completion, translation, summarization, and question answering, often with minimal fine-tuning.

Recent surveys have highlighted the evolution of large language models and their significant impact on the AI community. Zhao et al. (2023) discuss how model scaling has led to the development of LLMs with enhanced capabilities, such as in-context learning, which were

not present in smaller models like BERT. This evolution has been marked by the advent of models like GPT-3 and GPT-4, which have transformed the way AI algorithms are developed and utilized across various domains [10].

2.3.2 Applications and Case Studies

LLMs have been applied across various sectors, demonstrating their versatility and effectiveness. In the business sector, LLMs are used for automating customer support, generating marketing content, and analyzing customer feedback. In the creative industry, these models assist in generating stories, poems, and even music. In the scientific community, LLMs help researchers by summarizing papers, generating hypotheses, and automating literature reviews.

The development of specialized models like Codex, which is fine-tuned on code from public repositories like GitHub, demonstrates the potential of LLMs in more technical domains. Chen et al. (2021) showed that Codex, which powers tools like GitHub Copilot, significantly outperforms previous models in generating correct code from docstrings. This progress illustrates the growing applicability of LLMs in software development and the potential broader impacts on the industry, including safety, security, and economics [11].

Case studies highlight the transformative impact of LLMs. For example, OpenAI's GPT-3 has been used to develop applications such as chatbots that provide mental health support, virtual assistants that help with programming tasks, and systems that generate personalized educational content.

2.4 Chatbots in Government Services

The integration of chatbots in government services has the potential to enhance citizen engagement, facilitate access to information, and optimize administrative processes.

2.4.1 Existing Applications

Several governments worldwide have implemented chatbots to improve service delivery and citizen interaction. For example, the Estonian government has developed the virtual assistant KRATT to provide information and services to citizens across various platforms.

Similarly, the Singapore government uses chatbots like “Ask Jamie” to assist users on multiple agency websites.

These chatbots handle a wide range of tasks, from answering frequently asked questions to helping citizens navigate government websites and access services. The benefits include improved accessibility, 24/7 availability, and reduced workload for human agents.

2.4.2 Case Studies

Case studies from different countries illustrate the success and challenges of integrating chatbots into government services. In the United States, the Internal Revenue Service (IRS) implemented a chatbot to assist with tax-related inquiries, resulting in increased efficiency and user satisfaction. This chatbot, known as the IRS Virtual Assistant, utilizes natural language processing to provide accurate information and support to taxpayers [12].

In Australia, the government uses chatbots to support taxation, property rights, income and deductions, filing declarations, and tax-related services, enhancing digital inclusion and public communication [13].

The IRS has also implemented the Winnie chatbot for internal operations, which has improved efficiency in handling employee inquiries and processing internal data [14].

These case studies highlight the potential for chatbots to transform government-citizen interactions, but they also underscore the importance of addressing challenges such as data privacy, security, and the need for continuous updates and improvements [15].

2.5 Diavgeia: Transparency and Accountability in the Greek Government

Diavgeia is a Greek government initiative aimed at promoting transparency and accountability through the publication of administrative acts and decisions online.

2.5.1 Overview of Diavgeia

Launched in 2010, Diavgeia requires all government agencies to publish their decisions and acts online, making them accessible to the public. This initiative aims to enhance transparency, foster accountability, and improve public trust in government operations. The plat-

form includes a searchable database where users can find various administrative documents, including decisions, circulars, and official announcements.

2.5.2 Challenges and Limitations in Current E-Government Systems

While Diavgeia represents a significant step toward transparent governance, Tsironidou (2021) highlights several challenges common to e-Government systems that could also impact Diavgeia. One major issue is the usability of these platforms, as users often struggle with the interface and navigation, especially when dealing with large volumes of documents. Another challenge is the limited accessibility for individuals with disabilities or those unfamiliar with digital technologies. These limitations can hinder the overall effectiveness of such platforms in promoting transparency and citizen engagement [16].

2.5.3 Potential for AI Integration

Despite its benefits, Diavgeia's vast amount of data can be challenging for users to navigate. Integrating AI and chatbots with Diavgeia can significantly enhance its usability and accessibility. An AI assistant can help users find specific documents, answer queries about government decisions, and provide summaries of lengthy documents.

Moreover, the integration of Application Programming Interfaces (APIs) in Diavgeia could play a crucial role in this process. APIs enable seamless interaction between different government systems and public interfaces, allowing for automated retrieval and integration of public documents into third-party applications. This feature would be instrumental in enabling an AI assistant to provide accurate, up-to-date, and contextually relevant responses to user queries, directly accessing the latest information from the government database [17].

The authors by [17] also discuss the impact of emerging technologies such as blockchain, cloud computing, and data analytics in e-Government platforms. These technologies can further enhance the transparency and efficiency of platforms like Diavgeia by ensuring data integrity, scalability, and more sophisticated data processing capabilities. Blockchain, for instance, could be employed to secure government documents' integrity, ensuring that they cannot be altered or tampered with once published. Cloud computing could enable more efficient data storage and processing, allowing Diavgeia to handle the growing volume of administrative acts and decisions. Data analytics could be utilized to extract valuable insights from

the vast amount of data stored in Diavgeia, helping both citizens and government officials make more informed decisions [17].

The AI assistant can offer accurate and contextually relevant responses by leveraging RAG and LLM models, improving the overall user experience. This integration can further support Diavgeia's mission of transparency and accountability by making government information more accessible and understandable to the public.

2.6 Summary

This literature review has provided an overview of the key concepts and developments in AI assistants, RAG, LLMs, and their applications in government services. The insights gained from this review will guide us through the development of our AI assistant for Diavgeia.

Chapter 3

OpenData API and Data

This chapter focuses on the data collection process, API integration, and critical components for developing the Diavgeia chatbot. We describe the Diavgeia OpenData API, the implementation of the data crawler, the methods for data cleaning and preprocessing, and the strategies for storing and integrating data into the chatbot.

3.1 Diavgeia OpenData API

The Greek Diavgeia API is pivotal in enhancing transparency and accountability within the country's public administration. This section explores the content housed within the API and its profound implications.

3.1.1 Comprehensive Repository of Governmental Decisions

At its core, the Diavgeia API is as a gateway to a vast repository of governmental decisions spanning various public institutions and administrative levels. These decisions encompass a wide array of matters, including:

- **Public Procurement:** Detailed records of procurement decisions, bids, and contracts issued by governmental bodies, providing insights into public spending and vendor relationships.
- **Legal Actions:** Documentation of legal proceedings initiated or concluded by public authorities, ensuring visibility into judicial processes and outcomes.

- **Appointments and Personnel Changes:** Information on appointments, transfers, and dismissals within the public sector, promoting transparency in staffing decisions.
- **Regulatory and Administrative Acts:** Records of regulatory actions, administrative directives, and policy implementations affecting citizens and businesses.

3.1.2 Data Structure and Accessibility

The API facilitates structured access to this wealth of information through standardized data formats and protocols. Key features include:

- **Search and Retrieval Capabilities:** Users can query the API to retrieve specific decisions based on criteria such as date range, decision type, and issuing authority, enabling targeted research and analysis.
- **Real-Time Updates:** The API provides real-time access to newly published decisions, ensuring stakeholders have access to the latest information as it becomes available.
- **Metadata and Contextual Information:** Each decision entry includes metadata such as decision date, issuer, and decision number, supplemented by contextual information that enhances understanding and interpretation.

3.1.3 Implications for Transparency and Accountability

The availability of comprehensive decision data through the Diavgeia API has profound implications for governance, transparency, and accountability in Greece:

- **Enhanced Citizen Oversight:** Citizens can monitor governmental actions and decisions affecting their interests, promoting informed civic engagement and oversight.
- **Facilitated Research and Analysis:** Researchers and analysts can conduct in-depth studies on public policy, administrative practices, and regulatory compliance, contributing to evidence-based policymaking and academic inquiry.
- **Deterrence of Malpractice:** The API acts as a deterrent against corruption and malpractice by promoting visibility and scrutiny of governmental decisions, thereby fostering ethical conduct among public officials.

3.2 API Description

The Diavgeia OpenData API provides access to a wide range of government documents and decisions. The primary endpoints used in this project are:

- `/decisions`: Retrieves government decisions.
- `/organizations`: Provides information about government organizations.
- `/units`: Retrieves details about organizational units.
- `/signers`: Provides data about individuals who sign the decisions.
- `/decisionTypes`: Retrieves information about different types of decisions.

These endpoints return data in JSON format, which includes detailed information about each decision, such as the unique identifier (ADA), issue date, subject, status, and associated documents.

3.3 Data Format

In this section, we will present the format and structure of the data sourced from the Diavgeia OpenData API, which serves as the foundation for our thesis research. The API provides a structured dataset comprising essential details of governmental decisions, including administrative metadata, decision dates, and legal references. This structured format ensures the data's reliability and accessibility, facilitating comprehensive analysis and insights into governmental operations. By exploring this rich dataset, we aim to uncover valuable information regarding public policy trends and decision-making processes, crucial for understanding the dynamics of public administration and its implications.

3.3.1 Decision Format

Below you can see the main structure fields of every decision in the API:

Field	Description
ada	Αριθμός Διαδικτυακής Ανάρτησης (Διαδικτυακή Ανάρτηση means Web Posting Number)
protocolNumber	Αριθμός Πρωτοκόλλου (Αριθμός Πρωτοκόλλου means Protocol Number)
subject	Θέμα πράξης (Θέμα πράξης means Subject of action)
Issue Date	Ημερομηνία έκδοσης σε μορφή Unix Timestamp (Milliseconds from epoch)
decisionTypeId	Κωδικός του τύπου πράξης
organizationId	Κωδικός φορέα
unitIds	Λίστα κωδικών που αντιστοιχούν στις μονάδες οι οποίες εμπλέκονται στην έκδοση της συγκεκριμένης πράξης
signerIds	Λίστα κωδικών που αντιστοιχούν στους τελικούς υπογράφοντες της πράξης
extraFieldValues	Ενότητα ειδικών πεδίων. Το περιεχόμενο αυτής της ενότητας είναι ανάλογο του τύπου της πράξης.
submissionTimestamp	Ημερομηνία και ώρα τελευταίας τροποποίησης πράξης σε μορφή Unix Timestamp (Milliseconds from epoch)
status	Ένδειξη για την κατάσταση της πράξης. Οι τιμές μπορεί να είναι PUBLISHED, PENDING REVOCATION, REVOKED ή SUBMITTED
versionId	Αριθμός έκδοσης. Ο αριθμός αυτός αλλάζει σε περίπτωση που γίνει διάρθρωση των μεταδεδομένων μιας αναρτημένης πράξης, ή όταν αλλάξει η κατάστασή της

Table 3.1: Description of Decision Fields

3.3.2 Organizations Struct Format

Through the Diavgeia OpenData API, users can retrieve detailed information about these organizations, such as their structure, activities, and financial dealings. This data enables researchers, journalists, and the general public to monitor government actions, ensuring that

public funds are used responsibly and decisions are made transparently. For each organization registered in Diavgeia, the following information is provided:

Field	Description
uid	Μοναδικός κωδικός που ταυτοποιεί τον φορέα
label	Επωνυμία φορέα
abbreviation	Συντομογραφία φορέα
latinName	Συντομογραφία φορέα με λατινικούς χαρακτήρες
category	Κωδικός κατηγορίας φορέα
organizationDomains	Λίστα κωδικών τομέων αρμδιότητας
status	Ένδειξη για το αν ο φορέας είναι ενεργός
supervisorId	Κωδικός εποπτεύοντος φορέα
supervisorLabel	Επωνυμία εποπτεύοντος φορέα
website	Ιστότοπος φορέα
odeManagerEmail	E-mail υπεύθυνου ΟΔΕ φορέα
vatNumber	ΑΦΜ φορέα
fekNumber	Αριθμός σχετικού ΦΕΚ
fekIssue	Κωδικός τεύχους σχετικού ΦΕΚ
fekYear	Έτος έκδοσης σχετικού ΦΕΚ

Table 3.2: Description of Organization Fields

3.3.3 Units Struct Format

Within the framework of the Diavgeia (Διαύγεια) program, units refer to the various departments, divisions, and sub-entities that operate under larger public organizations such as ministries, municipalities, and other governmental bodies. The Diavgeia OpenData API allows users to access comprehensive data about these units, including their roles, decisions, and financial transactions:

Field	Description
uid	Μοναδικός κωδικός που ταυτοποιεί τη μονάδα
label	Επωνυμία μονάδας
abbreviation	Συντομογραφία μονάδας
active	Ένδειξη για το αν η μονάδα είναι ενεργή
activeFrom	Ημερομηνία και ώρα ενεργοποίησης μονάδας
activeUntil	Ημερομηνία και ώρα απενεργοποίησης μονάδας
category	Κωδικός κατηγορίας μονάδας
unitDomains	Λίστα κωδικών τομέων αρμδιότητας
parentId	Κωδικός φορέα ή μονάδας ανώτερου επιπέδου

Table 3.3: Description of Unit Fields

3.3.4 Signers Struct Format

In the Diavgeia (Διαύγεια) program, signers are the authorized individuals within public organizations who validate and approve decisions and documents. These signers play a critical role in the administrative process, ensuring that all actions and expenditures comply with legal and regulatory standards. The Diavgeia OpenData API provides access to detailed information about these signers, including their names, positions, and the specific decisions they have endorsed.

