UNIVERSITÀ
DEGLI STUDI
DI PADOVA

DIPARTIMENTO
DI INGEGNERIA
DELL'INFORMAZIONE

DIPARTIMENTO DI INGEGNERIA DELL'INFORMAZIONE

CORSO DI LAUREA MAGISTRALE IN COMPUTER ENGINEERING

Retrieval-Augmented Generation with Large Language Models for Genetic Counseling on Rare Diseases and Mutations

Relatore: Prof. Fabio Vandin

Laureando: Da Re Leonardo

ANNO ACCADEMICO 2024–2025

Data di laurea 08/04/25

Abstract

The purpose of this thesis is to develop a system aimed at streamlining the search for information regarding rare diseases and genetic mutations to assist researchers at R&I Genetics. Currently, when researchers encounter rare genetic mutations, it is likely that they have never encountered them before and that these are not well-documented. This necessitates a lengthy research phase involving the review of numerous medical articles without any certainty of finding relevant information. Depending on the specific case, this can significantly increase the time required for diagnosis. The proposed solution is a chatbot-like interface that autonomously analyzes user queries, conducts searches in relevant literature, and subsequently processes the retrieved data to produce consistent and coherent responses to the questions posed. This system integrates the Retrieval-Augmented Generation (RAG) technique with the ”Llama3.1” large language model from the American company Meta AI and ”Qwen2.5” from the Chinese company Alibaba Cloud, both of which are open-source models. However, it has been designed to be modular in terms of the large language model used, allowing it to be easily replaced with future versions, thus ensuring a good lifespan for the system.

Sommario

Lo scopo di questa tesi consiste nello sviluppo di un sistema per agilizzare la ricerca di informazioni riguardanti malattie e mutazioni genetiche rare per offrire un aiuto ai ricercatori presso R&I Genetics.
Attualmente, quando i ricercatori si trovano di fronte a delle mutazioni genetiche rare, è probabile che non le abbiano mai viste prima, e che queste non siano molto documentate, ragione per cui è necessaria una lunga fase di ricerca che consiste nella lettura di numerosi articoli medici senza la certezza di trovare informazioni rilevanti e che, dipendendo dal caso specifico, può comportare un aumento non indifferente nel tempo richiesto per la diagnosi.
La soluzione proposta consiste in un interfaccia del tipo chatbot che si occupi in modo autonomo dell'analisi delle richieste poste dagli utenti e effetui ricerche in letteratura rilevante, seguite poi da un'ulteriore fase di analisi per processare i dati raccolti e produrre una risposta consistente e coerente alla domanda posta.
Questo sistema integra la tecnica di Retrieval-Augmented Generation (RAG) con i large language models "Llama3.1" dell'azienda americana Meta AI e "Qwen2.5" della cinese Alibaba Cloud, entrambi modelli open-source, ma è stato progettato per essere modulare sotto il punto di vista del large language model, in modo che questo sia facilmente sostituibile da versioni future, garantendone una buona prospettiva di utilizzo.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

This thesis has as objective the investigation of how state-of-the-art large language models can be leveraged to support genetic counseling. The goal is to develop an easy-to-use tool that allows biologists and researchers to have access to the potentialities of LLMs in order to reduce the human workload required for information gathering and disease diagnostic.

This work has been developed during my curricular internship at R&I Genetics, an overview of the company and of the problem that this thesis is aiming to solve are explained in the following.

1.1 R&I Genetics

R&I Genetics is a private, nationally accredited laboratory specializing in diagnostics for rare genetic diseases. Established in 2007, it is based in Padova at the Torre di Ricerca Pediatrica of the foundation Città della Speranza. The laboratory is equipped with state-of-the-art genetics and genomics platforms and collaborates with major hospitals and universities, as well as many geneticists across Italy.

The process of generating genetic reports begins with a specialist's request, accompanied by a patient's tissue or blood sample, which is sent to R&I Genetics. Often parents' DNA analysis is also performed, so the amount of samples to process may vary depending on the case. The laboratory then extracts the DNA from the sample and conducts analysis using next-generation sequencing (NGS[1]).

Subsequently, bioinformatics analyses are performed to extract meaningful data and detect potential genetic mutations linked to the patient's phenotype. A biologist then interprets this data, identifying genes that carry relevant mutations to be reported.

When significant mutations related to the patient's condition are found they get documented in the final report, elaborated by a specialist. Following the ACMG (American College of Medical Genetics) guidelines, each mutation is classified into one of five categories: pathogenic, likely pathogenic, uncertain significance, likely benign, or benign.

[1] https://emea.illumina.com/science/technology/next-generation-sequencing.html

1.2 The problem

When biologists encounter rare genetic mutations, they often have not seen them before, and it is likely that these are not well-documented. Hence, to accurately categorize and diagnose genetic mutations, biologists rely on extensive scientific research, often reviewing numerous articles and spending large amounts of time meddling in medical literature. This is an unavoidable step in the pipeline of the genetic report, but it can easily become the most time-consuming, as other parts of the process are more automated and require less human intervention. Biomedical literature analysis plays a key role in ensuring precise interpretation.

The tool developed in this thesis supports biologists in this phase, aiding them in searching and analyzing biomedical literature on genes, phenotypes, and other critical information. The objective is to speed up the process of interpreting genetic mutations within the context of a patient's phenotype with automated literature research and analysis to provide the information necessary to compose the final report.

Integrating a tool capable of processing and analyzing scientific literature offers significant advantages within the workflow at R&I Genetics: given the vast number of biomedical articles published annually, it could provide a fast and efficient way to stay up to date on new discoveries when searching through genetic literature, and quickly retrieve them.

This work implements the Retrieval-Augmented Generation (RAG) framework along with the open-source LLMs Llama3.1 from Meta AI[2] and Qwen2.5 from Alibaba Cloud[3] to analyze medical literature from PubMed[4]. Design choices allow for easy replacement of the LLM thanks to the use of the Ollama framework[5], enabling for the possibility of easily updating to new and more powerful models in the future, ensuring the long-life expectancy of the project. Some insights of the use of Deepseek-r1 for this project will also be discussed as it has been considered as a potential alternative. The following chapters will explore these aspects in greater detail.

