15.05.2025

UCY RAG Project

Maria Tsilidou

Mikhail Sumskoi

Marios Menelaou

Implementing a Multi-modular RAG System

ABSTRACT

The following report is made under the MAI623 / DSC514 course and outlines the development of a multi-modular agentic RAG System. Drawing inspiration from numerous existing cases of educational institutions deploying QnA Assistants for public usage, we have set out to implement a similar system for the University of Cyprus. The project is implemented in the LangGraph / LangChain framework, and the report consists of several stages:

- **Introduction / Definitions** - the part where agentic systems are defined;
- **Implementation details** - the way that the system was built;
- **Evaluation** - several evaluation metrics;
- **Conclusion** - the final part that provides an overview of possible gains.

INTRODUCTION

Nowadays, Retrieval-Augmented Generation is considered to be bread and butter of AI Engineering, and many companies find it lucrative to set up on-site RAG pipelines for enhanced interaction with documents [1]. Educational institutions are no exception in this regard, and the existing pool of approaches to RAG is only getting larger - for instance, one of the more recent approaches is GraphRAG [2] combining retrieval with knowledge graphs to achieve better results.

Nevertheless, it would be fair to say that these technologies go at an extremely fast pace, and their real application generally lags behind - with many public institutions neglecting the great many benefits that these technologies could bring them by streamlining their work processes and enhancing their customer experience. Knowing this and also drawing upon personal experience of interaction with the informational flow of the university, our team has explored the possibility of applying the recent advancements in RAG to its processes with the potential to improve student experience and ensure better and more efficient provision of informational corpora in public access. Granted, the public access information that the UCY website provides is quite abundant, and accessing it does not pose a major challenge. However, an obvious question lies in the fact that newcomers need to know of these resources in the first place, e.g. not every Ph.D student may know that the University of Cyprus provides mental care services, or that the gym is located on the ground floor, etc. This informational ‘grey zone’ is what we have set out to resolve using the latest advancements in agentic systems and RAG.

---

1. Retrieval-Augmented Generation with Knowledge Graphs for Customer Service Question Answering - Xu, Wang et al.
2. From Local to Global: A GraphRAG Approach to Query-Focused Summarization - Darren Edge, Ha Trinh et al.

In order to better understand the premise of our work, it is necessary to define what we mean by RAG and Agentic Systems:

- **Retrieval Augmented Generation** - Retrieval-Augmented Generation (RAG) is an approach in natural language processing that combines **retrieval** and **generation** to improve the quality and accuracy of generated responses. Instead of relying only on the knowledge stored in a language model, RAG retrieves relevant information from an external knowledge source (like documents, a database, or a vector store) and then uses that information to generate a response.
- **Agentic Systems** – Agentic systems are AI-driven frameworks designed to operate **autonomously** or semi-autonomously in pursuit of specific goals. These systems can make decisions, take actions, and adapt based on their environment, inputs, and internal reasoning processes. Unlike simple reactive tools, agentic systems often incorporate planning, memory, and feedback mechanisms to handle complex tasks over time and across multiple steps. They can **interact with users,** APIs, or other systems to complete objectives **with minimal human intervention**.