Field	Description
uid	Μοναδικός κωδικός που ταυτοποιεί τη μονάδα
lastName	Επώνυμο υπογράφοντα
firstName	Όνομα υπογράφοντα
active	Ένδειξη για το αν η μονάδα ο υπογράφων είναι εν ενεργεία
activeFrom	Ημερομηνία και ώρα ενεργοποίησης υπογράφοντα
activeUntil	Ημερομηνία και ώρα απενεργοποίησης υπογράφοντα
organizationId	Κωδικός φορέα στον οποίο υπάγεται ο υπογράφων
hasOrganizationSignRights	Ένδειξη για το αν ο υπογράφων έχει δικαίωμα υπογραφής σε όλες τις μονάδες του φορέα του
units	Οργανωτικές μονάδες στις οποίες ο υπογράφων έχει δικαίωμα υπογραφής . List of complex values, each having the following structure: • **uid:** Κωδικός μονάδας (ή φορέα, αν το hasOrganization-SignRights είναι true) όπου ο υπογράφων έχει δικαίωμα υπογραφής. • **positionId:** Κωδικός θέσης υπογράφοντα (Signer position ID) • **positionLabel:** Όνομα θέσης υπογράφοντα (Signer position label)

Table 3.4: Description of Signer Fields

3.3.5 Example of a JSON Response

This section provides an example of a JSON response from the Diavgeia API when querying for a decision. This example illustrates the typical structure and data fields returned by the API.

```
{   "ada": "ΒΛΕΖΝΡΝ-7",
    "protocolNumber": "Α.200",
    "subject": "Πράξη μετάταξης",
    "issueDate": 1380585600000,
    "organizationId": "30",
    "signerIds": [
        "28006"
    ],
    "unitIds": [
        "10000"
    ],
    "decisionTypeId": "Γ.3.5",
    "thematicCategoryIds": [
        "611"
    ],
    "extraFieldValues": {
        "eidosYpMetavolis": "Μετάταξη",
        "documentType": "ΠΡΑΞΗ",
        "relatedDecisions": [
            {
                "relatedDecisionsADA": "ΒΛΛΝΝ9-32"
            }
        ]
    },
    "privateData": false,
    "submissionTimestamp": 1389700140183,
    "status": "PUBLISHED",
    "versionId": "6deb2804-ef9b-4640-8a4e-92f2acecb970",
    "documentChecksum": "034a544f359fe152ca646347502653c5276c786a",
    "attachments": [
        {
            "id": "c5e2ad78-6c6b-452a-8ef9-4cae2b9b691a",
            "description": "Συνοδευτικό έγγραφο",
            "filename": "attachment.pdf",
            "mimeType": "application/pdf",
            "checksum": "e357f6261bcd123bb868535ddeb58471c3265ca7"
        }
    ] }
```

3.4 Exploring Key Functions in OpenData Diavgeia API

3.4.1 get_simple_search_results Function

The get_simple_search_results function is designed to perform a basic search operation within a dataset or database. This function typically accepts search parameters such as keywords or phrases and returns a list of items that match the search criteria. The implementation details can vary, but the core functionality revolves around querying a data source and filtering results based on the input parameters.

```
    def get_simple_search_results(self, **kwargs):
        """Performs search with the given criteria and returns the
           results.

        Keyword arguments:
        ada: Diavgeia decision identifier
        subject: Subject of the decision
        protocol: Protocol number
        term: General search term
        org: uid or latin name of the issuing organization
        unit: uid of the organization issuing organization unit
        signer: uid of the signer
        type: decision type uid
        tag: thematic category uid
        from_date: Search decisions published/edited/revoked
                   after this timestamp (format: YYYY-MM-DD)
        to_date: Search decisions published/edited/revoked
                 before this timestamp (format: YYYY-MM-DD)
        from_issue_date: Search decisions with issue date
                         after this date (format: YYYY-MM-DD)
        to_issue_date: Search decisions published/edited/revoked
                       before this date (format: YYYY-MM-DD)
        status: Accepted values are 'published', 'revoked',
               'pending_revocation', and 'all'. Default: 'all'
        page: Result page number. Default: 0
        size: Result page size. The default value is based on whether
              the client is authenticated or not
        sort: Accepted values are 'recent', 'relative'. Default: 'recent'
        """
```

```
        args = ["{0}={1}".format(kw, kwargs[kw]) for kw in kwargs]
        return self._get_resource("/search?" + "&".join(args))
```

3.4.2 get_decision_type Function

The get_decision_type function determines the type of decision associated with a given item or record. This could involve analyzing the content or metadata of an item to classify it into predefined categories such as "Approval", "Rejection", or "Pending". This function is crucial for applications that process or route items based on their decision type.

```
    def get_decision_type(self, type_id):
        """Returns the decision type with the specified ID.

        Arguments:
        type_id: uid of the decision type

        Output format:
          - uid: decision type identifier
          - label: decision type title
          - parent: identifier of parent type, if any
          - allowedInDecisions: true if the decision type
            can be used to publish decisions
        """
        return self._get_resource("/types/{0}/".format(type_id))
```

3.4.3 get_unit Function

The get_unit function aims to identify the unit or department associated with a particular record or item. This could involve parsing the item's details to extract information about the responsible unit, such as its name or code. This function is essential in contexts where organizational structure plays a role in the processing or analysis of items.

```
    def get_unit(self, unit_id):
        """Returns the unit with the specified ID.

        Arguments:
        unit_id: unit uid
        """
```

```
        return self._get_resource("/units/{0}/".format(unit_id))
```

3.4.4 get_signer Function

The get_signer function is designed to identify the individual who has signed or authorized a particular document or record. This might involve extracting signature information from the document's metadata or analyzing the document's content to find the signer's name. This function is particularly important in legal, financial, or administrative processes where authorization is a key factor.

```
    def get_signer(self, signer_id):
        """Returns the signer with the specified ID.

        Arguments:
        signer_id: signer uid
        """
        return self._get_resource("/signers/{0}/".format(signer_id))
```

3.4.5 get_organization Function

The get_organization function focuses on determining the organization associated with a specific record or item. This could involve analyzing the item to extract information about the organization, such as its name, ID, or other identifying details. This function is crucial for applications that need to categorize or route items based on organizational affiliation.

```
    def get_organization(self, org):
        """Returns the organization with the specified uid or latin name.

        Arguments:
        org: uid or latin name of an organization
        """
        return self._get_resource("/organizations/{0}/".format(org))
```

3.4.6 _get_resource

The get_resource function is a private method within a class designed to interact with a web API. It serves as a utility function to make HTTP GET requests to specific resources

on the API. The function takes two parameters: resource, which is a string representing the specific API endpoint to be accessed, and addheaders, a dictionary allowing for additional HTTP headers to be specified alongside the default headers.

This function is designed to be used internally within the class to abstract the complexities of making HTTP requests and handling responses. It allows other methods of the class to access various resources on the API by providing a simple interface to make authenticated requests and parse JSON responses.

```
    def _get_resource(self, resource, addheaders={}):
        headers = self.default_headers.copy()
        for addh in addheaders.keys():
            headers[addh] = addheaders[addh]
        response = requests.get(
            self._get_resource_url(resource),
            auth=self._create_auth(),
            headers=headers,
            verify=False,
        )
        #import ipdb; ipdb.set_trace()
        print(response)
        return response.json()
```

3.5 Summary

Overall, this chapter highlighted the meticulous approach to data collection and API integration, emphasizing the importance of structured and accessible data for the chatbot's knowledge base. By leveraging the comprehensive data from the Diavgeia OpenData API, we have established a solid foundation for a chatbot that enhances the accessibility and transparency of governmental decisions, fostering a more informed and engaged citizenry.

Chapter 4

Data Crawling and Preprocess

In this chapter, we present the process of data crawling from the Diavgeia platform, a crucial step in building our AI assistant bot. The Diavgeia platform publishes government decisions, and our goal is to download, extract, and index this content for further processing. This involves using a web crawler to automate the retrieval of data and storing it in a structured format.

4.1 Crawler Implementation

Our crawler is implemented in Python and utilizes the `OpendataClient` class to interact with the Diavgeia API. The process involves several key steps:

- **Setup and Configuration:** Import necessary modules, set the path for the project, and initialize the client with credentials.
- **Fetching Data:** Make multiple requests to the Diavgeia API to fetch data, incrementing the page number with each request.
- **Saving Data:** Extract and save all information from each decision into structured text files.

Below, we provide an overview of the implementation along with relevant code excerpts.

4.1.1 Setup

First, we import the necessary modules and set the path for the project.

```
import sys
import os
import datetime
import time

# Setting path
sys.path.append('your_path_to_project')

from diaygeia.crawlers.opendata import OpendataClient,
    extract_text_from_pdf_url
from diaygeia.logging_tools import get_logger
```

4.1.2 Credentials and Logging

We initialize the logger and set the credentials for the `OpendataClient`.

```
if __name__ == "__main__":
    logger = get_logger("diaygeia_opendata")
    client = OpendataClient()
    client.set_credentials("USERNAME","PASSWORD")
```

4.1.3 Fetching Data

We start fetching data from the API by making multiple requests with increasing page numbers until no more data is available. The results are saved in a specified download path.

```
page = 0
download_path =
    "C:/Users/lkoll/PycharmProjects/diaygeia/diaygeia/entrypoints/data/"

while True:
    data = client.get_simple_search_results(
        from_issue_date="2024-01-01", to_issue_date="2024-01-02",
            page=page, size=500
    )["decisions"]

    if not data:  # If no more data, break the loop
        print("No more data")
```

```
        break
```

4.1.4 Purpose of `time.sleep()`

When interacting with APIs, especially those that impose rate limits, it is crucial to manage the frequency of requests to avoid exceeding these limits. The `time.sleep()` function is used to introduce a delay between successive requests. This delay helps in several ways:

- **Avoid Rate Limiting:** Many APIs have rate limits to prevent abuse and ensure fair usage. By introducing a delay, we can ensure that our request rate remains within acceptable limits.
- **Prevent Server Overload:** Rapid successive requests can overwhelm the server, leading to slower responses or even temporary bans. `time.sleep()` helps distribute the requests over time, reducing the load on the server.
- **Improve Reliability:** APIs might reject requests if they are too frequent. Introducing a delay improves the reliability of the data fetching process, ensuring that each request is processed successfully.

In the given script, `time.sleep(1)` is used before making requests to retrieve decision types, units, signers, and organizations. This helps in complying with any potential rate limiting policies of the API and ensures smooth and consistent data retrieval.

Implementation in the Script

Here is a snippet from the script showing how `time.sleep()` is used:

```
for d in data:
    ...
    time.sleep(1)
    decision_type = client.get_decision_type(d["decisionTypeId"])
    ...
    for i in range(len(d["unitIds"])):
        time.sleep(1)
        unit = client.get_unit(d["unitIds"][i])
        ...
    for i in range(len(d["signerIds"])):
```

```
        time.sleep(1)
        signer = client.get_signer(d["signerIds"][i])
        ...
    time.sleep(1)
    organization = client.get_organization(d["organizationId"])
    ...
```

Listing 4.1: Usage of `time.sleep()` in the Script

Each call to `client.get_decision_type`, `client.get_unit`, `client.get_signer`, and `client.get_organization` is preceded by `time.sleep(1)`. This ensures a one-second delay between each API request, reducing the likelihood of hitting rate limits or overloading the server.

4.1.5 Handling Multiple Pages of API Responses

Pagination in API Requests

APIs often return results in pages, with a fixed number of results per page. To retrieve all the data, especially when dealing with large datasets, the script needs to handle pagination. This involves making repeated requests, incrementing the page number until no more results are returned.

Implementation in the Script

The script handles pagination using a loop that continues until no more data is returned from the API. Here is the relevant part of the script:

```
page = 0
while True:
    data = client.get_simple_search_results(
        from_issue_date="2024-01-01", to_issue_date="2024-01-02",
           page=page, size=500
    )["decisions"]

    if not data:  # If no more data, break the loop
        print("No more data")
        break

```

```
    for d in data:
        file_path = download_path + d["ada"] + ".txt"
        if not os.path.exists(file_path):
            ...
            f.close()
        else:
            print(f"Ada {d['ada']} already downloaded")
    print(f"Page {page} done\n")
    page += 1  # Increment the page number
```

Listing 4.2: Handling Pagination in the Script

- **Initialization:** The `page` variable is initialized to 0.
- **Loop Until No More Data:** The `while True` loop runs indefinitely until it explicitly breaks out when no more data is returned (`if not data`).
- **API Request with Pagination:** `client.get_simple_search_results` is called with the current `page` number and a fixed `size` of 500. This fetches the results for the specified page.
- **Processing Data:** The script processes each decision in the data. If a decision has not already been downloaded (`if not os.path.exists(file_path)`), it fetches additional details and writes them to a file.
- **Incrementing Page Number:** After processing all decisions on the current page, the script increments the `page` number (`page += 1`) and proceeds to the next iteration, fetching the next page of results.

By handling pagination this way, the script ensures that all available data is retrieved, even if it spans multiple pages. This approach is efficient and ensures completeness in data collection.

4.1.6 Saving Data

For each decision fetched, we save the information in a text file. We include important details such as the ADA, protocol number, issue date, subject, status, and URLs and also the whole text of the decision.

Saving the Document's Main Information

The code first formats and writes the main information of a document into a file. This includes details such as submission date, publish date, subject, status, URL, decision type, organizational units involved, signatories, organization name, budget category, financial year, entry number, and total amount with VAT.In figure 4.1

```
ΑΔΑ: 6696469ΗΕΒ-ΝΨ9
Αριθμός Πρωτοκόλλου: 41/2024
Ημερομηνία έκδοσης: 2024-01-01
Ημερομηνία τελευταίας τροποποίησης: 2024-01-01 01:34:57
Ημερομηνία ανάρτησης: 2024-01-01 01:34:57
Θέμα: Ανάληψη Υποχρέωσης #1675816 #195384
Κατάσταση: PUBLISHED
url: https://diavgeia.gov.gr/decision/view/6696469ΗΕΒ-ΝΨ9
Είδος: ΑΝΑΛΗΨΗ ΥΠΟΧΡΕΩΣΗΣ
Οργανωτικές Μονάδες: ΕΙΔΙΚΟΣ ΛΟΓΑΡΙΑΣΜΟΣ ΚΟΝΔΥΛΙΩΝ ΕΡΕΥΝΑΣ /
Υπογράφοντες: ΜΑΡΙΝΑ ΦΟΝΤΑΡΑ - Προϊστάμενος Γραμματείας/
Φορέας: ΕΚΕΦΕ ΔΗΜΟΚΡΙΤΟΣ
Κατηγορία Προϋπολογισμού: Ίδια Έσοδα
Οικονομικό έτος: 2024
Αριθμός Καταχώρησης: None
Συνολικό ποσό: {'amount': 103505.88, 'currency': 'EUR'}
```

Figure 4.1: Important info of a diavgeia document.