[2] http://llama.meta.com
[3] https://www.alibabacloud.com/en/solutions/generative-ai/qwen
[4] https://pubmed.ncbi.nlm.nih.gov/
[5] https://ollama.com/

Chapter 2

Large Language Models

Large language models (LLMs) are advanced artificial intelligence systems designed to process and generate human-like text. They are built on a deep learning architecture called transformer, these models are trained on vast amounts of textual data, allowing them to understand context, recognize patterns, and generate coherent responses.

LLMs, such as the well known GPT-4 [1], can perform a wide range of natural language processing (NLP) tasks, including text summarization, translation, question-answering, and content generation. Their ability to retrieve and synthesize information from large datasets makes them valuable tools in various domains, including medicine, research, and customer support.

One of their key strengths is their ability to adapt to different prompts and contexts, making them highly versatile. However, their effectiveness depends on both the quality of the training data and the prompting techniques used to guide their responses.

2.1 Overview of LLMs

Natural language processing consists in the processing and generation of human language from a machine. The research in this field began as soon as the 1940s, when World War II scientists realized the importance that an automatic machine capable of translating would have, but it was long before modern transformer architectures LLMs were created. According to the work of Xin Zhao et al.[2], the evolution process can be divided into four major stages.

The first models created were called Statistical Language Models (SLMs), based on statistical methods that emerged in the 1990s. SLMs relied on a probabilistic framework design to capture the structure of human language. The way it worked was by modeling the likelihood of word sequences in a given topic inferring the probability distribution of each word considering a finite history of preceding words. Even if they have been widely applied in the fields of information retrieval and natural language processing, they lack in terms of incrementability: the exponential number of transition probability estimation needed for high-order language models leads to the curse of dimensionality.

Following SLMs, Neural Language Models (NLMs) took advantage of neural networks like multi-layer perceptrons (MLPs) and recurrent neural networks (RNNs) to characterize the

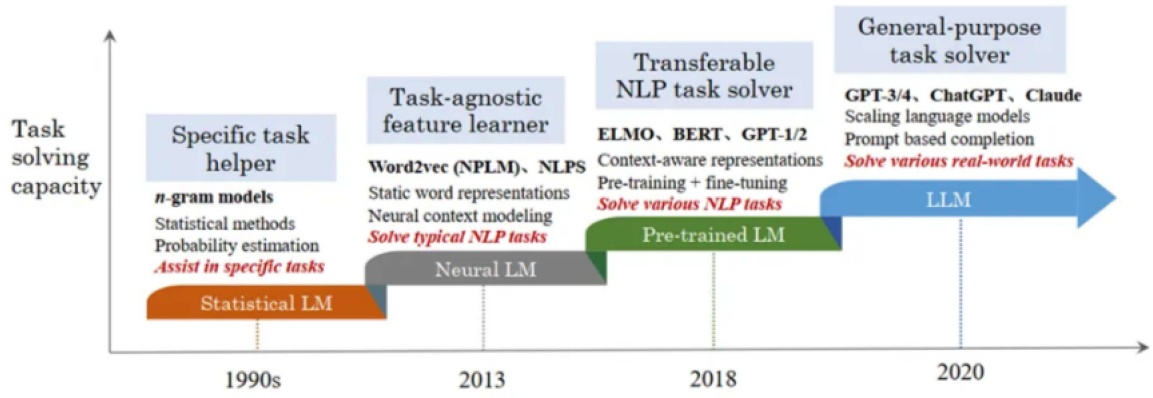


Figure 2.1: Evolution of from Statistical Language Models to Large Language Models. Image from https://medium.com/@researchgraph/the-journey-of-large-language-models-evolution-application-and-limitations-c72461bf3a6f

probability of word sequences. The concept of distributed representation of words and the word prediction function conditioned on the aggregated context features was introduced by the work of [2] to fight the curse of dimensionality which largely limited the obtainable results of the time. Word2vec [3] was introduced as a shallow neural network that could efficiently learn distributed word representations, effectively initiating the use of language models for representation learning, going beyond the concept of word sequencing models.

The next step in development were Pre-trained Language Models (PLMs), which presented a shift in how language understanding was being approached. These models are pre-trained on large-scale unlabeled data that are very effective as general-purpose semantic features, leaving the fine-tuning for a later phase, specific to the NLP task to solve.

ELMo (Embeddings from Language Models) [4] was a pioneer project in the field, taking advantage of the bidirectional LSTM (biLSTM) architecture instead of learning fixed word representations. It introduced the concept of contextualized word embeddings: unlike previous works like word2vec, it generates different embedding for the same word based on the context in which this is found, and the use of different layers of LSTMs allowed for the understanding of different levels of linguistic information, with the lower layers capturing syntactic information and the higher layers semantic information. But the limit of ELMo was his own architecture: in fact, LSTMs process data sequentially which makes computationally expensive to stack them into deeper networks.

Following the paradigm of pre-training and fine-tuning, along with the need of optimize larger networks, the Transformer architecture was introduced [5], and BERT (Bidirectional Encoder Representations from Transformers) [6] was developed by Google AI in 2018. This new model was based on the newly developed architecture, which processes words in parallel, giving it the upper hand against LSTM based models, but it also introduced bidirectional context learning, which means that BERT reads text both left-to-right and right-to-left simultaneously

and allows for better language understanding.

Large Language Models (LLMs) is the term introduced by researchers when they found that scaling PLMs both in model size and in data size improves the model capacity on downstream tasks. One of the key features of these larger-sized PLMs are the so-called *emergent abilities* [7], which are unexpected abilities not present in smaller models to solve complex NLP tasks, like for example the ability to solve few-shot tasks through in-contex learning.