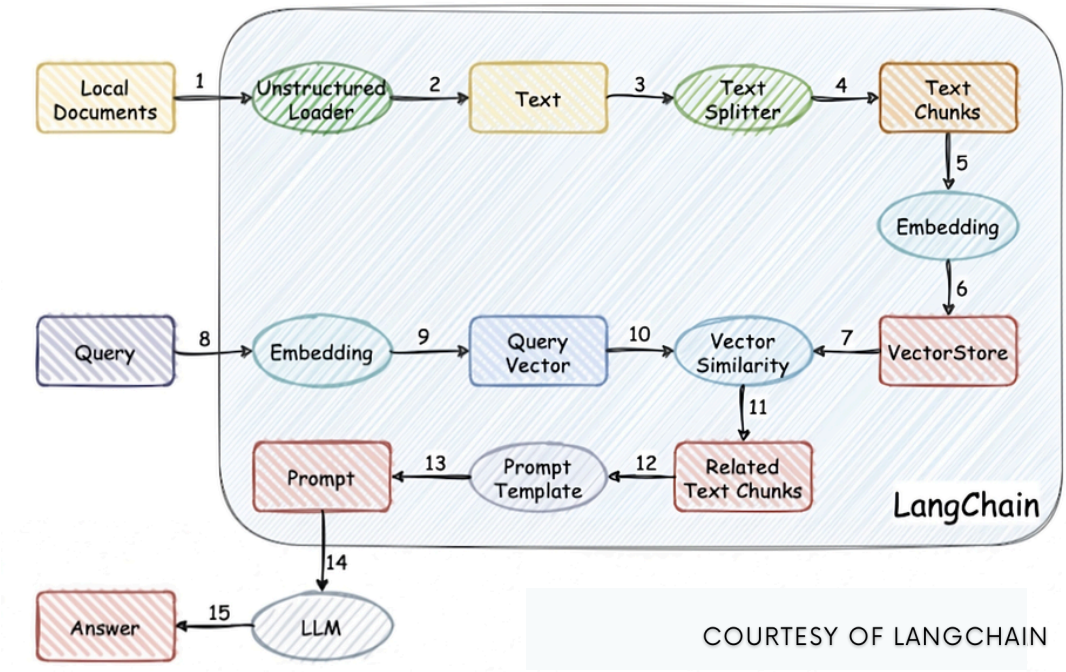
As we can see, these definitions clearly state the main feature of our agent: in lieu of simply answering a user question using the knowledge from pre-training, this approach augments generation with real-world knowledge. The agentic element in our system is grounded in its ability to decide which tools to call. In this particular case, the system involves 3 separate functions / tools:

- **The General Question Node** for general questions
- **The Schedule Node** for files that list all the subjects in a certain department.
- **The Form Node** for links that allow users to get the desired fill-out form.

The details on the implementation of these functions are outlined in the following section.

IMPLEMENTATION DETAILS

Our workflow can roughly be represented using the following map (provided by the LangChain team):

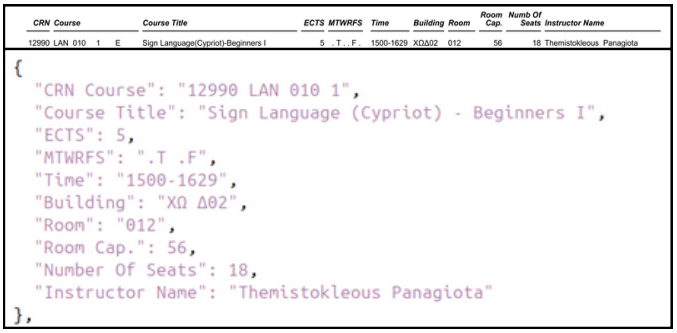


Thus, each of these steps has to be elaborated on. We will begin by describing the way we have extracted our data, and then we will proceed by explaining the chunking / embedding strategy.

- **Data Acquisition & Preprocessing**

The first step of our workflow is gathering the document corpus. Since our assistant must have at least 3 capabilities (schedule, general questions and forms), we need to understand the specifics of all of these corpora (e.g. Schedule files, URLs). There exist multiple approaches to how data can be queried (for example, the schedules could have been represented as CSV files, and then a CSV agent could have been put on top of it, but this approach is not robust enough and frequently fails). This is why it was decided to stick to regular similarity search with document embeddings. Eventually, our data corpus was built in the following way:

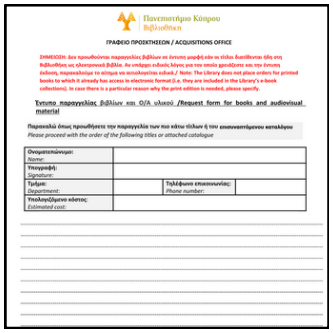
- URL extraction has been done with **ParseHub** and **Spider Web Crawler**. The main benefit of such external services is that they allow us to scrape websites in either markdown or raw HTML content. We have preferred to stick to the latter, since the markdown data tended to miss important artifacts like emails or phones. **ParseHub** allowed our team to scrape the core information manually, while **Spider Web Crawler** was used as a shotgun approach that greedily scraped links but also had a tendency to scrape non-existent or deleted websites. Webscraping was done behind the scenes, and this part is not included in the submitted code.
- Forms extraction was relatively easy, since all the data are gathered in one place [3].
- Schedule extraction has proven the most difficult and unreliable part, because these files were in the form of PDF documents. For these purposes, the SOTA model **Mistral OCR** was used to represent this information as JSON objects. The results / samples of our data can be seen below:



CRN	Course	Course Title	ECTS	MTWRFS	Time	Building	Room	Room Cap.	Numb Of Seats	Instructor Name
12990	LAN 010 1	E Sign Language(Cypriot)-Beginners I	5	.T..F..	1500-1629	ΧΩΔ02	012	56	18	Themistokleous Panagiota


```
{
  "CRN Course": "12990 LAN 010 1",
  "Course Title": "Sign Language (Cypriot) - Beginners I",
  "ECTS": 5,
  "MTWRFS": ".T .F",
  "Time": "1500-1629",
  "Building": "ΧΩ Δ02",
  "Room": "012",
  "Room Cap.": 56,
  "Number Of Seats": 18,
  "Instructor Name": "Themistokleous Panagiota"
},
```


- Schedule data (PDF to JSON)



- Form 1

- Form 2

```
### >>Overview

The Master in Data Science is a highly-selective programme for students who want to begin or advance their careers in Data Science.

The duration of the programme is 1,5-years (90 ECTS), while the language of instruction is English. The programme offers 3 tracks (Computer Science Track / Statistics Track / Business Analytics Track). The first two semesters will be dedicated to core courses; students will select a track at the end of the second semester. Part of the programme is the Capstone project in Data Science, where students tackle specific and practical problems of an interdisciplinary nature. In this course students engage in all aspects of the lifecycle of data-science projects - from process modelling, data extraction, cleaning and validation, to data interpretation and visualization. The capstone project will begin in the summer term, after the end of the second semester.
```

- URL Data Sample

---

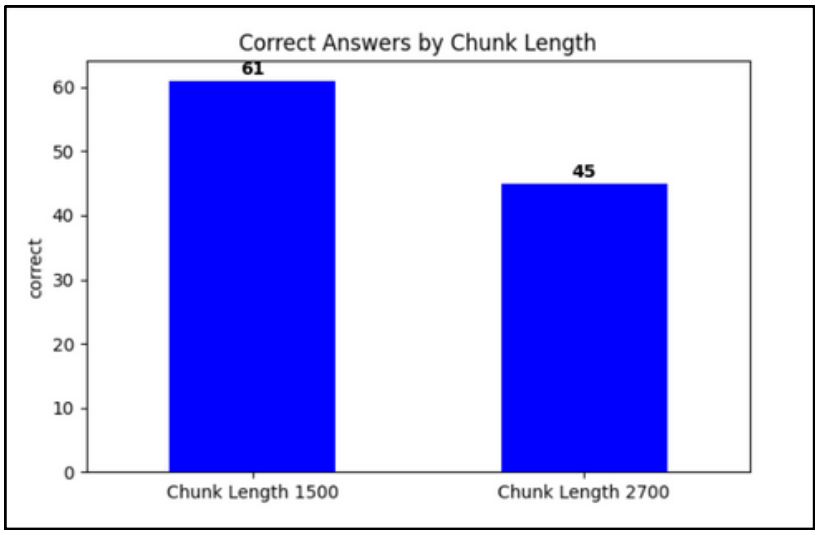
3. https://www.ucy.ac.cy/phy/research/forms-and-information-for-academic-personnel/?lang=en