Extracting and Saving Text from the Document's PDF

After saving the main information, the code checks if the document has an associated PDF URL (if len(d["documentUrl"]) > 0). If so, it proceeds to extract text from the PDF. The text extraction from the PDF is done by a function extract_text_from_pdf_url(d["ada"], d["documentUrl"]), which takes the document's unique identifier (d["ada"]) and the URL to the PDF (d["documentUrl"]) as arguments:

```
1 def extract_text_from_pdf_url(url):
2     # Download the PDF from the URL
3     response = requests.get(url)
4     response.raise_for_status()  # Ensure the download succeeded
5 
6     # Create a temporary file to save the PDF
7     with tempfile.NamedTemporaryFile(suffix=".pdf") as temp_pdf:
8         temp_pdf.write(response.content)
9         temp_pdf.seek(0)  # Go to the beginning of the file
```

```
10
11          # Open the PDF with fitz
12          with fitz.open(temp_pdf.name) as doc:
13              text = ""
14              # Extract text from each page
15              for page in doc:
16                  text += page.get_text()
17      return text
```

After this step the stored data appears in a .txt file as shown in the figure below: 4.2

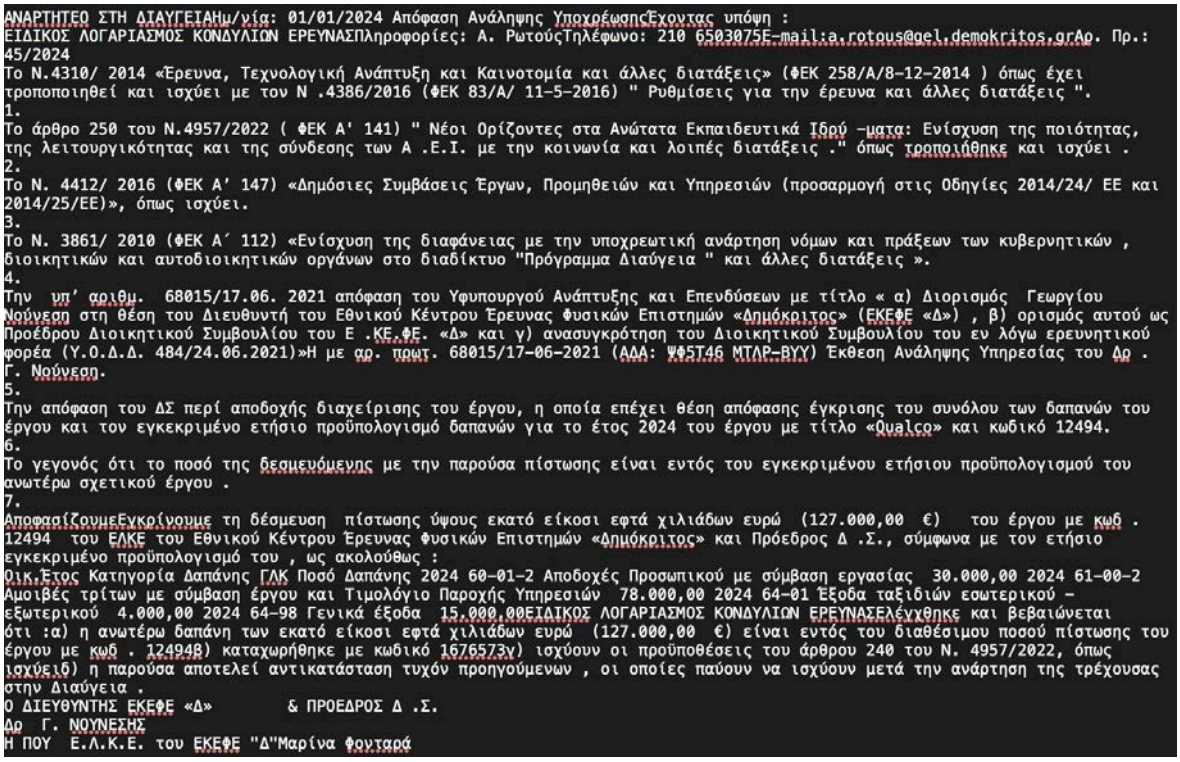

Figure 4.2: Data of a stored text file

4.2 Data Indexing Using Elasticsearch

4.2.1 Introduction to Elasticsearch

Elasticsearch is a highly scalable, open-source, full-text search and analytics engine based on Apache Lucene. It is designed to facilitate real-time search, analysis, and visualization of large volumes of data. Elasticsearch is widely adopted across various industries for its speed, scalability, and ease of use.

Architecture and Core Concepts

Elasticsearch is built on a distributed architecture, making it capable of handling extensive datasets across multiple nodes. The core components of Elasticsearch include:

- **Index**: Similar to a database in the context of a relational database, an index in Elasticsearch is a collection of documents that share similar characteristics.
- **Document**: The basic unit of information that can be indexed. Each document is a JSON object consisting of fields and values.
- **Shard**: Indices are subdivided into shards, which can be distributed across multiple nodes. Shards allow Elasticsearch to horizontally scale and distribute data for parallel processing.
- **Replica**: For high availability and fault tolerance, each shard can have multiple replicas. Replicas ensure that the system can recover quickly in the event of a hardware failure.

Key Features

Elasticsearch offers a range of features that make it a powerful tool for various applications:

- **Full-Text Search**: Elasticsearch provides sophisticated full-text search capabilities, including support for multiple languages, relevance scoring, and phrase matching.
- **Real-Time Data**: It supports near real-time indexing and searching, making it suitable for applications requiring immediate data insights.
- **Scalability and Flexibility**: With its distributed nature, Elasticsearch can scale horizontally by adding more nodes. This allows it to handle large volumes of data and high query loads efficiently.
- **Extensible Query Language**: It provides a rich query DSL (Domain Specific Language) based on JSON to define complex queries and filters.

4.2.2 Setting Up Elasticsearch

Integrating Docker with Elasticsearch service significantly enhances the ease of deployment and management of Elasticsearch instances. Docker, a platform for developing, shipping, and running applications in isolated containers, allows Elasticsearch to be encapsulated in a lightweight, portable, and self-sufficient environment. This encapsulation simplifies the setup process, as Elasticsearch and its dependencies can be packaged together, ensuring consistency across different development and production environments. Additionally, Docker facilitates scaling Elasticsearch services horizontally by enabling the creation of multiple container instances across distributed systems, which aligns perfectly with Elasticsearch's distributed architecture. This combination ensures that Elasticsearch can efficiently handle large volumes of data and high query loads, offering robustness and flexibility in various application scenarios. To set up Elasticsearch using Docker, we used the following configuration:

```
elasticsearch:
    container_name: document_store_bm25-container
    image: docker.elastic.co/elasticsearch/elasticsearch:7.11.0
    environment:
      - xpack.security.enabled=false
      - "discovery.type=single-node"
    ports:
      - "9200:9200"
    volumes:
      - elasticsearch-data:/usr/share/elasticsearch/data
      - elasticsearch-logs:/usr/share/elasticsearch/logs
```

4.2.3 Indexing Data in Elasticsearch

To index data into Elasticsearch, we first need to establish a connection to an Elasticsearch instance and define the necessary index configuration. The following Python code demonstrates how to connect to an Elasticsearch server, create an index with specified settings and mappings, and iterate through a directory of text files to index each document into Elasticsearch. The indexing is done using python in the following script:

```
import os
from elasticsearch import Elasticsearch
```

```
3
4  # Create a connection to Elasticsearch
5  es = Elasticsearch("http://localhost:9200")
6
7  # Index name
8  index_name = "diaygeia"
9
10 if not es.indices.exists(index=index_name):
11     es.indices.create(index=index_name, body=index_config)
12
13 # Iterate over downloaded data and index each document
14 path = "path_to_files"
15 for file_name in os.listdir(path):
16     if file_name.endswith(".txt"):
17         with open(os.path.join(path, file_name), "r", encoding="utf-8")
               as file:
18             key = file_name[:-4]  # Extract key from file name (remove
                   ".txt" extension)
19             content = file.read()
20             doc = {
21                 "content": content
22             }
23             # Indexing the document
24             es.index(index=index_name, id=key, document=doc)
25
26 print("Indexing complete.")
```

After running this script, all Diavgeia documents that we have obtained through crawling are now indexed in Elasticsearch. This enables us to leverage Elasticsearch's robust search and analytics features to efficiently query and analyze the indexed documents.

Elasticsearch.index Function

When using Elasticsearch, indexing is the process of adding a JSON document to a designated index so that it may be found through searches inside the Elasticsearch ecosystem. This process is essential to Elasticsearch data management since it makes it possible for the system to efficiently store, update, and retrieve documents. The es.index() function is essential to this procedure. It adds a document to the specified index, which is usually represented

as a Python dictionary. The procedure adds the new content to the existing document and increases its version number if it is already present in the index. Data consistency must be maintained by using this versioning method, particularly in dispersed situations where concurrent updates may take place. There are multiple steps in the indexing process. The JSON format of the document is first created, containing the data fields and their corresponding values. Subsequently, the target index and document ID are specified when the es.index() method is used. Elasticsearch will generate an ID automatically if one is not supplied. After that, the process updates the underlying data structures that provide quick search and retrieval operations and saves the document in the index. Elasticsearch uses this function to make sure the document is included in the dataset that can be searched, so users can query it in addition to other documents that have been indexed. All things considered, Elasticsearch's data storage and retrieval capabilities are supported by the robust es.index() function, which enables effective document management and real-time searchability.

4.3 Conclusion

In this chapter, we detailed the implementation of a Python-based web crawler designed to extract government decisions from the Diavgeia platform. Through meticulous setup, credential management, and systematic pagination handling, our crawler efficiently retrieves decision data while adhering to API constraints. By incorporating features such as metadata extraction and optional PDF text parsing, we ensure comprehensive data capture for further analysis. Additionally, we demonstrated how to index the retrieved documents into Elasticsearch. This indexing process allows for efficient querying and analysis of the Diavgeia documents, thereby enhancing the value of the collected data.

Chapter 5

Architecture

This chapter delves into the technical execution and architectural design of our chatbot system, which was created for the Greek government platform Diavgeia. A chatbot's architecture is a key factor in determining its functionality, dependability, and performance. This chapter gives a thorough description of the system's architecture, outlining each component's functions and the reasoning behind the design choices made.

To give precise and contextually relevant answers to user queries, our chatbot system uses a Retrieval-Augmented Generation (RAG), which combines the best features of both retrieval-based and generative models. The chatbot can answer various inquiries on decisions and actions made by the government administrative branch by utilising information from the Diavgeia open data API. The chatbot's ability to produce human-like responses is further improved by the usage of Large Language Models (LLMs), namely the GPT-3.5 API, which guarantees a smooth and interesting user experience.

We start by providing an overview of the system architecture and showing how different parts work together to process and reply to customer inquiries. Preprocessing, which involves cleaning and organising raw data from the Diavgeia API so that it can be used in the model, is included. After that, we go to great lengths on the RAG technique, outlining how the retrieval and generation components are integrated and the advantages of this hybrid method.

The technical details of the retriever and generator components are also covered in this chapter, along with an explanation of the strategies and techniques used for information retrieval and response production. We also go over the LLM integration, covering the methods for training and fine-tuning it to fit our particular application.

In conclusion, this chapter aims to present a thorough and technical explanation of the

model architecture while emphasizing its importance in accomplishing the goals of our chatbot project for the Diavgeia platform. Through this detailed review, we show how architectural decisions affect the overall performance and usefulness of the chatbot.

5.1 Retrieval-Augmented Generation (RAG)

The Retrieval-Augmented Generation (RAG) architect is a pivotal component of our chatbot system, leveraging both retrieval-based and generative approaches to enhance response accuracy and relevance. This section provides an in-depth discussion of the RAG approach, including its architecture, its advantages, and its implementation within our system.

5.1.1 Overview of RAG approach

RAG combines two powerful techniques in natural language processing: retrieval of relevant information and generating of coherent responses. By integrating these approaches, RAG can produce more informed and contextually appropriate answers than either method alone. The model operates in two main stages:

1. **Retrieval Stage**: The model retrieves relevant documents or passages from a pre-indexed corpus based on the user's query.

2. **Generation Stage**: The retrieved information is then used as input to a generative model, which produces a coherent and contextually relevant response.

This dual approach allows RAG to leverage large-scale knowledge bases while maintaining generative models' flexibility and creativity.
We can see the Combination of Retrieval and Generation for Advanced ML Models in the next figure: 5.1

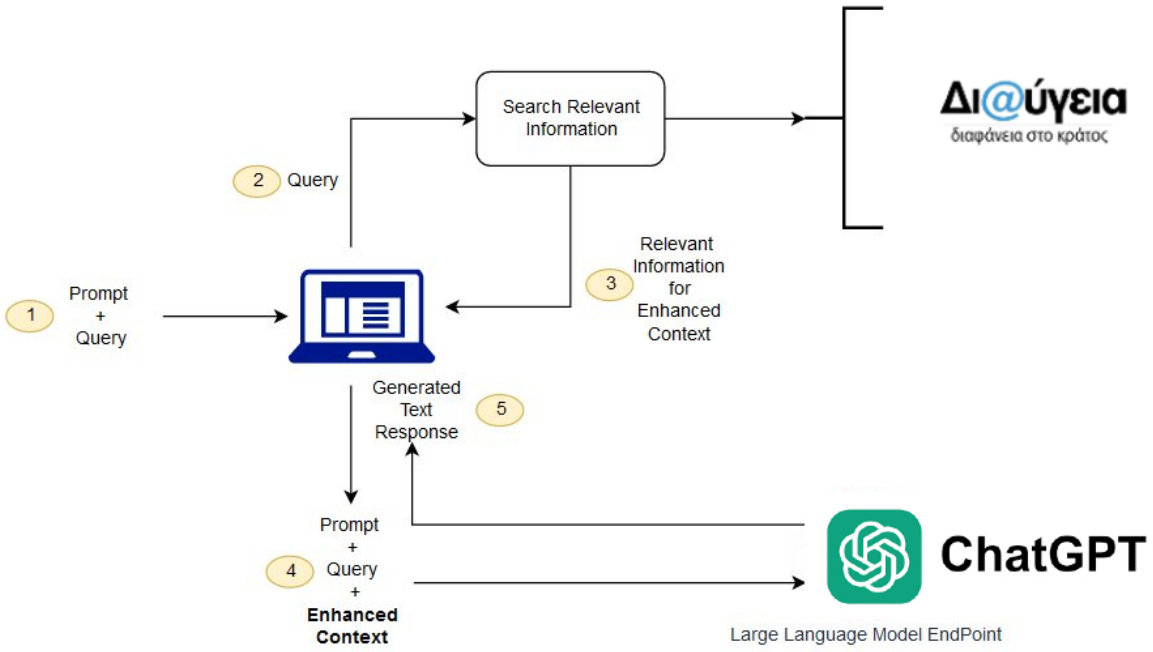

Figure 5.1: This is our RAG approach

5.1.2 RAG Architecture

The architecture of the RAG model consists of two primary components: the retriever and the generator. These components work together to ensure the chatbot provides accurate and relevant responses.

Retriever Component

The retriever component identifies relevant information from the corpus. We utilize Dense Passage Retrieval (DPR) as our retrieval method, which involves the following steps:

- **Document Indexing**: Diavgeia open data API documents are preprocessed and indexed using dense vector representations. This allows for efficient similarity searches.
- **Query Encoding**: When a user submits a query, it is encoded into a dense vector representation.
- **Similarity Search**: The encoded query vector is compared against the indexed document vectors to retrieve the most relevant passages.

The use of dense vector representations enables the retriever component to perform more accurate and contextually aware searches compared to traditional keyword-based methods.

Generator Component

The generator component uses a fine-tuned version of a Large Language Model (LLM), such as GPT-3.5, to generate responses based on the retrieved information. The process involves:

- **Input Preparation**: The retrieved passages and the user's query are combined to form the input for the generative model.
- **Response Generation**: The LLM processes the input and generates a coherent and contextually relevant response.
- **Fine-Tuning**: The LLM is fine-tuned on domain-specific data from the Diavgeia API to improve its understanding and generation capabilities in the context of government-related queries.