Model	**Architecture**	**Directionality**	**Context Understanding**	**Use Case**
ELMo	BiLSTM	Bidirectional	Word-level embeddings	Contextual word representations
GPT	Transformer (Decoder)	Left-to-right	Sentence-level	Text generation
BERT	Transformer (Encoder)	Bidirectional	Deep context understanding	NLP tasks (QA, classification)

Table 2.1: Comparison of BERT, ELMo, and GPT

2.1.1 The Transformer Architecture

The Transformer is a specific deep learning architecture that has been the core behind the current boom of generative AI. It was introduced by Google in 2017 in the famous paper by Vaswani et al. [5].

They can be used for many different tasks, like in Vision, where Vision Transformers (ViTs) [8] have proven to be superior to the widely used Convolutional Neural Networks (CNNs), or Speech Processing for automatic speech recognition (ASR) and text-to-speech, among others, but for the sake of this work, we focus on the text-to-text application.

Transformers have a big advantage compared to the previously used architectures like LSTMs or RNNs: they process input tokens in parallel, using a mechanism called self-attention, which tells the model which words in a sentence are most relevant to each other. This makes them both faster in execution and better in capturing word dependencies in the text.

A Transformer consists of an encoder-decoder structure. The encoder is in charge of processing the input data and create contextualized representations for the words, using self-attention and feed-forward layers. The decoder generates the output step by step using masked attention, meaning that it only considers past words.

What a Transformer does is, given a sequence of words, predict the next one in the output sequence. The word is chosen over a probability distribution over all the possible words that might follow in the sequence. In order to generate a text, an initial snippet is provided, triggering the model to sample a word from the predicted distribution, then it appends it to the end of the text and repeats the process using the updated sequence as input.

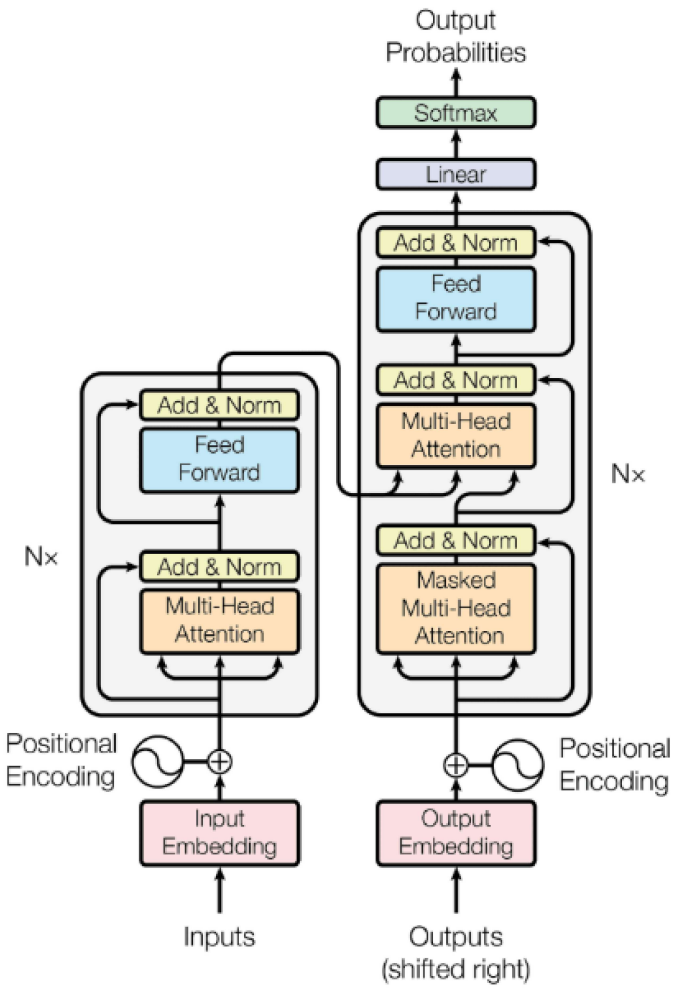


Figure 2.2: Scheme of the Transformer Architecture. Image from [5]

In more detail, the input is first broken into tokens, each of which is mapped into a highly dimensional vector that encodes its meaning, in a way that tokens of similar meaning have closer vector representations. This vectores are stored in the embedding matrix W_e, learned through training.

The vectors then pass through the attention block, which determines which tokens influence the meaning of others and updates their representation based on contextual relevance. If we, for example, consider the word light, it has different meanings in the sentences 'turn the light on' and 'the package was too light'.

A feed-forward network is attached to the attention block, further processing the vectors, and finally producing the correct encoding for the input tokens.

To obtain the probability distribution for the next token, this final vector is mapped to a list of values,one for each token in the vocabulary, using another matrix, known as the unembedding matrix W_u. The softmax function is then applied to normalize these values into probabilities. Like the embedding matrix, W_u is initially random but trained over time.

This process continues iteratively, and could extend a given text indefinitely.

2.1.2 LLMs used in this work

Initially this project was meant to implement the open-source LLM *Llama3.1* from Meta AI, but the Ollama framework, which will be discussed in the next chapter, is equipped with a standardized API that allows for same, or simlilar access to many different models. Hence, while the project has been mainly run on Llama3.1, we have tested it using also the models *Qwen2.5* from Alibaba Cloud and *DeepSeek-r1* from DeepSeek-AI. We will briefly discuss such models.

- **Llama3.1:** open-source model from Meta AI presented in [9], got released in summer 2024 in three different versions: 8B, 70B and 405B (B standing for billion of parameters) and belongs to the Llama [10] family of LLMs released by Meta AI. Due to the high computer power required for the two larger models, we have opted for the 8B version, being more lightweight and fast. It is multilingual and has a maximum context length of 128k token, making it perfect for this kind of application. Again, due to hardware limitations, the maximum number of context tokens has been limited to 10k for all LLMs used in this project, still showing very good results.
  The model features 32 Transformer layers, incorporating Grouped Query Attention (GQA) [11] with 8 key-value heads, which enhances inference speed and reduces memory usage. One of its most notable advancements is its context length—initially trained on 8K tokens, it supports 128K tokens, made possible through long-context pre-training and Rotary Positional Embeddings (RoPE) [12].
  The tokenization process has also been improved, featuring an expanded vocabulary of 128K tokens, enhancing its ability to compress multilingual content efficiently and perform well across different languages and specialized domains.