- **Chunking & Embedding**

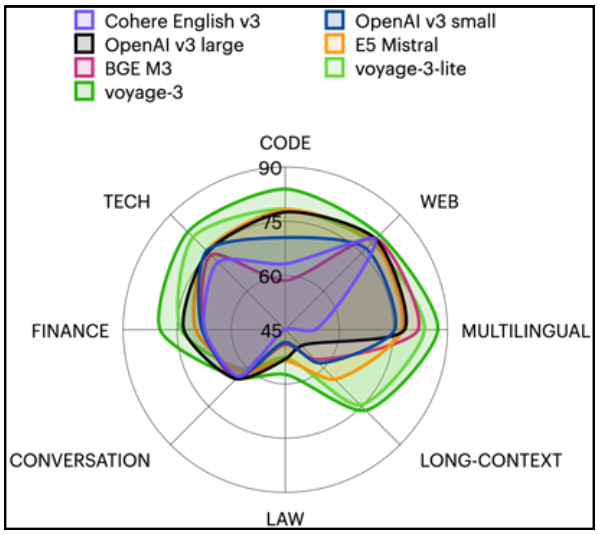
Having gathered our information, we proceed with chunking and embedding. Our strategy looked like the following:

- **Chunking strategy** - create large chunks with strong overlap using LangChain's RecursiveCharacterSplitter
- **Embeddings** - create embeddings using a state-of-the-art embedding model
- **Document Vectorization Strategy** - instead of relying on SQL-search agents, we have decided to create embeddings for all the data - including web-scraped URLs (the university website), JSONs (schedules) and generic PDF / docx files. Our decision is primarily based on the fact that there is a high chance of misspellings, as well as bilingual requests written in both English and Greek.

As stated above, we needed a model that would be able to handle both languages. This is why we relied on an embedding model by VoyageAI called **voyage-3-lite** [4] with 1024 dimensions - this model provided the results on a par with OpenAI embedding models and beat open-source models like **BGE M3** by a large margin. However, apart from just embedding documents, we also had to adjust our chunking strategy that, as we have soon figured out, was really influential for our performance. Using the llm-as-a-judge approach, we have compared several chunk & overlap sizes, and, as a result, smaller chunks were preferable, producing 61 correct answers with the size of 1500 characters, as opposed to 45 correct answers with the size of 2700 characters per chunk. We needed to ensure that general ideas will be kept in our embeddings and not diluted by excessive amounts of information.



- Comparing performance with different chunk params

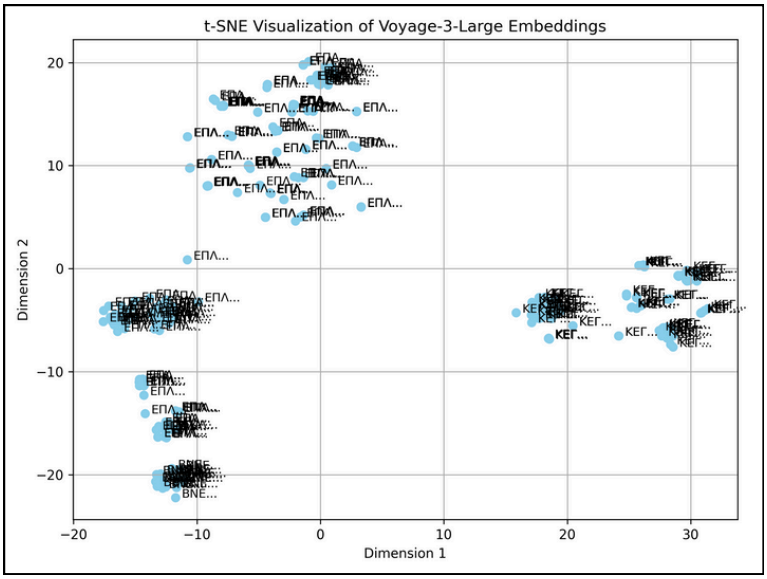


- Voyage-3-lite comparison

Eventually we have acquired around 5647 chunked documents, each of length 1500 characters, for our general corpus. Forms had to be embedded as separate JSON objects - the result is 55 forms, both in Greek and English. Since a lot of schedules are now obsolete and need to be updated every semester, they have been embedded partly, with 1037 entries for each lecture. The resulting embeddings have been upserted to 3 dedicated **Pinecone** [5] indexes set up for **cosine similarity search**.