5.1.3 Large Language Models (LLM)

Models such as GPT-3.5 is an advanced version of the Generative Pre-trained Transformer (GPT) series developed by OpenAI. It is built upon the transformer architecture, which utilizes self-attention mechanisms to process and generate text. The key characteristics of GPT-3.5 include:

- **Model Size**: GPT-3.5 has billions of parameters, making it one of the largest and most powerful language models available. This vast number of parameters allows it to capture a wide range of language patterns and nuances.
- **Training Data**: GPT-3.5 is trained on a diverse dataset comprising a significant portion of the internet, including books, articles, and websites. This extensive training enables it to generate text that is coherent and contextually relevant across various domains.
- **Few-Shot Learning**: One of the notable features of GPT-3.5 is its ability to perform few-shot learning. This means that it can understand and respond to new tasks with minimal examples, making it highly adaptable and versatile for different applications.

- **Fine-Tuning**: For our specific use case, GPT-3.5 is fine-tuned on domain-specific data from the Diavgeia API. Fine-tuning involves additional training on a smaller, specialized dataset, allowing the model to better understand and respond to queries related to Greek government administrative acts and decisions.

The transformer architecture of GPT-3.5 comprises multiple layers of self-attention and feedforward neural networks. The self-attention mechanism enables the model to weigh the importance of different words in a sentence, allowing it to generate more coherent and contextually appropriate responses. The extensive training and fine-tuning process ensure that GPT-3.5 can produce high-quality, human-like text, making it an ideal choice for our chatbot's generator component.

5.1.4 RAG Advantages

RAG offers several advantages for our chatbot system:

- **Enhanced Accuracy**: By retrieving relevant information before generating responses, RAG ensures that the answers are well-informed and contextually appropriate.
- **Scalability**: The modular architecture of the RAG model allows for easy scaling and integration with large knowledge bases, such as the Diavgeia open data API.
- **Flexibility**: The combination of retrieval and generation provides the flexibility to handle a wide range of queries, from simple factual questions to complex, multi-turn interactions.
- **Improved User Experience**: The ability to generate natural and engaging responses enhances the overall user experience, making interactions with the chatbot more satisfying and effective.

5.1.5 Integration and Workflow

The integration of the retriever and generator components within the RAG architect follows a specific workflow:

1. **Query Processing**: The user's query is processed and encoded by the retriever component.

2. **Information Retrieval**: Relevant passages are retrieved from the indexed corpus based on the encoded query.

3. **Response Generation**: The retrieved passages, along with the user's query, are input into the generator component, which produces the final response.

4. **Response Coordination**: The response is coordinated and refined to ensure coherence and relevance before being sent back to the user interface.

This workflow ensures that the chatbot can quickly and accurately respond to user queries by leveraging both retrieval and generative capabilities.

5.1.6 Search Concepts in Elasticsearch

Elasticsearch is a powerful tool for search and analytics, designed to handle large volumes of data in real-time. Key concepts include:

- **Index**: A collection of documents that share similar characteristics.
- **Document**: The basic unit of information that can be indexed, represented in JSON format.
- **Query**: A search request to find documents that match specific criteria.
- **Match Query**: A query that analyzes the input text and constructs a query that finds matching documents.

5.2 The `search` Function in Elasticsearch Python Client

The `search` function in the Elasticsearch Python client is used to query the Elasticsearch cluster and retrieve search results based on the specified criteria. This function is highly versatile, allowing for a wide range of search operations, from simple queries to complex aggregations and filtering.

5.2.1 Importing the Client

To use the `search` function, you first need to import and configure the Elasticsearch client:

```
from elasticsearch import Elasticsearch

# Create an instance of the Elasticsearch client
es = Elasticsearch(
    hosts=[{"host": "localhost", "port": 9200}],
    http_auth=("user", "password"),
    scheme="https",
    verify_certs=False
)
```

5.2.2 Function Signature

```
Elasticsearch.search(index=None, body=None, params=None, headers=None)
```

5.2.3 Parameters

- **`index`** (`str` or `list` of `str`, optional): The name of the index or indices to search. If not specified, the search will be performed across all indices.
- **`body`** (`dict`, optional): The search definition using Query DSL. This parameter contains the query, filters, aggregations, and other search options. If not provided, a `match_all` query is executed by default.
- **`params`** (`dict`, optional): Additional URL parameters to pass to the query. These can include options like `scroll`, `size`, `from`, etc.
- **`headers`** (`dict`, optional): Additional HTTP headers to send with the request.

5.2.4 Return Value

The function returns a dictionary containing the search results. The structure of the response includes metadata about the search, such as the total number of hits, and the actual search results.

5.2.5 Example Usage

Basic Search

```
# Simple match query to search for documents containing "elasticsearch"
    in any field
response = es.search(index="my_index", body={
    "query": {
        "match": {
            "content": "elasticsearch"
        }
    }
})

# Print the search results
for hit in response['hits']['hits']:
    print(hit['_source'])
```

Search with Aggregations

```
    # Search query with aggregation to get the count of documents per
        category
response = es.search(index="my_index", body={
    "query": {
        "match_all": {}
    },
    "aggs": {
        "categories": {
            "terms": {
                "field": "category.keyword"
            }
        }
    }
})

# Print the aggregation results
for bucket in response['aggregations']['categories']['buckets']:
    print(f"Category: {bucket['key']}, Count: {bucket['doc_count']}")
```

Paginated Search

```
# Paginated search with size and from parameters
response = es.search(index="my_index", body={
    "query": {
        "match_all": {}
    }
}, params={"size": 10, "from": 20})

# Print the search results
for hit in response['hits']['hits']:
    print(hit['_source'])
```

Scrolling

For large result sets, you may need to use the scroll API to retrieve results in batches:

```
# Initialize the scroll
response = es.search(index="my_index", body={
    "query": {
        "match_all": {}
    }
}, params={"scroll": "2m", "size": 100})

scroll_id = response['_scroll_id']

# Fetch all results using scroll
while True:
    response = es.scroll(scroll_id=scroll_id, scroll="2m")
    if not response['hits']['hits']:
        break
    for hit in response['hits']['hits']:
        print(hit['_source'])

    scroll_id = response['_scroll_id']
```

5.2.6 Error Handling

The `search` function may raise exceptions in case of errors. Common exceptions include:

- **`elasticsearch.exceptions.ConnectionError`**: Raised when the client cannot connect to the Elasticsearch cluster.
- **`elasticsearch.exceptions.NotFoundError`**: Raised when the specified index does not exist.
- **`elasticsearch.exceptions.RequestError`**: Raised when there is an issue with the search query, such as a syntax error.

The `search` function in the Elasticsearch Python client is a powerful tool for querying and retrieving data from Elasticsearch indices. By understanding its parameters and usage patterns, you can effectively leverage Elasticsearch's capabilities to perform complex searches and aggregations.

5.3 Our ChatBot Approach with RAG

Building on the concepts discussed earlier, we present our approach to developing a chatbot using the Retrieval-Augmented Generation (RAG) architecture. This method leverages the strengths of both retrieval and generation to enhance the chatbot's performance and accuracy.

5.3.1 Elasticsearch Integration

We integrated Elasticsearch into our system to index and search the Diavgeia documents. The integration allows for scalable and efficient search operations. The following code snippet demonstrates the connection setup and search implementation.

Setting Up the Elasticsearch Client

We initialize the Elasticsearch client, ensuring connection stability with retries and timeouts.

```
from elasticsearch import Elasticsearch, ConnectionError

class DiaygeiaBot:
    def __init__(self, name, logger=None):
        if not logger:
            self.logger = logging.getLogger(__name__)
        self.session_manager = SessionManager(logger=self.logger)
        self.index_name = name
        try:
            self.client = Elasticsearch(
                hosts=[{"host": "elasticsearch", "port": 9200}],
                timeout=30,
                max_retries=10,
                retry_on_timeout=True
            )
            if not self.client.ping():
                raise ValueError("Connection to Elasticsearch failed")
        except ConnectionError as e:
            self.logger.error(f"Failed to connect to Elasticsearch: {e}")
            raise
```

Performing Search Queries

We define a method to perform search queries on the indexed data. The get_context method constructs and executes a match query to retrieve relevant documents.

```
NUM_RESULTS = 3

def get_context(self, question: str):
        search_query = {
            "query": {
                "match": {
                    "content": question
                }
            }
        }
        resp = self.client.search(index=self.index_name,
           body=search_query, size=NUM_RESULTS)
```

```
        results = [{"id": hit['_id'], "content":
            hit['_source']['content']} for hit in resp['hits']['hits']]

        return results
```

5.3.2 Integrating OpenAI for Contextual Responses

To enhance the bot's responses, we integrate OpenAI's language model. This model processes the search results from Elasticsearch and generates coherent and contextually relevant responses.

Setting Up OpenAI

We use the OpenAI API to generate responses. The API key is loaded from environment variables.

```
import openai
from dotenv import load_dotenv

openai.api_key = os.getenv("OPENAI_KEY")
```

Generating Responses

In our AI-driven chatbot system, one of the key components is the function that generates responses to user queries by interacting with OpenAI's language models. This function, named `get_bot_response`, plays a critical role in managing conversation flow by:

- Handling user session data
- Gathering relevant context
- Leveraging the language model to produce accurate and contextually appropriate responses

1. Session Management The process kicks off with retrieving the user's session history:

```
user_history_data = self.session_manager.get_user_session(session_id)
```

By preserving the history of past interactions, stored in `user_history_data`, the chatbot maintains continuity in the conversation. This ensures that responses are relevant and coherent, as the model can reference previous exchanges.

2. Context Gathering Next, the function constructs the context:

```
context = "\n\n".join([str(item) for sublist in
    self.get_context(question) for item in sublist.values()])
context = " ".join(context.split()[:500])
```

This context is derived by invoking `self.get_context(question)`, designed to gather additional relevant information. The context is trimmed to the first 500 words to keep within the token limits of the OpenAI API, ensuring the model focuses on the most pertinent details.

3. Logging for Transparency Before making the API call, the function logs the user's history:

```
self.logger.warning(f"History: {history}")
```

Logging is essential for debugging and monitoring. It gives developers insight into the information being passed to the model, especially during development and testing.

4. Interacting with OpenAI's API At the core of the function is the API interaction:

```
response = openai.ChatCompletion.create(
    model="gpt-4o-mini",
    messages=[
        {"role": "system", "content": "You are a helpful assistant."}
    ] + history + [
        {"role": "user", "content": context},
        {"role": "user", "content": question}
    ],
    max_tokens=300,
    n=1,
    stop=None,
    temperature=0.7,)
```

This call to `openai.ChatCompletion.create()` is carefully structured:

- **System Message:** The conversation starts with `{"role": "system", "content": "You are a helpful assistant."}` to set the assistant's role and guide its behavior.
- **History & Context:** Integrates the conversation history and gathered context, so the model is fully aware of past interactions and relevant information.
- **User's Query:** The current question is appended, allowing the model to address it while considering the context.

5. Error Handling To ensure robustness, the API call is wrapped in a `try-except` block:

```
try:
    response = openai.ChatCompletion.create(...)
    # Extracting the completion text from the response
    completion = response.choices[0].message['content'].strip()
except Exception as e:
    print(e)
```

This error-handling mechanism prevents the system from failing silently and aids in diagnosing issues effectively.

6. Session Update Finally, the user's session history is updated:

```
self.session_manager.update_user_session(session_id, user_history_data,
    question, completion)
```

This ensures that the chatbot maintains an up-to-date record of the conversation, allowing it to recall past interactions seamlessly.

By breaking down the function into these steps, it's easier to understand how get_bot_response effectively manages the conversation flow, ensuring a natural and coherent interaction with the user.

5.3.3 Conversation History

Importance of Conversation History

In the realm of conversational AI, maintaining a history of user interactions is crucial for creating meaningful and contextually aware responses. Conversation history allows the

system to track the flow of dialogue, ensuring that responses are coherent and relevant to the ongoing discussion. By storing past interactions, the system can maintain context over time, enabling it to provide more personalized and informed answers. This is particularly important in applications where users may revisit topics or reference previous interactions, as it allows the AI to recognize these references and respond appropriately.

Furthermore, conversation history contributes to enhancing the overall user experience by enabling the AI to build a relationship with the user. For instance, in customer support scenarios, maintaining a record of previous queries and resolutions can help in providing faster and more accurate assistance. In our system, the conversation history plays a pivotal role in feeding relevant context to the language model, which is essential for generating responses that are not only accurate but also contextually aligned with the ongoing conversation.

Leveraging Redis Service

Redis is an open-source, in-memory data structure store that is widely used as a database, cache, and message broker. In our system, Redis plays a critical role in managing conversation history by providing a fast and efficient storage solution for session data. Its in-memory nature allows for quick retrieval and updating of conversation history, which is essential for maintaining the real-time responsiveness of the AI. Redis also supports various data structures, including strings, hashes, and lists, making it well-suited for storing and handling serialized conversation data. By leveraging Redis, our system can efficiently store user interactions, ensuring that the conversation history is readily available for the language model to generate contextually relevant responses.

Implementation of Conversation History in Our System

In our thesis, we implemented the conversation history feature using two primary components: the `SessionManager` class from `manage_session.py` and the `DiaygeiaBot` class from `response.py`. These components work in tandem to manage, retrieve, and utilize the conversation history in generating responses from the language model.

Session Management with `SessionManager`

The `SessionManager` class is responsible for handling user sessions, including storing, retrieving, and updating the conversation history. This class leverages Redis as the under-

lying storage mechanism to efficiently manage session data. The key functionalities provided by `SessionManager` are:

Retrieving User Sessions The `get_user_session` method is used to retrieve the conversation history for a given session ID. It checks if the session data exists in Redis, decompresses it, and loads it into a `Conversation` object. If no session data is found, an empty `Conversation` object is returned.

```
def get_user_session(self, session_id):
    try:
        key = session_id
        if self.engine.get(key):
            result = Conversation(
                pd.read_json(zlib.decompress(self.engine.get(key)).
                decode("utf-8"))
            )
        else:
            result = Conversation(pd.DataFrame(),
                columns=Conversation.column_names)
    except Exception as e:
        self.logger.warning(f"Could not retrieve user session due to
            {e}")
        result = Conversation(pd.DataFrame(),
            columns=Conversation.column_names)
    return result
```

Listing 5.1: Retrieving User Sessions

Resetting User Sessions The `reset_user_session` method allows the system to clear the conversation history for a given session, effectively resetting the session. This is useful in scenarios where a fresh start is required, such as when a new conversation begins.

```
def reset_user_session(self, session_id):
    try:
        key = session_id
        self.engine.delete(key)
    except Exception as e:
        self.logger.warning(f"Could not reset user session due to {e}")
```

Listing 5.2: Resetting User Sessions

Updating User Sessions The `update_user_session` method is central to maintaining the conversation history. It appends the latest user question and the corresponding bot response to the existing conversation history and then stores the updated conversation back into Redis.

```
def update_user_session(
    self, session_id, conversation: Conversation, question: str,
        response: str
):
    try:
        user_conversation = Conversation(
            pd.concat(
                [
                    conversation,
                    pd.DataFrame(
                        [
                            {
                                "user": session_id,
                                "utterance": question,
                                "time":
                                    dt.now().strftime(DATETIME_FORMAT_STR),
                            }
                        ]
                    ),
                    pd.DataFrame(
                        [
                            {
                                "user": "BOT",
                                "utterance": response,
                                "time":
                                    dt.now().strftime(DATETIME_FORMAT_STR),
                            }
                        ]
                    ),
                ],
```

```
                ignore_index=True,
            )
        )
        key = session_id
        json_s =
            zlib.compress(user_conversation.to_json().encode("utf-8"))
        self.engine.set(key, json_s)
    except Exception as e:
        self.logger.warning(f"Could not update user session due to {e}")
```

Listing 5.3: Updating User Sessions

These methods ensure that the conversation history is accurately maintained and easily accessible for each user session.