- **Qwen2.5:** the Qwen2.5 14B model, developed by Alibaba's Qwen team, is part of the Qwen2.5 series, an open-source family of model with great capabilities in language modeling, reasoning, and multilingual conversation. Designed to balance computational efficiency and performance, Qwen2.5 14B is a mid-sized dense model that builds upon previous iterations with improved pre-training, long-context processing, and reinforcement learning alignment techniques. Again, we chose this model size since it was the larger version of Qwen2.5 that was able to run on the 16 Gb of VRAM we had to our disposal, but this model is structured to provide higher accuracy and better scalability than its smaller counterparts (e.g., Qwen2.5 7B) while remaining more resource-efficient than larger models like Qwen2.5 32B and 72B.
  The model employs 48 Transformer layers, incorporating Grouped Query Attention (GQA) [11] with 40 query heads and 8 key-value heads, which improves memory efficiency and faster inference. One of its most notable enhancements is its context length—initially trained on 8K tokens, Qwen2.5 14B supports up to 128K tokens through long-context adaptation. This improvement significantly enhances performance in tasks requiring long-range dependencies, making it ideal for applications like this project.

The model's tokenization process utilizes Byte-level Byte Pair Encoding (BBPE) [13] [14] with a vocabulary of 151,643 tokens, ensuring better multilingual support and structured data processing. Additionally, an expanded set of 22 control tokens enables improved tool use and structured outputs (e.g., JSON, tables, code generation).

- **DeepSeek-r1:** in this project we also experimented with the novel DeepSeek-R1 14B model, developed by DeepSeek-AI and part of the DeepSeek-R1 series [15]. This series of models are designed to incentivize reasoning capabilities through a novel reinforcement learning (RL)-driven training pipeline. Unlike traditional models that rely heavily on supervised fine-tuning (SFT), DeepSeek-R1 prioritizes self-evolution through RL, allowing it to self-learn reasoning patterns without explicit labeling hence making it one of the most advanced open-source models in logical reasoning, problem-solving, and structured decision-making.
  The 14B version of DeepSeek-R1 is a mid-sized dense model, balancing efficiency and performance. While it inherits the reasoning-focused enhancements of the series, it is specifically optimized to provide competitive performance while maintaining a lower computational footprint compared to its larger counterparts, such as DeepSeek-R1 32B and 70B.
  This particular version is a *distilled*[16] version of the model based on Qwen2.5:14b.
  An important point to note for this model is that its raw output contains and extra portion of text that is not present in the other two models. This portion of text is contained within a tag `<think>` and explains the reasoning that is behind the model's response to a prompt, which could provide interesting insights and additional help to researchers that might want to use this project. Due to the very recent launch of Deepseek-r1 we didn't fully implement it into the project and we limited to test the two previous models, but some examples of its output will be provided in the next chapters.

	Llama3.1:8b	**Qwen2.5:14b**
Layers	32	48
Attention heads	32	40
Key/Value heads	8	8
Context length	128k	128k
Vocaulary length	128k	151k
Activation function	SwiGLU	SwiGLU

Table 2.2: Specifics of the LLMs used in this project. DeepSeek-R1 was not included in this table since the version that we have tried is actually a distilled version of Qwen2.5:14b, which means it has the same underlying architecture.

2.2 RAG

Retrieval-Augmented Generation (RAG), presented by Lewis et al[17], is the process of optimizing the output of a large language model so that it references a knowledge base outside of its training data sources before generating a response. Large language models (LLMs) are trained on vast volumes of data and use billions of parameters to generate output for tasks such as answering questions, translating languages, and completing sentences. RAG extends the already advanced capabilities of LLMs to specific domains or an organization's internal knowledge base, all without the need to retrain the model. It is a cost-effective approach to improve LLM output so that it remains relevant, accurate, and useful in various contexts.

Without RAG, the LLM takes the user's input and creates a response based on the information it has been trained on or what it already knows. But what will happen if we ask about something that is not alredy stored within the parameters of the model, either because it was not trained on that specific topic or because we asked about something that happened after its training phase? Not always LLMs will respond saying that they do not know about it, especially if we don't explicitly specify them do so. Instead they will try to answer based on what they currently know. This can lead to the generation of false information, to hallucinations or to a confusing response, which might not seem too worry in some applicatons, for use cases like the one of this thesis, where precise information about genes mutations or pathologies is essential, it can be extremely harmful.

RAG is a technique that aims to prevent this type of problems, gathering the necessary information first and delivering it to the LLM before asking him to answer the the prompt.

With RAG, an information retrieval component is introduced that uses the user's input to first extract information from a new data source, in our case the PubMed database. The user's query and the relevant information are both provided to the LLM. The LLM uses the new knowledge and its training data to create better responses.

2.2.1 RAG Procedure

The original process for a RAG system consists in three phases:

Creation of external data: we call external data the new information that exists outside the training set used to train an LLM. This data can exist in different ways, but we can consider it as a "library" that we use to store everything we might consider useful. Understandably, a structure of this type could potentially become very large, making it difficult and time-consuming to retrieve the exact information we seek with precision. For this it gets split into chunks, and every chunk gets converted into a vectorized representation.

Intuitively, with the passage of time the data stored in this library might become obsolete, and for this is important that we keep our stored data updated.