---

4. https://blog.voyageai.com/2024/09/18/voyage-3/
5. https://docs.pinecone.io/guides/get-started/overview



As a visual confirmation of our approach, we can plot a t-SNE plot of our entries. We can see that entries for specific departments tend to condense together, i.e. BNE (Byzantine Studies) are located close to each other, while ΚΕΓ (Language Centre) are located far away from all the other entries. At the same time, other entries like ΕΠΛ (Computer Science) are rather scattered, but still cluster together, and this low-level visualization does not indicate that entries from different departments are mixed.

- **Generation & Prompting**

The last part of our pipeline is the generation part. At this stage, we set up our agent so that it will be able to get a user query, choose one of the tools tied to their dedicated Pinecone indexes and invoke them with the user question. After retrieving the desired documents, the LLM will provide a response based on this retrieved real-world knowledge. Instead of relying on task-agnostic prompts, we have decided to outline the purpose of each node, so the LLM will know what exactly it is tasked with.

```python
@tool
def fillout_form_fetcher(user_question):
    """This node is needed for fetching URLs for forms to fill out. Usually has to be used in conjunction with the general node or by itself"""
    form_retriever = vectorstore_formfetcher.as_retriever(search_kwargs={"k": 3})

    form_prompt = ChatPromptTemplate.from_template("""This is a University Fill-out Form fetcher. I am going to give you a user question and
                                                   some JSON objects that contain the following:
    1) The name of a document
    2) The URL of the document
    3) The description of a document.
    Your task is to return an appropriate form and its URL based on the user question I'll provide below. Don't double
    the URL and remove any casing so as not to break the link. Don't give me a lot of narration, but do summarize the description
    for these forms. Your answer should be concise and to the point. Do not double the link, e.g. this - Form URL: []
    should be parsed into two separate links. Be polite.
    User question: {question}
    Jsons that were retrieved: {jsons}""")
    chain = {"jsons": form_retriever, "question": RunnablePassthrough()} | form_prompt | llm | StrOutputParser()
    response_acquired = chain.invoke(str(user_question))
    return {"response": str(response_acquired)}

@tool
def schedule_question(user_question):
    """For questions regarding schedules, lesson rooms (where a lesson is conducted - navigating the user to a room is another node's responsibility),
    question = str(user_question)
    retriever_schedule = vectorstore_schedule.as_retriever(search_kwargs={"k": 12})
    template = ChatPromptTemplate.from_template("""This is a node within a student RAG system.
    This particular node is responsible for providing answers regarding schedules for students. These
    shedules are in the form of json objects and mention multiple aspects like: CRN (course number), Course ,Term,
    ,Course, Title, ECTS points for the course, MTWRFS (day of the week, R is thursday), Time, Building, Room, Room Cap(acity),
    number of Seats, Instructor Name. Sometimes some of these aspects are not mentioned. Some building names may accidentally
    be distorted by the OCR system, so please ensure they're written properly: ΘΕ01, ΘΕ02, ΟΕΔ02, ΚΕΠ, ΧΩΔ02, ΧΩΔ01, ΠΤΕΡ, ΗΛΙΑΔΗ
    I need you to provide me an answer based on the context I am going to provide. Do not hallucinate and avoid
    giving wrong information. If you are not sure about the answer, please say so. If there are multiple subjects with some name, return all of them
    The answer should be concise and to the point. After the answer, ask if the user would like to be directed to the building / room
    Context: {context}
    Question: {question}""")
    chain = {"context": retriever_schedule, "question": RunnablePassthrough()} | template | llm | StrOutputParser()
    response_acquired = chain.invoke(str(question))
    return {"response": str(response_acquired)}
```