Utilizing Conversation History in `DiaygeiaBot`

The `DiaygeiaBot` class in `response.py` is where the conversation history is utilized to generate responses from the language model. The key methods within this class that interact with the conversation history include:

`get_bot_response` This method is the core of the bot's response generation. It retrieves the conversation history using the `SessionManager`, processes it to extract relevant context, and then passes this context along with the user's current question to the language model. The history is formatted and included in the prompt to the model, ensuring that the response is informed by the ongoing dialogue.

`get_llm_response` This method calls the OpenAI API to generate a response from the language model. It constructs a message sequence that includes the system prompt, conversation history, and the current user question. This sequence is crucial for maintaining context within the conversation, allowing the model to generate responses that are coherent and contextually relevant.

`get_context` Although primarily focused on retrieving related information from Elasticsearch, this method indirectly interacts with the conversation history by considering the user's previous interactions when forming search queries. This helps in providing contextually relevant documents that can assist the model in generating accurate responses.

5.4 Summary

In this chapter, we provided a comprehensive overview of the architectural design and technical execution of our chatbot system developed for the Greek government platform, Diavgeia. We emphasized the significance of a robust architecture in ensuring the chatbot's functionality, reliability, and performance.

We introduced the Retrieval-Augmented Generation (RAG) approach, which combines the strengths of retrieval-based and generative models to enhance the accuracy and relevance of responses. We detailed the two main stages of the RAG model: the retrieval stage, which utilizes Dense Passage Retrieval (DPR) to fetch relevant documents, and the generation stage, where a fine-tuned Large Language Model (LLM), specifically GPT-4o mini, generates coherent and contextually appropriate responses.

The chapter thoroughly explained the retriever component, focusing on document indexing, query encoding, and similarity search. We also explored the generator component's process, from input preparation and response generation to fine-tuning on domain-specific data from the Diavgeia API.

We discussed the integration of Elasticsearch for efficient search operations and how the chatbot leverages Elasticsearch's capabilities through its Python client.

Furthermore, we demonstrated the integration of OpenAI's language model to produce contextually enriched responses according to past and present in a dialogue. Code snippets were provided to illustrate the setup and operation of the Elasticsearch client, the search query execution, and the response generation using OpenAI's API.

As for the conversation history in our system it serves as a foundational element for generating meaningful and contextually aware responses. By carefully managing and utilizing this history, our AI can maintain continuity in conversations, deliver personalized interactions, and ultimately enhance the user experience. The integration of `SessionManager` for session handling and `DiaygeiaBot` for response generation demonstrates a well-coordinated approach to leveraging conversation history in a practical AI application.

Chapter 6

System Architecture and Implementation

This chapter will explore each component of the system in detail, starting with an overview of the architecture and the specific roles of backend services, frontend development, and Docker containerization. We will then provide insights into the implementation of the Flask application, the configuration of Docker services, and the design of the user interface. By examining the system architecture and its implementation comprehensively, this chapter aims to offer a clear understanding of how each part contributes to the overall functionality and performance of the chatbot system.

6.1 Overview of System Architecture

The system architecture for our chatbot encompasses a range of components that work together to deliver a seamless and efficient user experience. This architecture includes backend services that handle data processing and retrieval, frontend development for the user interface, and Docker services for containerization and deployment. The following sections provide detailed descriptions of these components and their roles in the overall system.

The architecture is designed to be modular, scalable, and efficient, ensuring that each component can be developed, tested, and deployed independently while maintaining smooth integration with other parts of the system. This modularity facilitates easier maintenance and updates, allowing us to introduce new features and improvements without disrupting the entire system.

The key components of our system architecture include:

- **Backend Services**: Responsible for handling data retrieval, processing, and interac-

tions with the Diavgeia open data API. This includes the implementation of the Retrieval-Augmented Generation (RAG) model, which combines retrieval-based and generative approaches to provide accurate and contextually relevant responses.

- **Frontend Development**: Encompasses the user interface of the chatbot, designed using HTML and CSS. The frontend ensures that users can interact with the chatbot seamlessly, receiving timely and accurate responses to their queries.

- **Docker Services**: Used for containerizing the various components of the system, ensuring consistent and reproducible deployment across different environments.

6.2 Backend Services

The backend services form the core of our chatbot system, handling data retrieval, processing, and interactions with various APIs. This section focuses on the Flask application, which serves as the backend, and its Docker configuration.

6.2.1 Flask Application

The Flask application acts as the backend for our chatbot system. It is responsible for processing user queries, interacting with the Elasticsearch document store, and utilizing the Retrieval-Augmented Generation (RAG) model to generate responses. The Flask application is containerized using Docker to ensure consistency and ease of deployment across different environments.

6.2.2 Dockerfile for Flask Application

The Dockerfile defines the steps to create a Docker image for the Flask application. Below is the detailed configuration:

Base Image

The Dockerfile begins by specifying the base image:

```
#Use an official Python runtime as a parent image
FROM python:3.10-slim-buster
```

Listing 6.1: Base Image

We start with the official python:3.10-slim-buster image, a lightweight version of Python 3.10, which serves as the parent image for our Flask application.

Package Installation

Next, we update the package lists and install necessary system packages:

```
#Update and install necessary packages
RUN apt-get update &&
apt-get install -y build-essential git &&
apt-get install -y libglib2.0-0 libsm6 libxext6 libxrender-dev &&
apt-get install -y libgl1-mesa-glx &&
apt-get install -y libopencv-dev &&
apt-get install -y libboost-dev &&
apt-get install -y libopenblas-dev &&
apt-get clean
```

Listing 6.2: Package Installation

This step installs essential packages, such as build tools, Git, OpenCV, Boost, and others required for the application. The apt-get clean command is used to remove unnecessary files, reducing the image size.

GCC and Rust Installation

We then install GCC and set up Rust for later use in the application:

```
#Install GCC
RUN apt-get -y install gcc
```

Listing 6.3: GCC Installation

```
#Install curl and Rust
RUN apt-get update &&
apt-get install -y curl &&
curl https://sh.rustup.rs -sSf | sh -s -- -y

#Set the PATH for Rust
```

```
ENV PATH=/root/.cargo/bin:$PATH
```

Listing 6.4: Rust Installation

The installation of GCC is required for compiling any C extensions that the application or its dependencies might use. Rust is installed via curl, and its binary path is added to the PATH environment variable.

Application Directories and Environment Variables

We then create directories for the application and NLTK data, and set environment variables:

```
#Create application directories
RUN mkdir -p /app RUN mkdir -p /app/nltk_data
```

Listing 6.5: Application Directories

```
#Set environment variable for NLTK data
ENV NLTK_DATA=/app/nltk_data

#Set the environment variable for Flask
ENV FLASK_APP=run.py
```

Listing 6.6: Environment Variables

These commands create the necessary directories within the container and set the NLTK_DATA and FLASK_APP environment variables, which will be used by the application at runtime.

Dependency Installation

We then upgrade pip, set the working directory, and install the application dependencies:

```
#Upgrade pip
RUN pip install --upgrade pip

#Set the working directory to /app
WORKDIR /app
```

Listing 6.7: Pip Upgrade and Working Directory

```
#Copy the requirements file to the container
COPY requirements.txt .

#Install the dependencies
RUN --mount=type=cache,target=/root/.cache pip install -r
    requirements.txt
```

Listing 6.8: Install Dependencies

The working directory is set to /app, and pip is upgraded to the latest version. The requirements.txt file is copied into the container, and the dependencies are installed using the –mount=type=cache option to cache pip downloads, which helps to speed up subsequent builds.

Application Code and Port Exposure

Finally, the application code is copied into the container, and the port is exposed:

```
#Copy the current directory contents into the container at /app
COPY . /app

#Expose port 8000 for the Flask app
EXPOSE 8000
```

Listing 6.9: Application Code and Port Exposure

All application code is copied to the /app directory, and port 8000 is exposed, which is the port that the Flask application will use to listen for incoming requests.

6.2.3 Docker Compose for Multi-Container Setup

To orchestrate the multi-container application, we use Docker Compose. Below is the `docker-compose.yml` file:

Redis Service

The Redis service in our system utilizes the Redis database, a highly performant in-memory data structure store that is widely used for caching, session management, and real-time analytics. Redis excels at handling key-value data with extremely low latency, making it

an ideal choice for applications requiring fast data access and high throughput. In our setup, Redis facilitates efficient caching and data storage, allowing for rapid retrieval of frequently accessed information and ensuring that the system can scale effectively under heavy load.

```
version: '3.5'

services:

  redis:
    container_name: redis
    image: redis:latest
    ports:
      - "6379:6379"
    volumes:
      - redis_data:/data
```

Listing 6.10: RedisGraph Service Configuration

Elasticsearch Service

Elasticsearch is a powerful search and analytics engine based on Apache Lucene. It is designed to handle large volumes of data and provides near real-time search and analytics capabilities. In our system, the Elasticsearch service is configured as a single-node cluster, which simplifies deployment and management. By disabling security (xpack.security.enabled=false) and exposing the service on port 9200, we facilitate ease of access for other components. Persistent storage is ensured through data and log volumes, preserving data across container restarts and facilitating debugging and performance monitoring.

```
  elasticsearch:
    container_name: document_store_bm25-container
    image: docker.elastic.co/elasticsearch/elasticsearch:7.11.0
    environment:
      - xpack.security.enabled=false
      - "discovery.type=single-node"
    ports:
      - "9200:9200"
    volumes:
      - elasticsearch-data:/usr/share/elasticsearch/data
      - elasticsearch-logs:/usr/share/elasticsearch/logs
```

Listing 6.11: Elasticsearch Service Configuration

Kibana Service

Kibana is a visualization tool designed to work with Elasticsearch. It provides a user-friendly interface for querying, analyzing, and visualizing data stored in Elasticsearch. In our setup, the Kibana service is linked to the Elasticsearch service through the ELASTICSEARCH_HOSTS environment variable, ensuring seamless integration. Kibana depends on Elasticsearch to be running first, which is managed through the depends_on directive. By exposing Kibana on port 5601, users can access the Kibana interface to create visualizations, dashboards, and perform ad-hoc queries, enhancing the usability of our data.

```
kibana:
  container_name: kb-container
  image: docker.elastic.co/kibana/kibana:7.11.0
  environment:
    - ELASTICSEARCH_HOSTS=http://document_store_bm25-container:9200
  depends_on:
    - elasticsearch
  ports:
    - "5601:5601"
```

Listing 6.12: Kibana Service Configuration

We used Kibana to make sure all of our documents were correctly indexed as shown in the figure below: 6.1

Flask Application Service

The Flask application serves as the backend of our chatbot system, handling user queries, processing data, and generating responses. The service is built using a Dockerfile located in the current directory, ensuring that all dependencies and configurations are properly set up. The Flask environment is set to development for easier debugging and troubleshooting during development. The service relies on Elasticsearch for data storage and retrieval, which is indicated by the ELASTICSEARCH_HOSTS environment variable. By exposing the Flask

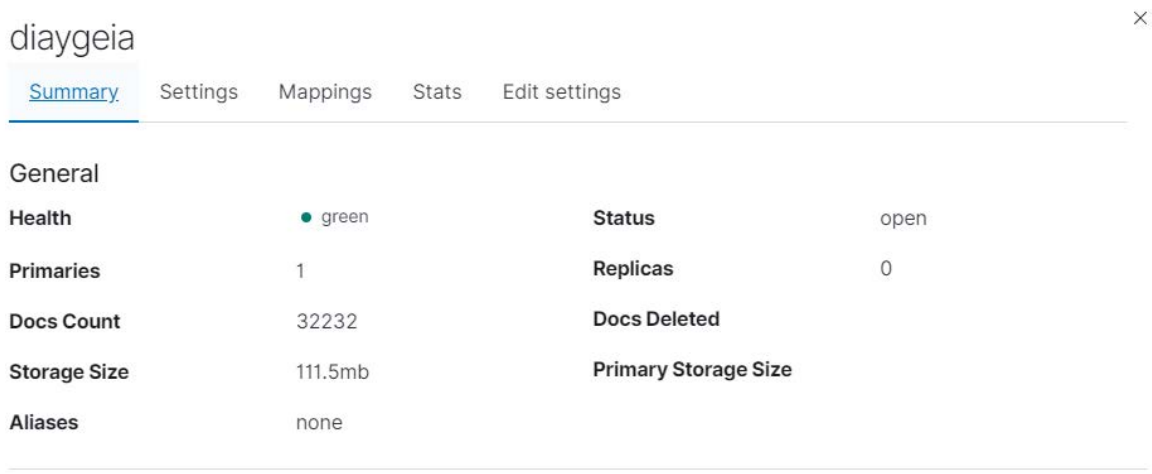

Figure 6.1: Indexer through Kibana service

application on port 8000 and using the appropriate command to run it, we ensure that the application is accessible and ready to handle requests.

```
flask_app:
  container_name: service
  build:
    context: .
    dockerfile: ./Dockerfile
  environment:
    - FLASK_ENV=development
    - ELASTICSEARCH_HOSTS=http://document_store_bm25-container:9200
  depends_on:
    - elasticsearch
  ports:
    - "8000:8000"
  command: flask run --host=0.0.0.0 --port=8000
```

Listing 6.13: Flask Application Service Configuration

Networks and Volumes

To facilitate efficient communication between services and ensure data persistence, we define custom networks and volumes in our Docker Compose configuration. The custom network named `external-product` provides isolated and organized networking for our services, enhancing security and manageability. Persistent volumes are configured for Elasticsearch data, logs, and Kibana data, ensuring that crucial information is retained across

container restarts. Additionally, a volume for the database is included to support any future database services that may be integrated into the system.

```
networks:
  default:
    name: external-product

volumes:
  elasticsearch-data:
    driver: local
  db:
    driver: local

  elasticsearch-logs:
    driver: local
  kibana-data:
    driver: local

  redis_data:
    driver: local
```

Listing 6.14: Networks and Volumes Configuration

This setup ensures that all necessary backend services are containerized and can be deployed consistently across different environments. Docker Compose simplifies the and management of these services, providing a scalable infrastructure for our chatbot system.