Information Gathering: the next step consists in converting the user prompt into a vector

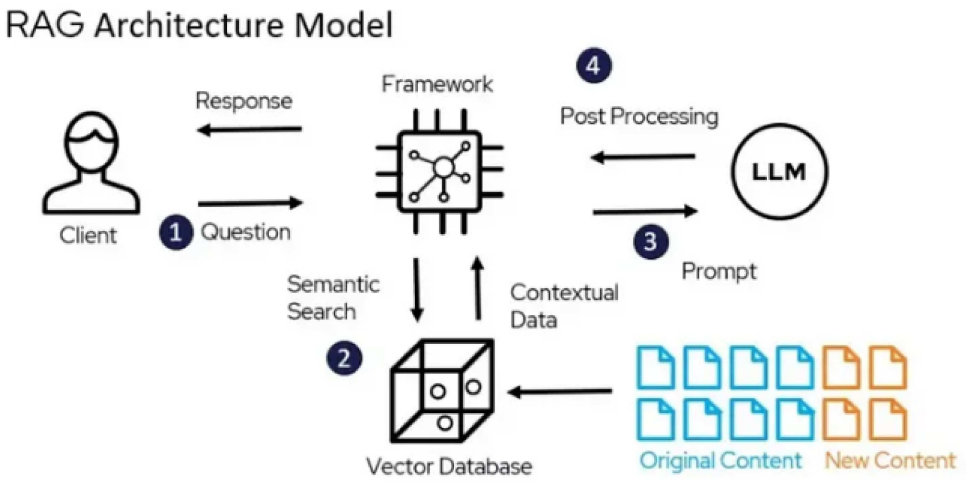


Figure 2.3: RAG system. Image from https://medium.com/@bijit211987/designing-high-performing-rag-systems-464260b76815

representation and follow with a search in our external library for relevant information. Having both the user prompt and the library stored as vectors allows us to take advantage or similarity search techniques, which dramatically speeds up the process and actually allows the use of an external library for this type of applications.

Augmenting the prompt: successively we add up the gathered information to the user prompt and pass it to the LLM to generate a response. Various techniques can be used for the creation of the optimal final prompt to pass to the model, such as chain-of-thought, to ensure the generation of a response that is up to standard. It is relevant to say that for the newly introduced reasoning models, like DeepSeek-R1, this type of prompting may not be as relevant, since they are equipped with an automated process that triggers the chain-of-thought on its own. In fact, for the examples of use of DeepSeek-R1 in this work, we used different prompts than the ones used for the other two models.

For this thesis what we want is access to the PubMed database in order to quickly gather data to respond to a researcher query. If we were to proceed like indicated in the original RAG paper we would need to download and locally store large amounts of the database, to then transform it into a vector representation, making sure that we continuously iterate this process in order to not leave out potential important new discoveries in the genetic field.

To avoid this situation in this project we take advantage of both the Entrez API [1] and the extended context length that the newer LLMs we use are equipped with.

The pipeline for a user request consists in 5 steps:

- Processing of user input
- Search of relevant paper on PubMed
- Download of the relevant documents

[1] https://www.ncbi.nlm.nih.gov/home/develop/api/

- Creation of prompt containing both context from PubMed and the user query
- Response generation by the selected LLM

The database in which we want to look into to find useful information already exists, and it's PubMed, so it would be redundant and costly to create a local copy of it, unless we had a very specific reason to do so. Instead, we use the APIs we are given, effectively eliminating the need to create and maintain a local library, along with the use of vectorization and search algorithms.

Also, considering the extended context now supported by newer LLMs, we can pass to them a lot of information to base their response on, as you will see.

Chapter 3

Methods and Implementation

3.1 Ollama Framework

Ollama[1], an acronym for Omni-Layer Learning Language Acquisition Model, is a framework designed to facilitate the local deployment and customization of LLMs. It provides a streamlined and common interface for downloading, managing, and running models directly on personal computers, without requiring cloud-based APIs or internet access. Many models are available on the Ollama library, and we can communicate with all of them with a standardized interface.

Once the model of our interest is downloaded, Ollama runs it in a local server, allowing for interaction with programs using its REST API. Depending on what action we want to perform, we post a different request to the server, for example, if we want to generate a single chat response:

Of the various functionalities provided by the Ollama API, the most important for our purposes are the following:

- **Generate:** This endpoint is used to generate a response from the model given a prompt. The prompt can be a single sentence or a paragraph, and the model will generate a response based on the context provided.
- **Chat:** This endpoint is used to chat with the model. The user can send multiple prompts to the model, and the model will generate a response for each prompt.

These are the two tools that Ollama gives us to communicate with the model, and we will use them to generate responses to user queries.

3.1.1 Generate endpoint

This endpoint is used to generate a single response from the model given a prompt[2]. The prompt can be a single sentence or a paragraph, but no context is maintained between different calls to

[1] https://ollama.com/

[2] https://github.com/ollama/ollama/blob/main/docs/api.md#generate-a-completion

this endpoint. This means that the model will generate a response based only on the prompt provided, and not on any previous prompts. The communication with the generate endpoint is done using a POST request to the local server `http://localhost:11434/api/generate` (where Ollama runs its server by default), with the model name and the prompt as the body of the request. The response from the server is a JSON object containing the generated response.

```
# headers defined previously
...
url = "http://localhost:11434/api/generate"
llm_options = {
    "model": model,
    "prompt": prompt,
    "stream": False,
    "raw": True,
    "options": {
        "temperature": temp,
        "seed": seed,
        "num_ctx": 10000
    }
}
response = requests.post(url, json=llm_options, headers=headers)
```

Figure 3.1: Example of a request to the generate endpoint.

Extra options besides the `prompt` and the `model` can be specified in the request, such as the `temperature` and the `seed`, which are used to control the randomness of the generated response. It is also here that we specify the maximum length of the context for the conversation using the option `num_ctx`. The `stream` parameter is used to specify if the response should be streamed or be returned as a single piece of text, which makes for an easier handling of the process.
An interesting feature of this endpoint is the fact that we can use a *raw* prompt when communicating with the model: this means that the model will interpret the prompt as it is, without any preprocessing and considering all the text present in the prompt as *user input*. When using this option, we can add tags to the prompt to specify what part of the prompt are instructions on how the model should behave when generating the response and what part is the actual user input.
An example of a prompt with these characteristics is shown in Figure 3.2.