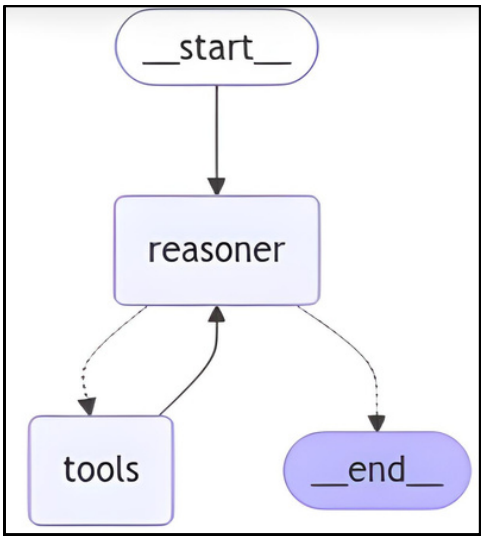
- Tools for the LLM

With this, the system is now equipped with necessary tools and retrievers, which means that we can wrap an agent around these tools for orchestration purposes. The results are outlined below.

- **Agent Architecture**

The architecture of our agent is quite austere but also really effective. The LLM (**reasoner**) gets full information about each of the tools stored in the **tool_node**. The information about these tools is provided in docstrings and the tool names, and that allows the LLM to query these nodes when needed.

```
def chatbot(state: State):
    messages = state["messages"]
    if sys_msg not in messages:
        messages = [sys_msg] + messages
    return {"messages": [llm_with_tools.invoke(messages)]}

graph_builder = StateGraph(State)
tools = [general_question, schedule_question, fillout_form_fetcher]
multiple_tools = ToolNode([general_question, schedule_question, fillout_form_fetcher])
llm_with_tools = llm.bind_tools(tools)

graph_builder.add_node("chatbot", chatbot)

graph_builder.add_node("tools", multiple_tools)

graph_builder.add_conditional_edges(
    "chatbot",
    tools_condition,
)

graph_builder.add_edge("tools", "chatbot")

graph_builder.add_edge(START, "chatbot")

memory = MemorySaver()

config = {"configurable": {"thread_id": "1"}}

graph = graph_builder.compile(checkpointer=memory)

user_input = str(input("Enter your question: "))
events = graph.stream(
    {"messages": [{"role": "user", "content": user_input}]},
    config,
    stream_mode="values",

for event in events:
    latest_message = event["messages"][-1]

assistant_reply = latest_message
```



The reasoner fetches information from the **tool_node**, outputs this information to the user and finishes the execution. There is a lot of space for improvement, e.g. adding self-reflection or toxicity guardrails, but it goes beyond the scope of this course project.

Because the agent is stateful, every query makes the agent stream all previous messages into the state_dict, which means that the agent possesses memory capabilities and is able to build relatively good conversation chains.
This part concludes our architecture explanation. The next and the last section before conclusion will focus on evaluation of the system.

EVALUATION

The last section of our report is evaluation. When it comes to Retrieval-Augmented Generation systems, evaluation can be really complicated inasmuch as the groundedness of our responses is not always well-explained by metrics. Because we wanted to be objective in our evaluation, the most natural strategy is to evaluate the retrieval and the generation parts separately. Hence, it was decided to:

- Evaluate the retriever with deterministic metrics
- Evaluate the whole pipeline using the llm-as-a-judge approach.

We have created a synthetic dataset of questions consisting of shuffled LangChain documents, from which we have extracted the metadata and passed it into an LLM to generate some factoid questions.

The prompt that was used in creating this dataset can be seen below. We have controlled for the chance that the model will generate incoherent or document-specific questions (e.g. “Who is the author of the document?” - cannot be answered out-of-context).