6.3 Flask Application Logic

The `run.py` file sets up the Flask application, configures routes, and handles requests for the chatbot system. The Flask application is designed to interact with the DiaygeiaBot, a custom bot for interfacing with the Diaygeia Greek government data.

6.3.1 run.py

```
import ...

app = Flask(__name__)
```

```
CORS(app)

@app.route('/')
def index():
    return render_template('index.html')

@app.route("/diaygeiachat", methods=["POST"])
def diaygeiachat():
    user_input = request.json
    user_text = user_input.get("param1")
    session_id = "1"
    client_name = "Username"
    bot_instance = DiaygeiaBot(name="diaygeia")
    if user_text == "reset":
        bot_instance.session_manager.reset_user_session(session_id)
        return {"response": "Let's start over!"}
    response = {
        "response": bot_instance.get_bot_response(
            user_text,
            session_id,
        )
    }
    return json.dumps(response)

if __name__ == "__main__":
    app.run(port=8000, debug=True)
```

Listing 6.15: Flask Application Logic in run.py

Explanation of run.py

- **Imports**: The necessary modules and classes are imported, including Flask for creating the web application, CORS for handling cross-origin requests, and custom classes for interacting with the Diaygeia data and bot responses.
- **App Initialization**: A Flask application instance is created, and CORS is enabled to allow cross-origin requests.
- **Index Route**: The `index` route renders the main HTML page of the application.

- **Chat Route**: The `/diaygeiachat` route handles POST requests containing user input. The input is processed by an instance of `DiaygeiaBot`, which generates a response based on the input text and session information.
- **Main Block**: The application runs on port 8000 in debug mode when executed directly.

The functionality of this script is done according to class DiaygeiaBot and get_bot_response we presented in previous chapters

6.4 Frontend Development

The HTML file for the DiaygeiaBot chat interface serves as the foundational structure for the web page, organizing the visual and functional elements necessary for user interaction. It includes a header to identify the application, a chat log area where messages between the user and the bot are displayed, and an input form for the user to type and send messages.

6.4.1 HTML Code

The HTML links to a CSS file for styling and a JavaScript file to handle the dynamic functionalities. This separation of structure, style, and behavior allows for a clean, maintainable, and efficient web application design.

index.html

The HTML file serves as the structure for the web page. It defines the layout and elements that will be displayed to the user. The key components include a container for the chat interface, a header, a log area for displaying chat messages, and an input form for user interaction. Here's the code:

```
<!DOCTYPE html>
<html>
<head>
    <title>DiaygeiaBot</title>
    <link rel="stylesheet" type="text/css" href="styles.css">
</head>
<body>
<div class="chat-container">
```

```
    <div class="chat-header">
        <h2>DiaygeiaBot</h2>
    </div>
    <div class="chat-log" id="chat-log">
        <!-- chat messages will be displayed here -->
    </div>
    <form onsubmit="sendData(event)">
        <div class="chat-input">
            <input type="text" id="user-input" placeholder="Type your
                message here...">
            <button type="submit" id="send-button">Send</button>
        </div>
    </form>
</div>
<script src="script.js"></script>
</body>
</html>
```

Listing 6.16: index.html

6.4.2 CSS Code

styles.css

The CSS file is responsible for styling the HTML elements. It defines the appearance of the chat container, header, log area, and input fields. This includes setting dimensions, colors, padding, margins, and other visual properties. Here is the styling code:

```
.chat-container {
    max-width: 600px;
    margin: 0 auto;
    border: 1px solid #e0e0e0;
    border-radius: 6px;
    overflow: hidden;
    font-family: Arial, sans-serif;
}
.chat-header {
    background-color: #5a60ff;
    color: white;
```

```
    padding: 10px;
    font-size: 18px;
    font-weight: bold;
}
.chat-log {
    height: 400px;
    overflow-y: scroll;
    padding: 10px;
    background-color: #f7f7f7;
}
.chat-input {
    padding: 10px;
    display: flex;
    align-items: center;
    background-color: #f7f7f7;
}
#user-input {
    flex: 1;
    padding: 8px;
    border-radius: 6px;
    border: none;
    font-size: 16px;
    background-color: white;
    box-shadow: 0 2px 4px rgba(0, 0, 0, 0.1);
}
#send-button {
    padding: 8px 15px;
    border-radius: 6px;
    border: none;
    background-color: #5a60ff;
    color: white;
    font-size: 16px;
    margin-left: 10px;
    cursor: pointer;
    box-shadow: 0 2px 4px rgba(0, 0, 0, 0.1);
}
```

Listing 6.17: styles.css

6.4.3 JavaScript Code

script.js

The JavaScript file handles the functionality of the chat application. It includes an asynchronous function to send user input to the server and display the response in the chat log. The code captures the form submission event, prevents the default action, retrieves the user input, sends it to the server using the Fetch API, and updates the chat log with the server's response.

```
async function sendData(event) {
    event.preventDefault(); // Prevent the form from submitting the
        traditional way

    const param1 = document.getElementById('user-input').value;

    const response = await fetch('/diaygeiachat', {
        method: 'POST',
        headers: {
            'Content-Type': 'application/json'
        },
        body: JSON.stringify({ param1 })
    });

    const result = await response.json();
    document.getElementById('chat-log').textContent =
        JSON.stringify(result, null, 2);
}
```

Listing 6.18: script.js

6.4.4 Overall interface

After combining these 3 files we get the outcome as shown in the figure below:6.2

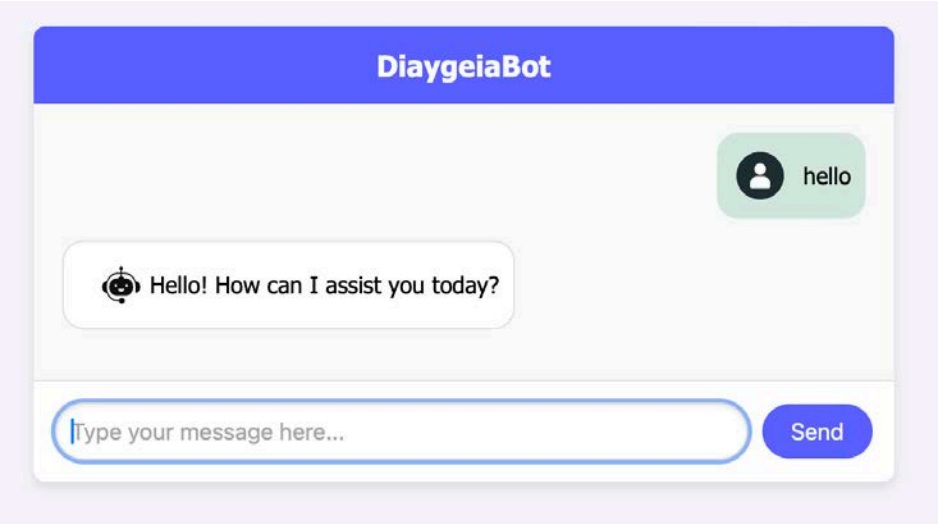

Figure 6.2: Chatbot's interface

6.5 Summary

In this chapter, we began by discussing the overall architecture of the system, highlighting the integration of various components such as the RedisGraph, Elasticsearch, Kibana, and the Flask application, all orchestrated using Docker.

We described the Docker Compose setup that facilitates the management of these services. Each service's configuration was meticulously explained, showcasing the roles and dependencies that define the system's operation.

The core of the backend implementation was laid out through the Flask application logic, emphasizing how the `run.py` script sets up the web application, handles incoming requests, and interacts with the DiaygeiaBot to generate responses based on user input. This section provided a comprehensive understanding of the server-side operations and the flow of data.

The frontend development section covered the creation of the user interface using HTML and CSS. We detailed how the chat interface allows users to interact with the chatbot, with JavaScript handling the asynchronous communication between the client and the server.

Through this chapter, we provided a thorough walkthrough of the deployment and execution environment of the chatbot system.

Chapter 7

Evaluation, Testing, and Results

In this chapter, we will discuss the methodology and process we used to evaluate the performance of our DiaygeiaBot, a chatbot designed to process and retrieve information from official documents. This evaluation involves testing the chatbot's accuracy in responding to specific queries about the structured information in each document. We will detail the steps taken, the custom queries used, and the overall results.

7.1 Evaluation Methodology

The evaluation process involves several steps:

1. **Document Preparation**: We prepared a dataset consisting of documents stored in text format. Each document contains various structured information, such as the total amount, signatories, issue date, issuing body, and protocol number.

2. **Query Formulation**: For each document, we formulated specific queries to test the chatbot's ability to accurately extract information. These queries were designed to target particular fields in the documents.

3. **Ground Truth Extraction**: For each document, we manually extracted the correct answers (ground truth) for each query. This step ensures that we have a reliable benchmark against which to compare the chatbot's responses.

4. **API Interaction**: We used a Python script to automate the process of sending queries to the chatbot and receiving responses. The script reads each document, formulates the query, sends it to the chatbot's API, and compares the response to the ground truth.

5. **Normalization and Comparison**: To ensure accurate comparison, we normalized both the chatbot's responses and the ground truth answers by removing diacritics and converting text to lowercase. This step helps mitigate issues related to text formatting and casing.

```
        import unicodedata

        def remove_diacritics(text):
            normalized = unicodedata.normalize('NFD', text)
            return ''.join([char for char in normalized if
                unicodedata.category(char) != 'Mn'])

        def normalize(text):
            return remove_diacritics(text).lower()
```

Listing 7.1: Normalization and Preproccess

6. **Metrics Calculation**: We calculate metrics of the chatbot to examine its performance on different situations.

7.2 Evaluation Script for Structured Data

The Python script used for evaluation is detailed below. This script automates the entire process, from reading documents and formulating queries to sending requests to the chatbot's API and calculating accuracy.

```
#αριθμός πρωτοκόλλου
for file_path in file_list:
    file_name = os.path.basename(file_path)
    with open(file_path, 'r') as file:
        content = file.read()
        query = f"ποιος είναιοαριθμόςπρωτοκόλλουστοέγγραφομεΑΔΑ       :
            {file_name}"
        content1 = content.split("\n")[1].strip('Αριθμός
            Πρωτοκόλλου:').strip()
        print(content1)
        #ground_truth_line = json.loads(content1)
        ground_truth_answer = str(content1)
        data = {
```

```
            'param1': query
        }
        # Make the POST request to the Flask API
        response = requests.post(url, json=data)

        # Print the status code and response content
        print(f'Status Code: {response.status_code}')
        print(f'Response Content: {response.json()}')
        answer = response.json()['response']
```

Listing 7.2: Evaluation Script

7.2.1 Custom Queries and Ground Truth

The following custom queries were used to evaluate the chatbot's performance on the data saved as structed from the Diavgeia OpenData API. Each query was designed to extract specific pieces of information from the documents. The only thing needed is changing the query from the python scipt with the following questions:

- **Total Amount**: "ποιο είναι το συνολικό ποσό σε ευρώ που αναφέρεται στο έγγραφο με ΑΔΑ : file_name"
- **Signatories**: "ποιος είναι ο υπογράφοντας στο έγγραφο με ΑΔΑ : file_name"
- **Issue Date**: "ποια είναι η ημερομηνία έκδοσης του εγγράφου με ΑΔΑ : file_name"
- **Issuing Body**: "ποιος είναι ο φορέας στο έγγραφο με ΑΔΑ : file_name"
- **Protocol Number**: "ποιος είναι ο αριθμός πρωτοκόλλου στο έγγραφο με ΑΔΑ : file_name"

For each query, the corresponding ground truth answer was extracted manually from the documents and compared against the chatbot's response.

7.2.2 Results and Analysis

After running the evaluation script, we obtained the following results:

- **Correct Answers**: The number of correct answers given by the chatbot.
- **Accuracy**: The accuracy of the chatbot's responses, calculated as the ratio of correct answers to the total number of queries.

Accuracy Calculation

In this section, we describe the calculation of accuracy for our model. Accuracy is a metric that indicates the proportion of correct predictions out of the total number of predictions made.

The formula for accuracy A is given by:

$$A = \frac{N_{\text{correct}}}{N_{\text{total}}} \tag{7.1}$$

where:

- N_{correct} is the number of correct predictions.
- N_{total} is the total number of predictions.

In our implementation, we keep track of the number of correct predictions using the variable `correct_answers`. The total number of predictions is given by the length of the `list file_list`, which contains the data for which predictions are made.

The following code snippet demonstrates how accuracy is calculated in our python script:

```
    if normalize(ground_truth_answer) in normalize(answer):
        print("The answer is correct!")
        correct_answers += 1

accuracy = correct_answers / len(file_list)
print(f"Accuracy: {accuracy}")
```

Here, the function `normalize` is used to preprocess the answers before comparison, ensuring that variations in formatting do not affect the comparison result.

The accuracy is then computed as the ratio of `correct_answers` to the length of `file_list`, and the result is printed as shown below:

```
# Example output
print(f"Correct Answers: {correct_answers}")
print(f"Accuracy: {accuracy}")
```

Listing 7.3: Example Output

Results

After executing the evaluation script with our test files, we obtained impressive results:

- **Correct Answers**: The chatbot correctly answered all queries, resulting in a perfect score of correct answers.
- **Accuracy**: The accuracy of the chatbot's responses was calculated to be 100, as every answer matched the ground truth perfectly.

These results demonstrate that, for our test files, the chatbot achieved a flawless performance with structured data, validating its effectiveness in extracting specific information from the documents. Several factors contributed to this outstanding performance:

- **Quality of Structured Data**: The structured nature of the data from the Diavgeia Open-Data API likely facilitated accurate extraction and comparison. The consistency in data formatting and clear labeling helped the chatbot accurately identify and extract the required information.
- **Effective Query Formulation**: The custom queries were well-defined and targeted specific pieces of information. This precision in query design ensured that the chatbot's responses were evaluated against relevant and correctly defined ground truth answers.
- **Robustness of Normalization**: The use of the `normalize` function in the accuracy calculation played a crucial role in handling variations in formatting and ensuring that the comparison between chatbot responses and ground truth was fair and accurate.

Overall, these results are encouraging and suggest that the chatbot is well-equipped to handle similar structured data tasks with high accuracy.

7.3 Evaluation for Random Queries

To ensure the efficacy and accuracy of the Diavgeia chatbot, we developed a script that leverages advanced natural language processing (NLP) techniques. This script employs OpenAI's GPT-4o-mini model to generate questions and validate the chatbot's responses. This section explores the purpose, functionality, and advantages of each component of the script.