In this example, the prompt is divided into three parts: the first part is the system instructions, the second part is the user input, and the third part is the assistant tag. The system instructions are used to specify how the model it should behave when generating. This part can be also used to specify guidelines about what it can or cannot disclose, if we were interested in excluding any personal information from the answer, for example. The user input is the actual prompt that the user wants to send to the model, and the assistant tags are used to specify where the model should start generating from. The model will generate a response based on the user

```
<|begin_of_text|><|start_header_id|>system<|end_header_id|>
You are a helpful research assistant. You work in the medical field,
you interface with a biologist.
You will be provided with a context and question. Your task is to
answer the question using only the provided context.
If the context does not contain the answer, just say that you don't
know, don't use your knowledge to answer.
<|eot_id|><|start_header_id|>user<|end_header_id|>

Use the following pieces of context to answer the question at the end.
Feel free to ignore irrelevant information.

###CONTEXT###
-------
{context}
-------

###QUESTION###
{question}

<|eot_id|><|start_header_id|>assistant<|end_header_id|> Answer:
```

Figure 3.2: Example of a prompt with tags for the generate endpoint.

input, but it will also take into account the system instructions to generate a response that is coherent with the context specified.

This types of prompt can be very useful and practical when we are interested in a single response to a series of questions from the user that are intended to be treated in the same way. The prompt can be easily formatted using the class `PromptTemplate` from the `langchain.prompts` package, we just need to specify the context and the question that we want to ask, and the class will take care of the rest.
An example of how to use this class is shown in Figure 3.3.

```
# context, question and template_name defined previously
...
template = importlib.resources.files('prompts').joinpath(template_name).
read_text()

prompt = PromptTemplate(
    input_variables=[context, question],
    template=template
)

prompt.format(context=context, question=question)
```

Figure 3.3: Example of prompt formatting for Figure 3.2.

Nevertheless, even if the `generate` endpoint can provide all these utilities, it does not provide a chat history, so we cannot maintain a conversation with the model. This is why we focused on the use of the `chat` endpoint.

3.1.2 Chat endpoint

This is the endpoint provided by Ollama if we intend to interact with a model maintaining a history of the conversation[3], feature that can be very useful in use-cases like the ones this project is intended to solve.

Similarly to the `generate` endpoint, we still communicate to the model using the `requests` python package, but this time we send the request to the `http://localhost:11434/api/chat` URL.

```
url = "http://localhost:11434/api/chat"
llm_options = {
    "model": model,
    "messages": [
        {
            "role": "system",
            "content": system instruction
        },
        {
            "role": "user",
            "content": context
        },
        {
            "role": "user",
            "content": question
        }
    ],
    "stream": stream,
    "options": {
        "num_ctx": 10000,
    }
}
response = requests.post(url, json=llm_options, headers=headers)
```

Figure 3.4: Example of a request to the chat endpoint.

Message formatting and handling in contrary is very different: firstly we do not have the `raw` parameter available to use, and messages are organized as a list of dictionaries. In the dictionary we can specify the `role` of the message, which can be either `system`, for instructions on how to generate the response or `user`, when the message contains the user input, and the `content` of the message, which is the actual text that we want to send to the model.

[3] https://github.com/ollama/ollama/blob/main/docs/api.md#generate-a-chat-completion

The `role` `assistant` is automatically assigned to the model's response, which is also returned in form of a dictionary.

Due to this type of organization, we do not use standardized prompts that get formatted as with the `generate` endpoint. Instead, we create a list of dictionaries, where the first item is the system instructions. If we are passing context to the model to answer a particular question every snippet of text we pass to it is considered a different message, all with `role` set to `user`, followed by a final messages containing the user actual question.

This results to be a more flexible option than `generate`, since we can also do single message completions, making sure to just pass the user input and the system instructions to the model. This is why we decided to use the `chat` endpoint for the implementation of the chatbot.

3.2 Processing of User Requests

The processing of the user input is the first active step in this project. The user input is the question that the user asks the chatbot, and it is the starting point of the conversation. Usually this question is related to gene mutations, diseases, or other genetic topics, and we have to figure out how to use it to find the necessary information on PubMed before asking for a response from the model.

The initial approach for this task was to ask the model to generate a query that would effectively return the information we were looking for. This was done by providing the model with a prompt that contained the user input and system instructions to specify what was needed.

System Prompt:	You are an assistant in biotechnology research, interfacing with a biologist. Your task is to help the biologist researching articles on PubMed by creating a search query from the input. You will be penalized if you use parenthesis in the query, keep it simple but preserve the integrity of the keywords passed.
User Input:	What is the clinical relevance of the c.927G>A variant of the ADA2 gene?
Response:	(ADA2 gene) AND (c.927G>A variant) AND (clinical relevance)

Table 3.1: Example of the generation of a query that produces no result on PubMed.

While this method has the potentiality of crafting the perfect query for the search, it is not very practical, as it requires consistency in the model's responses: firstly it is not guaranteed that the model will always generate a query that is useful for the search, and secondly, the model might generate a query that will not respect the PubMed search syntax, which would just generate zero results from the search. An example of this can be seen in Table 3.1. Other types of prompts have been tested, but even with the use of techniques like few-shot [13] results were still not consistent enough to justify their use in this project.

Instead, we decided for a more *safe* approach. As we mentioned before, in our project the context length is set to 10000 tokens, with the possibility of scaling the parameter up to 128k tokens with the right equipment. Having the possibility of using such a large context, we decided to not just do a single search on PubMed, but instead do multiple, and download multiple results per search in order to maximize the chances of retrieving the information we need.