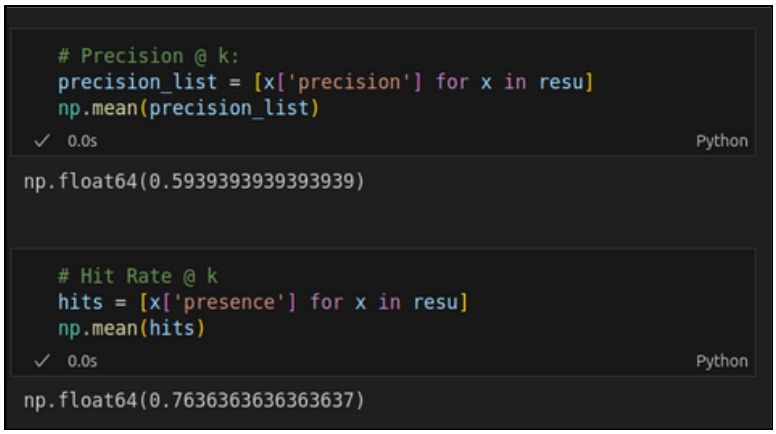
```
QA_generation_prompt = """
Your task is to write a factoid question and an answer given a context. The questions are concerned with university stuff and have
to be about some information relevant for the students. They should be natural and about some important info from the documents.
Your factoid question should be answerable with a specific, concise piece of factual information from the context.
Your factoid question should be formulated in the same style as questions users could ask in a search engine.
This means that your factoid question MUST NOT mention something like "according to the passage" or "context".

Provide your answer as follows:

Output:::
Factoid question: (your factoid question)
Answer: (your answer to the factoid question)

Now here is the context.
```

- The prompt for QA Generation

The results that we have achieved with the **k** parameter of **4** are:

- **Precision @ k**: 0.59 (URL data)
- **Hit Rate @ k**: 0.76 (URL data)
- **llm-as-a-judge**: 0.88 (URL data)
- **Hit Rate @ k**: 0.88 (Schedule Data)
- **llm-as-a-judge**: 0.85 (Form Fetcher)

It is worth noting that the **Hit Rate @ k** metric might not have been the most objective way to evaluate our system, since it only checks whether the desired document from the test set is present in the retrieved documents, but the desired information may also have been contained in other documents that are not marked as the reference.



$$\text{Hit Rate} = \frac{\text{Number of Validated 1s}}{\text{Total Pairs}} = \frac{79}{90}$$

$$\text{Hit Rate} = 0.8777 \text{ or } 87.78\%$$

Final Verdict on RAG System Performance

The RAG system performs **well** overall, with **85% of responses (51/60)** receiving a score of 2 or 3, indicating that the desired information was provided in the vast majority of cases. Specifically:

- Precision and Hit Rate for the URL data on the left, Hit Rate for both the Forms and the Schedules - above.

Nevertheless, the system provides plausible results with almost 90% accuracy score on llm-as-a-judge benchmarks, which means that the majority of responses are factually correct and contain the reference answer in them.

CONCLUSION

This section concludes our project report. In creating this RAG Chatbot and setting it up together, we have gained several important skills, and we intend to use them in our future careers. We have demonstrated the practical implementation of a Retrieval-Augmented Generation (RAG) system enhanced by agentic components to improve information accessibility within the University of Cyprus. By integrating various document sources—including web pages, forms, and complex schedule files—into a unified vectorized corpus, we enabled efficient retrieval and response generation capable of handling both English and Greek queries. Our choice of tools, such as ParseHub, Mistral OCR, and the VoyageAI embedding model, allowed us to overcome challenges like noisy data and multilingual input. Through experimentation, we found that chunk size and overlap significantly influence retrieval accuracy, highlighting the importance of careful preprocessing in RAG pipelines. The dual approach to evaluation—metric-based retrieval assessment and LLM-based response validation—ensured both objectivity and relevance in measuring system performance. Ultimately, our system not only addresses the informational gaps faced by newcomers but also lays a scalable foundation for applying RAG in public institutions, offering a compelling use case for AI-driven knowledge systems in education.

We would like to thank the coordinator of the NLP course, Demetris Paschalides, for providing us with recommendations, inspiration and workshops that reinforced our knowledge of RAG systems and NLP in general.

REFERENCES

1. Retrieval-Augmented Generation with Knowledge Graphs for Customer Service Question Answering - Xu, Wang et al.
2. From Local to Global: A GraphRAG Approach to Query-Focused Summarization - Darren Edge, Ha Trinh et al.
3. https://www.ucy.ac.cy/phy/research/forms-and-information-for-academic-personnel/?lang=en
4. https://blog.voyageai.com/2024/09/18/voyage-3/
5. https://docs.pinecone.io/guides/get-started/overview