7.3.1 The Need for Automated Evaluation

Evaluating chatbot performance manually can be a tedious and error-prone process. To streamline this, we harnessed the capabilities of GPT-4o-mini, an advanced NLP model devel-

oped by OpenAI. By automating question generation and comparison, we ensure a consistent and objective assessment of the chatbot's responses.

7.3.2 Code Components and Their Functions

Configuration and Setup

```
import os
import openai
import requests
from dotenv import load_dotenv

load_dotenv()

# Set up your OpenAI API key
openai.api_key = os.getenv("OPENAI_KEY")

# Define the URL of the Flask API endpoint
url = 'http://127.0.0.1:8000/diaygeiachat'
```

Listing 7.4: Configuration and Setup

This initial setup is crucial for connecting with OpenAI's API and the local Flask server hosting the chatbot. By loading environment variables from a '.env' file, the script keeps sensitive information, such as API keys, secure and out of the main codebase. The use of `dotenv` simplifies environment management, ensuring that configuration details are easy to update and maintain.

Generating Questions with GPT-4o-mini

```
def generate_question(content):
    prompt = f"Create a random question and its answer (ground_truth)
        from the following text :\n\n{content}\n\nPlease format your
        response as:\nQuestion: <question>\nAnswer: <answer>. I will use
        these for evaluation in my rag based bot assistant."
    response = openai.ChatCompletion.create(
        model="gpt-4o-mini",
        messages=[{"role": "system", "content": "You are a helpful
            assistant."},
```

```
                    {"role": "user", "content": prompt}],
        max_tokens=300,
        temperature=0.7,
    )
    response_text = response['choices'][0]['message']['content'].strip()
    # Ensure the response follows the specified format
    if "Question:" in response_text and "Answer:" in response_text:
        question, answer = response_text.split('\nAnswer: ')
        question = question.replace('Question: ', '').strip()
        answer = answer.strip()
        return question, answer
    else:
        raise ValueError("Response format is incorrect")
```

Listing 7.5: Generating Questions with GPT-4o-mini

The `generate_question` function is a core component that utilizes GPT-4o-mini to create meaningful and contextually relevant questions and answers. By providing a detailed prompt that includes specific instructions and context, the model generates questions that are tailored to the content of the provided text. This approach ensures that the questions are relevant and useful for evaluating the chatbot.

Why GPT-4o-mini? GPT-4o-mini is chosen for its ability to understand and generate human-like text. It offers a balance between performance and computational efficiency, making it ideal for creating diverse and high-quality questions without excessive computational costs. This capability is crucial for generating evaluation material that accurately reflects the nuances of the input content.

Interacting with the Chatbot

```
def get_chatbot_response(question):
    data = {
            'param1': question
    }
    response = requests.post(url, json=data)
    print(f'Status Code: {response.status_code}')
    answer = response.json()['response']
    return answer
```

The `get_chatbot_response` function sends a generated question to the local Flask API, which simulates the chatbot's response. This interaction is essential for testing how well the chatbot handles various questions and provides responses. By resetting the chatbot's state before each query, we ensure that the chatbot's answers are not influenced by previous interactions, maintaining the integrity of each test.

Why Local API Testing? Testing against a local API allows for immediate feedback and iterative improvements. It also enables developers to refine the chatbot's performance in a controlled environment before deploying it in a production setting. This approach helps in identifying and fixing issues promptly, ensuring a robust and reliable chatbot.

Processing Text Files

```
with open('test_log.txt', 'a') as log_file:
    # Iterate over each .txt file in the folder
    for filename in os.listdir(folder_path):
        if filename.endswith(".txt"):
            file_count += 1
            file_path = os.path.join(folder_path, filename)
            # Read the content of the file
            with open(file_path, 'r') as file:
                content = file.read().strip()
            # Generate a question based on the content
            question, ground_truth = generate_question(content)
            # Get the chatbot's response
            get_chatbot_response("reset")
            response = get_chatbot_response(question)
            log_file.write(f"Ground Truth: {ground_truth} | Chatbot
               Response: {response}\n")
```

Listing 7.6: Processing and Evaluating Text Files

The script concludes by processing each text file in a designated folder, generating questions for each, and evaluating the chatbot's responses. This batch processing approach allows for comprehensive testing across multiple content files, ensuring that the chatbot's performance is robust and consistent.

Why Batch Processing? Batch processing is efficient for evaluating the chatbot on a wide range of inputs. By processing multiple files in one go, we can gather a large dataset

of responses. This method provides valuable insights into the chatbot's strengths and weaknesses across diverse scenarios, enabling targeted improvements and ensuring that the chatbot can handle various types of queries effectively.

This script exemplifies how modern NLP tools like GPT-4o-mini can be leveraged to enhance the evaluation process of conversational agents. By automating question generation, and interaction, we achieve a scalable and efficient method for assessing and refining our Diavgeia chatbot. This approach not only saves time but also ensures a high standard of accuracy in evaluating the chatbot's performance.

7.3.3 Results and Analysis

BertScore

To evaluate the performance of the chatbot responses against the ground truth, we utilized the BERTScore metric. BERTScore is a sophisticated evaluation metric for natural language generation tasks that leverages deep contextualized embeddings, particularly those derived from BERT (Bidirectional Encoder Representations from Transformers). Unlike traditional metrics such as BLEU or ROUGE, which rely on surface-level token matching, BERTScore evaluates the semantic similarity between the generated text and reference text by comparing their embeddings. This makes it particularly effective in capturing nuances in meaning that may not be evident through simple n-gram overlap. BERTScore calculates three primary metrics: Precision, Recall, and F1 Score. These metrics are adapted from their traditional definitions to fit the context of text similarity:

- **Precision:** In the context of BERTScore, Precision measures how much of the content in the chatbot's response is relevant to the ground truth. It is computed by averaging the maximum cosine similarity between each token in the generated response and all tokens in the reference text. High Precision indicates that the chatbot's responses are closely aligned with the reference, focusing on relevant content.

- **Recall:** Recall assesses how well the ground truth is captured by the chatbot's response. It is calculated by averaging the maximum cosine similarity between each token in the ground truth and all tokens in the generated response. High Recall suggests that the

chatbot's responses cover a substantial portion of the relevant information found in the ground truth.

- **F1 Score:** The F1 Score is the harmonic mean of Precision and Recall, providing a balanced measure that considers both the relevance and coverage of the response. A high F1 Score implies that the chatbot's responses are not only relevant but also adequately encompass the information present in the ground truth, making it a crucial metric for overall performance evaluation.

The following Python script was developed to compute Precision, Recall, and F1 Score for the chatbot's responses.

```
import re
import matplotlib.pyplot as plt
from bert_score import score

# Path to the log file
file_path = 'test_log.txt'
# Initialize lists to hold the ground truth and chatbot responses
ground_truths = []
chatbot_responses = []

# Regular expressions to match ground truth and chatbot response lines
ground_truth_pattern = re.compile(r"Ground Truth: (.+?) \|")
chatbot_response_pattern = re.compile(r"Chatbot Response: (.+)")

# Read and parse the file
with open(file_path, 'r', encoding='utf-8') as file:
    lines = file.readlines()
    for line in lines:
        ground_truth_match = ground_truth_pattern.search(line)
        chatbot_response_match = chatbot_response_pattern.search(line)
        if ground_truth_match and chatbot_response_match:
            ground_truths.append(ground_truth_match.group(1).strip())
            chatbot_responses.append(
                chatbot_response_match.group(1).strip())
# Compute BERTScore
P, R, F1 = score(chatbot_responses, ground_truths, lang="el")
```

Listing 7.7: BERTScore Evaluation Script

The script performs the following steps:

1. It reads a log file containing pairs of ground truth and chatbot responses.
2. It uses regular expressions to extract and store the ground truth and chatbot responses.
3. The script computes BERTScore metrics (Precision, Recall, and F1 Score) using the `bert_score` Python library.

These steps provide a comprehensive evaluation of the chatbot's performance, allowing us to understand not just how closely the responses match the ground truth but also the quality of the match in terms of semantic similarity.

Results

Building on these results, we now examine the detailed BERTScore metrics across all sentence pairs in the dataset. For the comparison of a pair of ground truth and chatbot's response we get the following results in figure 7.1

```
Pair 15:
Ground Truth: Η κατάσταση της απόφασης είναι PUBLISHED.
Chatbot Response: Η κατάσταση της απόφασης με ΑΔΑ 6ΑΞΤΟΡΡ0-ΟΝΣ είναι "PUBLISHED".
BERTScore (Precision): 0.7814
BERTScore (Recall): 0.9410
BERTScore (F1): 0.8538

Pair 16:
Ground Truth: Το συνολικό ποσό της απόφασης είναι 26.000,00 Ευρώ.
Chatbot Response: Το συνολικό ποσό της απόφασης με ΑΔΑ 6ΑΗ1ΟΞ3Μ-Δ9Ε είναι 26.000,00 ευρώ.
BERTScore (Precision): 0.8143
BERTScore (Recall): 0.9660
BERTScore (F1): 0.8837
```

Figure 7.1: Output for an evaluation pair.

Summing up the results for all the test files we have we get the following average BertScores as in the figure 7.2

```
Average BERTScore (Precision): 0.7851
Average BERTScore (Recall): 0.8788
Average BERTScore (F1): 0.8267
```

Figure 7.2: Average BertScores calculated by our script.

These metrics, presented in the following figures, offer a deeper understanding of the chatbot's semantic accuracy and consistency.

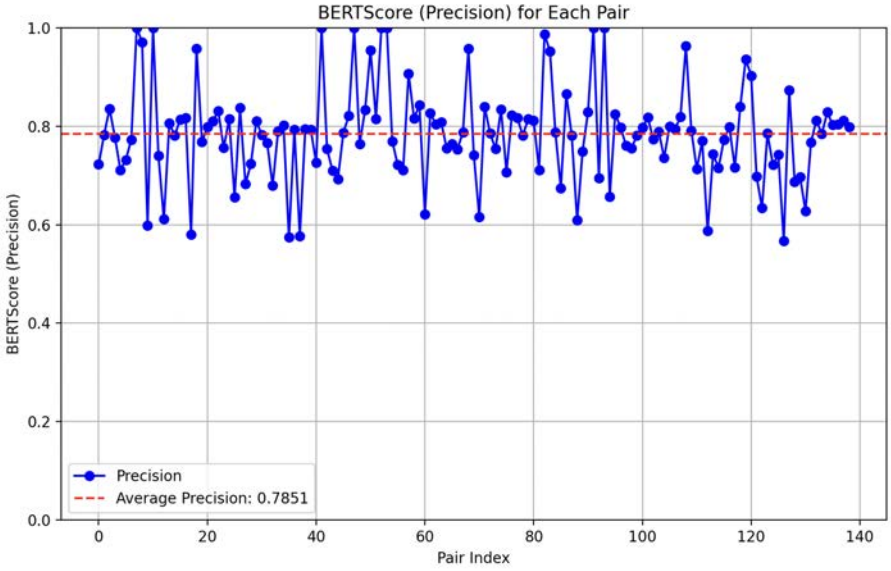

Figure 7.3: Precison Bertscore.

As seen in the figure 7.3 the Precision scores demonstrate more fluctuation compared to the F1 scores, with some pairs scoring as low as 0.6. The lower average Precision compared to F1 indicates that while the model is capable of generating semantically accurate responses, it sometimes includes tokens that do not perfectly match the reference sentence.

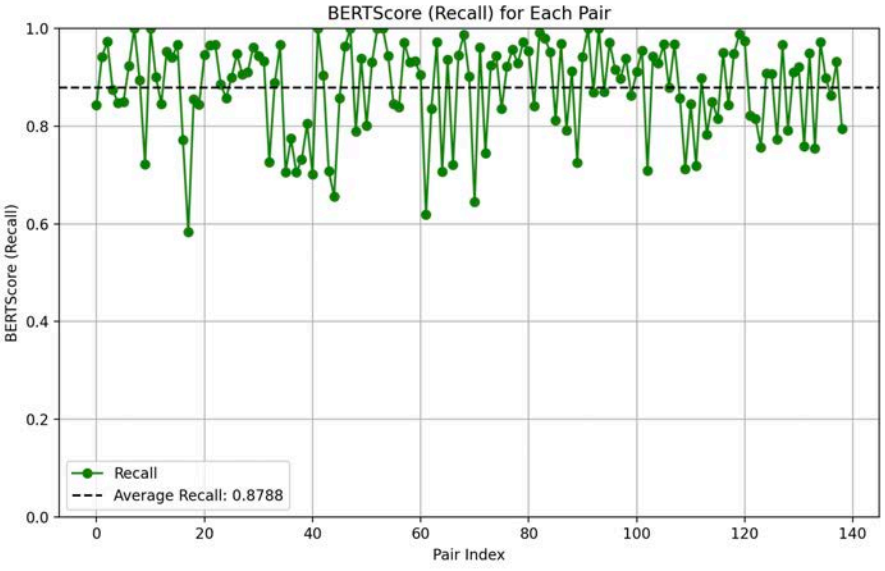

Figure 7.4: Recall BertScore.

In figure 7.4 the Recall scores are generally high, with many scores nearing 1.0. This suggests that the model is effective at capturing the essential tokens from the reference sentences, indicating that the chatbot rarely misses critical information. The higher average Recall compared to Precision suggests that the model prioritizes capturing all relevant content.

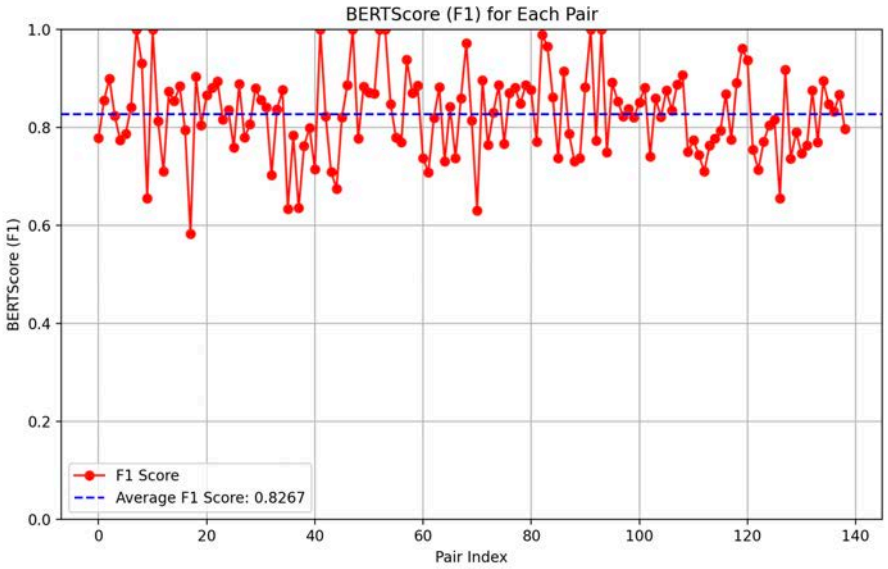

Figure 7.5: F1 BertScore.