To generate the necessary queries for this we used a fairly large system prompt:

Listing 3.1: Final prompt used to process user input.

```
You are en expert analyzing text and sentences.
The user will ask a question about genetics.
Your task is to analyze the question presented and create a
   json object with the fields:
'gene' : containing the gene cited in the question, if any,
'variants' : containing a list of the variants of that gene,
   if any. Do not include any reference snp in this list,
'subject' : containing only the main keyword of the question.
    It's important that it contains nothing more,
'keywords' : containing all the other keywords of the
   question. It's important that the subject is not included
   in the keywords. If you consider that two or more words
   are strongly enough associated insert them in the list
   BOTH as a single keyword and as multiple separated
   keywords.
'snp': containing the reference snp, if present in the
   question.
If no gene is present in the question fill the field with '
   null'.
If no variants are present in the question fill the field
   with an empty list.
If no reference snp are present in the question fill the
   field with 'null'.
Answer with ONLY the json object. Nothing else.
Example 1:
Question: 'Is the c.1367C>T variant in the RARS1 gene
   pathogenic?'
Your answer:
{
  "gene": "RARS1",
  "variants": ["c.1367C>T"],
```

```
  "subject": RARS1 gene,
  "keywords": ["c.1367C>T", "variant", "pathogenic"],
  "snp": null
}
Example 2:
Question: 'What is the alias of the ADA2 gene?'
Your answer:
{
  "gene": "ADA2",
  "variants": [],
  "subject": "ADA2 gene",
  "keywords": ["alias"],
  "snp": null
}
Example 3:
Question: 'What is the phenotype associated with Charcot-
   Marie-Tooth disease?'
Your answer:
{
  "gene": null,
  "variants": [],
  "subject": "Charcot-Marie-Tooth disease",
  "keywords": ["phenotype", "disease"],
  "snp": null
}
Example 4:
Question: 'Is the variant rs184957 pathogenetic?'
Your answer:
{
  "gene": null,
  "variants": [],
  "subject": "rs184957",
  "keywords": ["variant", "pathogenetic"],
  "snp": rs184957
}
```

With this approach we prompt the model to generate a json object using the few-shot technique. Models like Llama3.1 are greatly superior to their predecessors in json type generation, and inserting a few examples of what we expect from the model in the prompt is enough to get a good result.

The response generated will be a json object containing an analysis of the user input, indicating us if a gene or variants are mentioned, also identifies the SNP (Single Nucleotide Polymorphism[4]) if present, but the most important part for our project are the subject and keywords fields, since they are finally what we are going to use to search on PubMed. We experimented identifying other parts of the context in the case of a more complex search phase, to prevent the possible use of different databases in which information like the SNP or the gene need to be explicitly specified to trigger a search.

System prompt:	See large prompt above
User question:	What are the variants in the MCM2 gene, reported in literature, that cause hearing loss?
Answer:	{'gene': 'MCM2', 'variants': [], 'subject': 'MCM2 gene', 'keywords': ['hearing', 'loss'], 'snp': None}

Table 3.2: Example of the input processing by Llama3.1.

In general, we can tell that the model identifies all the fields that we required with fairly good consistency, and we experienced very few *hallucinations* events. An example of hallucination for the case of a very specific user input that we had is shown in the following:

User question:	What is the phenotype associated with Charcot-Marie-Tooth disease?
Subject identified:	Charcot-Marie-Tooth disease
Keywords identified:	['Charcot-Marie-Tooth', 'disease', 'phenotype', 'peripheral', 'neuropathy', 'weakness', 'muscle', 'atrophy', 'distal', 'sensory', 'loss']

Table 3.3: Example of the generation extra keywords by the model.

Here the model keeps generating more keywords than the ones that are present on the user input itself. This was more of an unexpected event than a disruptive problem, since all the extra keywords generate were still coherent with the context, besides being the only case something like this occurred, but it tells us that should keep in mind the possibility of errors from part of the model.

Another thing worth to notice is that, even with all the capabilities of current state of the art LLMs, they have a fairly good amount of randomness in them, which can lead, and will lead,

[4] https://www.genome.gov/genetics-glossary/Single-Nucleotide-Polymorphisms-SNPs

to sometimes outputs that are not what we expect. There will be cases where the subject is not correctly identified, and will be put among the keywords, or sometimes keywords will not be identified. The generation of the json object is also not perfect, and sometimes the model will insert text before or after the json object, which will make the response much harder to manage (think of the eventuality of the model adding brackets to the text or other elements that we do not expect to be present in the response). To avoid problems with the json object generation, which could lead to a crash of the system, we re-iterate the prompt to process the input in case we get something that is different.

Still, all these problems are much more run dependent that we would expect, and the model correctly identifies subject and keywords, along with a precise json object generation in the vast majority of the cases.

3.3 Information Retrieval

Once processed the input we can proceed with the actual search phase. For this we launch a series of queries to PubMed, the first containing only the identified subject, and the following containing the subject associated to each one of the keywords of the user input. In this way we want to achieve two objectives, both having as finality consistency:

- First: we want to maximize the chances of finding the information we need, so clearly doing multiple searches and looking into multiple results will result in higher consistency.
- Second: LLMs are excellent at processing text, but they are still not perfect. More than once during testing the model wrongly identified the subject of the user input, and this if further proof that we need to prevent possible errors. Doing this type of cross-search by associating keywords with the subject prevents the situation where the search phase do not produce any relevant result.

Search queries are managed thanks to the `BioPython` library, which provides a series of tools to interact with the PubMed database. The library provides the `Bio.Entrez` module, which allows access to the E-utilities, a set of nine functions built to search into NCBI (National Center for Biotechnology Information[5]) databases, which include PubMed, download the results, and parse them to extract the necessary information. The search is done using the `Bio.Entrez.esearch()`[6] function, which returns a list of PubMed IDs that match the query. The results are then downloaded using the `Bio.Entrez.efetch()` function.

Results of the search on PubMed can be sorted in different ways:

- by publishing date,
- by author,
- by journal name,

[5] `https://www.ncbi.nlm.nih.gov/`

[6] `https://www.ncbi.nlm.nih.gov/books/NBK25497/#chapter2.The_Nine_Eutilities_in_Brief`

- by relevance.

The default sorting, by relevance, is what we are most interested in, so we store the top 100 most relevant PubMed IDs for each individual search. We most likely will not need all of them, but we can keep them in memory to avoid the need of doing repeated searches on the same topics. Only the top five results per search will be downloaded and set as messages to pass to the LLM. This ensures a good amount of information being passed as context without the risk of overloading the model with possibly non-relevant data. If the context is not sufficient to respond to the user question we will always be able to iterate the fetching of articles using the IDs we stored.