In figure 7.5 the BERTScore F1 values fluctuate between 0.6 and 1.0, with the majority of the scores clustering around the average. The high variability suggests that while the chatbot performs well overall, there are specific instances where the semantic similarity between the chatbot's responses and the reference sentences is lower. The consistency of scores around the average implies that the model generally maintains good performance across different sentence pairs.

7.4 History service Evaluation

Evaluating the effectiveness of a chatbot's historical context handling is essential for understanding how well the system maintains coherence and relevance over a series of interactions. Historically, chatbot evaluation has focused on individual queries, often neglecting the importance of contextual continuity within conversations. To address this, we devised a targeted evaluation approach where multiple queries were crafted to relate specifically to the same file or document. This method allows us to examine how well the chatbot leverages

historical context from previous exchanges to maintain a coherent dialogue.

In our evaluation, we generated a series of related queries based on the content of a single file, ensuring that each question was connected to prior interactions. By observing the chatbot's responses to these queries, we assessed its ability to accurately recall and integrate information from earlier in the conversation. This targeted approach enabled us to measure the chatbot's performance in managing historical context, identifying whether it could provide consistent and contextually appropriate answers. Through this process, we gained insights into how well the chatbot maintains the thread of conversation, manages context over time, and responds effectively to complex, multi-turn dialogues. The use of history can be shown in the figure below 7.6

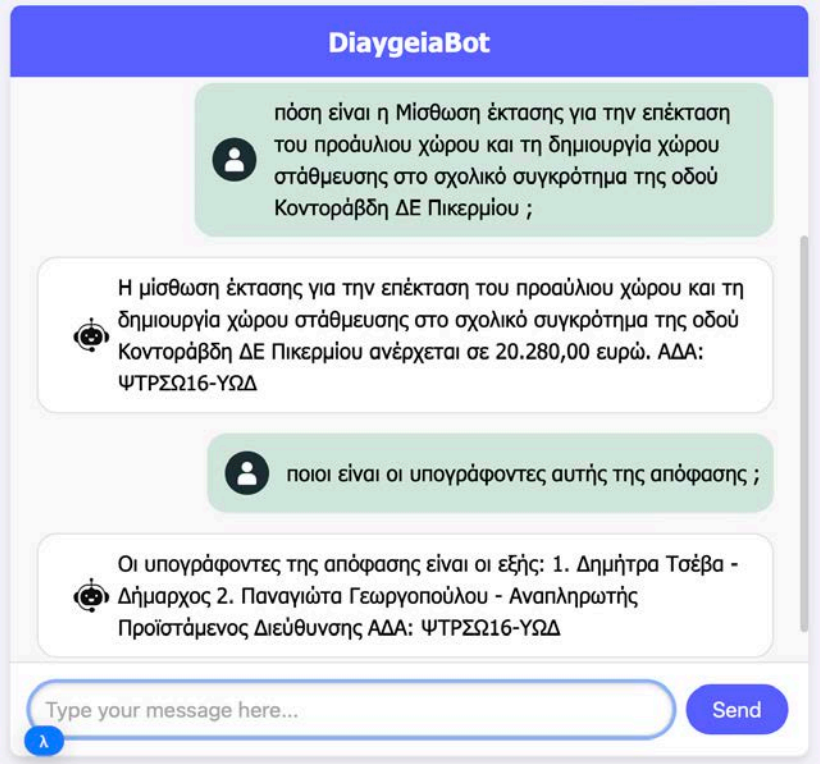

Figure 7.6: History service usage.

7.5 Conclusion

In conclusion, the evaluation of DiaygeiaBot demonstrates its strong performance in extracting structured information from official documents. The bot achieved high accuracy with custom queries, showcasing its ability to handle structured data effectively. The use of normalization, precise query formulation, and rigorous testing through an automated Python

script contributed to these results.

The bot also performed well in handling random queries, with BERTScore metrics indicating that it captures the essential information in most cases, though there is some variability in precision. Additionally, the bot's ability to maintain contextual coherence across multi-turn dialogues was effectively demonstrated through the historical context evaluation.

Overall, DiaygeiaBot proves to be a reliable tool for extracting and processing information from structured documents, with consistent performance across various evaluation scenarios.

Chapter 8

Additional Work

In this chapter, we will explore the implementation of streaming responses in DiaygeiaBot, a chatbot designed to offer real-time conversational capabilities. We will discuss the technical aspects of integrating streaming into the bot, the benefits it brings to user interactions, and the challenges faced during its implementation.

This chapter aims to provide a comprehensive overview of this enhancement, illustrating how streaming responses contribute to a more effective and engaging chatbot experience.

8.1 Implementing Streaming Responses

8.1.1 Introduction to Streaming Responses

In traditional web applications, responses are typically sent back to the client in a single, complete package after the server finishes processing a request. This approach, while straightforward, can lead to increased latency, especially when handling complex operations or generating large responses. Users must wait until the entire response is ready, which can create a less engaging experience, particularly in interactive applications such as chatbots.

Streaming responses offer an alternative by sending parts of the response to the client as they are generated. This technique reduces the perceived latency, as users start receiving information almost immediately. Streaming is particularly beneficial in scenarios where generating the full response takes time, as it keeps users engaged by providing them with incremental updates.

The Concept of Streaming in Web Applications

Streaming in web applications involves breaking down the server's response into smaller chunks that are sent to the client incrementally. Instead of waiting for the entire response to be ready, the server begins transmitting data as soon as it becomes available. This process continues until the full response is delivered.

Streaming is often implemented using protocols such as HTTP/1.1 with chunked transfer encoding, or more modern approaches like HTTP/2 and WebSockets. These protocols allow for continuous data flow between the server and the client, enabling real-time updates.

In the context of chatbots, streaming is used to deliver parts of the bot's response as they are generated by the underlying language model. This approach enhances the responsiveness of the bot, making the interaction feel more natural and less constrained by processing times.

8.1.2 Technical Implementation of Streaming in DiaygeiaBot

The streaming functionality in DiaygeiaBot was implemented by adding a new endpoint that handles streaming responses. This required modifications to both the backend (server-side) and frontend (client-side) components of the system.

Code Explanation

Below is a brief explanation of the key code snippets involved in implementing streaming responses:

- **Flask Endpoint for Streaming:** The endpoint `/diaygeiachat_stream` was created to manage the streaming process. This endpoint initiates the response generation and streams the output to the client in real-time.

```
@app.route("/diaygeiachat_stream", methods=["POST"])
def diaygeiachat_stream():
    ...
    def stream_generator():
        response = client.chat.completions.create(
            model="gpt-4o-mini",
            messages=[
                {"role": "system", "content": "..."},
            ] + history + [
```

```
                {"role": "user", "content": question}
            ],
            stream=True
        )

        for chunk in response:
            if chunk.choices[0].delta.content is not None:
                time.sleep(0.05)
                yield(chunk.choices[0].delta.content)

    return stream_generator(), {"Content-Type": "text/plain"}
```

Listing 8.1: Flask Endpoint for Streaming

This code sets up a streaming response where the server sends parts of the AI's response to the client as soon as they are generated.

- **JavaScript for Handling Streaming:** The client-side JavaScript code was adjusted to handle the streamed content, updating the chat interface with new content as it arrived.

```
const reader = response.body.getReader();
let output = "";

while (true) {
    const { done, value } = await reader.read();
    output = new TextDecoder().decode(value);
    lastElement.innerHTML += output;

    if (done) {
        return;
    }
}
```

Listing 8.2: JavaScript for Handling Streaming

This snippet continuously reads from the response stream, updating the chat log as new data arrives.

8.1.3 Benefits of Streaming Responses

The transition to streaming responses in DiaygeiaBot introduces several benefits, including:

- **Reduced Perceived Latency:** Users receive immediate feedback as soon as parts of the response are ready, significantly reducing the perceived waiting time.
- **Improved User Engagement:** Streaming creates a more interactive and dynamic experience, which can help keep users engaged, especially in lengthy conversations.
- **Efficient Resource Usage:** Streaming allows for better resource management on both the server and client sides, as data is processed and transmitted incrementally rather than in a single large payload.

8.1.4 Challenges and Considerations

While streaming responses offer many advantages, implementing them is not without challenges:

- **Handling Partial Outputs:** The system must ensure that each streamed segment is meaningful and coherent. Users should not receive incomplete thoughts or sentences that could cause confusion.
- **Network Reliability:** Streaming relies on a stable network connection. Interruptions can lead to incomplete responses, requiring mechanisms to handle reconnections or retries.
- **Error Handling:** The system must gracefully handle errors that occur during streaming, ensuring that users are informed of any issues and that the conversation can continue smoothly.

8.2 Conclusion

The implementation of streaming responses in DiaygeiaBot marks a significant improvement in the user experience by making interactions faster and more engaging. This approach aligns with the broader trend in web development toward real-time, responsive applications

that better meet user expectations. As we continue to refine this feature, future work will focus on optimizing performance, enhancing error handling, and further improving the naturalness of the conversation flow.

Bibliography

[1] R OpenAI. Gpt-4 technical report. arxiv 2303.08774. *View in Article*, 2(5), 2023.

[2] Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. Bert: Pre-training of deep bidirectional transformers for language understanding, 2019.

[3] Da Yin, Li Dong, Hao Cheng, Xiaodong Liu, Kai-Wei Chang, Furu Wei, and Jianfeng Gao. A survey of knowledge-intensive nlp with pre-trained language models. *arXiv preprint arXiv:2202.08772*, 2022.

[4] YuHe Ke, Liyuan Jin, Kabilan Elangovan, Hairil Rizal Abdullah, Nan Liu, Alex Tiong Heng Sia, Chai Rick Soh, Joshua Yi Min Tung, Jasmine Chiat Ling Ong, and Daniel Shu Wei Ting. Development and testing of retrieval augmented generation in large language models – a case study report, 2024.

[5] Rama Akkiraju, Anbang Xu, Deepak Bora, Tan Yu, Lu An, Vishal Seth, Aaditya Shukla, Pritam Gundecha, Hridhay Mehta, Ashwin Jha, Prithvi Raj, Abhinav Balasubramanian, Murali Maram, Guru Muthusamy, Shivakesh Reddy Annepally, Sidney Knowles, Min Du, Nick Burnett, Sean Javiya, Ashok Marannan, Mamta Kumari, Surbhi Jha, Ethan Dereszenski, Anupam Chakraborty, Subhash Ranjan, Amina Terfai, Anoop Surya, Tracey Mercer, Vinodh Kumar Thanigachalam, Tamar Bar, Sanjana Krishnan, Samy Kilaru, Jasmine Jaksic, Nave Algarici, Jacob Liberman, Joey Conway, Sonu Nayyar, and Justin Boitano. Facts about building retrieval augmented generation-based chatbots, 2024.

[6] Patrick Lewis, Ethan Perez, Aleksandra Piktus, Fabio Petroni, Vladimir Karpukhin, Naman Goyal, Heinrich Küttler, Mike Lewis, Wen-tau Yih, Tim Rocktäschel, et al. Retrieval-augmented generation for knowledge-intensive nlp tasks. *Advances in Neural Information Processing Systems*, 33:9459–9474, 2020.

[7] Vladimir Karpukhin, Barlas Oğuz, Sewon Min, Patrick Lewis, Ledell Wu, Sergey Edunov, Danqi Chen, and Wen tau Yih. Dense passage retrieval for open-domain question answering, 2020.

[8] Fabio Petroni, Aleksandra Piktus, Angela Fan, Patrick Lewis, Majid Yazdani, Nicola De Cao, James Thorne, Yacine Jernite, Vladimir Karpukhin, Jean Maillard, Vassilis Plachouras, Tim Rocktäschel, and Sebastian Riedel. KILT: a benchmark for knowledge intensive language tasks. In *Proceedings of the 2021 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies*, pages 2523–2544, Online, June 2021. Association for Computational Linguistics.

[9] Kelvin Guu, Kenton Lee, Zora Tung, Panupong Pasupat, and Mingwei Chang. Retrieval augmented language model pre-training. In *International conference on machine learning*, pages 3929–3938. PMLR, 2020.

[10] Wayne Xin Zhao, Kun Zhou, Junyi Li, Tianyi Tang, Xiaolei Wang, Yupeng Hou, Yingqian Min, Beichen Zhang, Junjie Zhang, Zican Dong, Yifan Du, Chen Yang, Yushuo Chen, Zhipeng Chen, Jinhao Jiang, Ruiyang Ren, Yifan Li, Xinyu Tang, Zikang Liu, Peiyu Liu, Jian-Yun Nie, and Ji-Rong Wen. A survey of large language models, 2023.

[11] Mark Chen, Jerry Tworek, Heewoo Jun, Qiming Yuan, Henrique Ponde de Oliveira Pinto, Jared Kaplan, Harri Edwards, Yuri Burda, Nicholas Joseph, Greg Brockman, Alex Ray, Raul Puri, Gretchen Krueger, Michael Petrov, Heidy Khlaaf, Girish Sastry, Pamela Mishkin, Brooke Chan, Scott Gray, Nick Ryder, Mikhail Pavlov, Alethea Power, Lukasz Kaiser, Mohammad Bavarian, Clemens Winter, Philippe Tillet, Felipe Petroski Such, Dave Cummings, Matthias Plappert, Fotios Chantzis, Elizabeth Barnes, Ariel Herbert-Voss, William Hebgen Guss, Alex Nichol, Alex Paino, Nikolas Tezak, Jie Tang, Igor Babuschkin, Suchir Balaji, Shantanu Jain, William Saunders, Christopher Hesse, Andrew N. Carr, Jan Leike, Josh Achiam, Vedant Misra, Evan Morikawa, Alec Radford, Matthew Knight, Miles Brundage, Mira Murati, Katie Mayer, Peter Welinder, Bob McGrew, Dario Amodei, Sam McCandlish, Ilya Sutskever, and Wojciech Zaremba. Evaluating large language models trained on code, 2021.

[12] R. Rouane. Ai-driven tax automation: An in-depth review. *Journal of Tax Practice*, 2024.

[13] J. Song. The implications of providing voice-based chatbots in public service for digital inclusion and public communication. *IntechOpen*, 2023.

[14] J.B. Michael. Understanding conversational artificial intelligence. *IEEE Computer Society*, 2022.

[15] J. Dev and L. Kisselburgh. Privacy and respectful discourse in ai chatbots. In *USENIX*, 2022.

[16] Eleni Tsironidou. E-government services: Usability and accessibility issues. Master's thesis, International Hellenic University, 2021.

[17] Efthimios Tambouris and Kostas Tarabanis. *Emerging Technologies and Electronic Government*. Kallipos, Open Academic Editions, Athens, Greece, 2024. Undergraduate textbook.