Chapter 4

The Application

The main objective of this thesis was creating a tool that could be utilized on the daily by researchers, so an easy and intuitive interface is needed, reason why we chose the Streamlit framework to develop it.

The final application is a web-based tool that allows users to interact with the LLM of their choice between the ones available, in our case Llama3.1:8b and Qwen2.5:14b, with an interface similar to the now familiar ChatGPT interface, with a couple of extra features to aid in their research. For the name we decided with *Ofiuco*, the Latin name for the constellation Ophiuchus, the serpent bearer, which represents the Greek god of medicine Asclepius, as a reference to the medical applications of the tool.

In this chapter we will introduce the Streamlit framework and its main features, and then we will describe the different parts of the application, explaining the purpose of each one and how they interact with each other.

4.1 The Streamlit Framework

Traditional methods of web applications development require knowledge of multiple programming languages, such as HTML, JavaScript, and CSS, in contrary Streamlit is a Python-based open-source framework for the creation and deployment of interactive web applications, with a focus on simplicity.

It has great support for media and real-time interactivity, and comes with a rich set of widgets ready to deploy and easy to customize depending on the application. There exist different options that could be chosen instead of it, but we can see its overwhelming increasing in popularity on GitHub since 2020 (Figure 4.1).

One of Streamlit's primary goals is to eliminate the complexity associated with traditional web development. Streamlit applications are written entirely in Python, leveraging a linear script execution model that resembles the typical data analysis workflow.

Streamlit supports real-time interactivity through widgets, such as sliders, dropdown menus, and text inputs. These widgets can be seamlessly integrated into applications to create dynamic user interfaces. As the user interacts with the application, Streamlit automatically reruns the

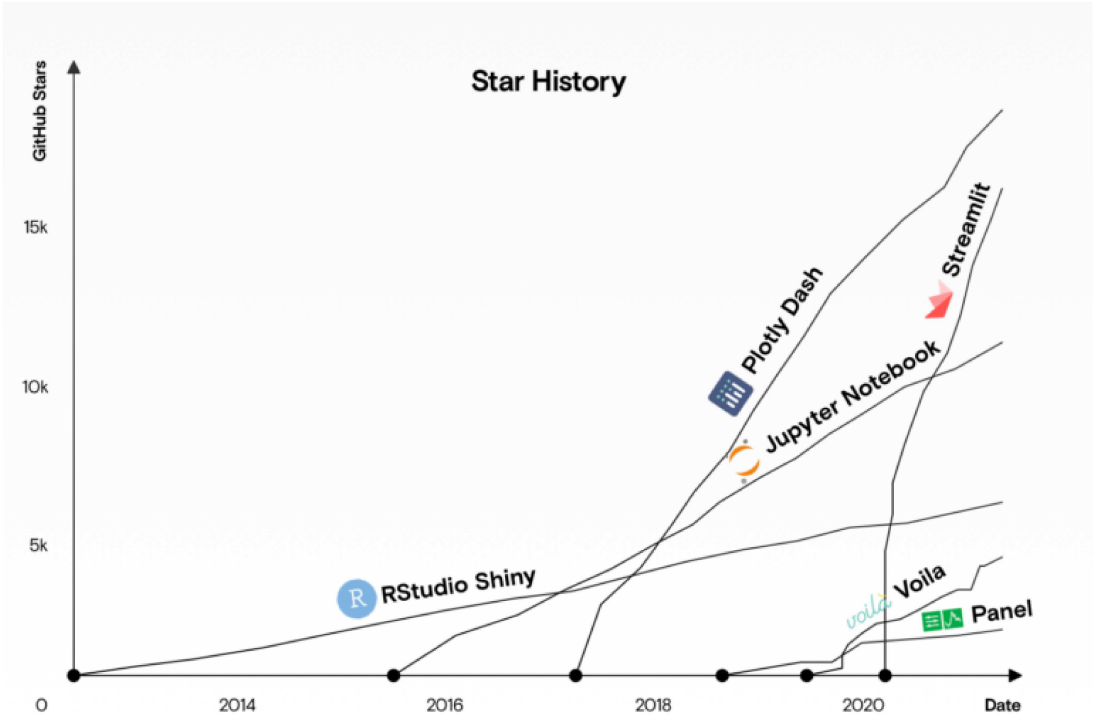


Figure 4.1: Streamlit popularity chart. Image from https://sakizo-blog.com/en/607/

code, updating the output in real time. The script is executed from top to bottom, rendering elements on the webpage as they appear in the script. Every time a widget is interacted with, the script is re-executed to update the results based on the new inputs. This behavior mimics the natural flow of data analysis in a Jupyter notebook environment, where users progressively build visualizations and models.

Between script reruns we are still able to preserve the application state. In fact, Streamlit provides a simplified state management mechanism using session state (`st.session_state`), which allows developers to store values across reruns of the script. This is useful for preserving inputs, counters, or selections as the user interacts with the application.

Streamlit applications are designed with a client-server architecture. When the Python script is run, Streamlit launches a local web server and opens the application in a web browser. The browser acts as the client, and the server (which runs the Python code) handles requests and interactions with the user. This architecture enables Streamlit apps to be deployed both locally and remotely on cloud platforms like Heroku, AWS, or Google Cloud.

Of course, all this simplicity comes at the cost of freedom in customizing the design and behavior of the application, along with more demanding computer execution time, which may make it not ideal for large scale project, but it is not a relevant issue in the case of this thesis.

4.2 Parts of the Application

As previously mentioned, Ofiuco has an interface similar to the one of ChatGPT. It is composed of two parts:

- The sidebar, where the user can select the chat and access to some settings
- The body, where the actual chatting happens. Downloaded context and articles are also displayed here to remind the user of the context of the conversation.

Initially the body of the application is not accessible, and will appear mostly blank prompting to select an already existing chat or to create a new one from the sidebar:

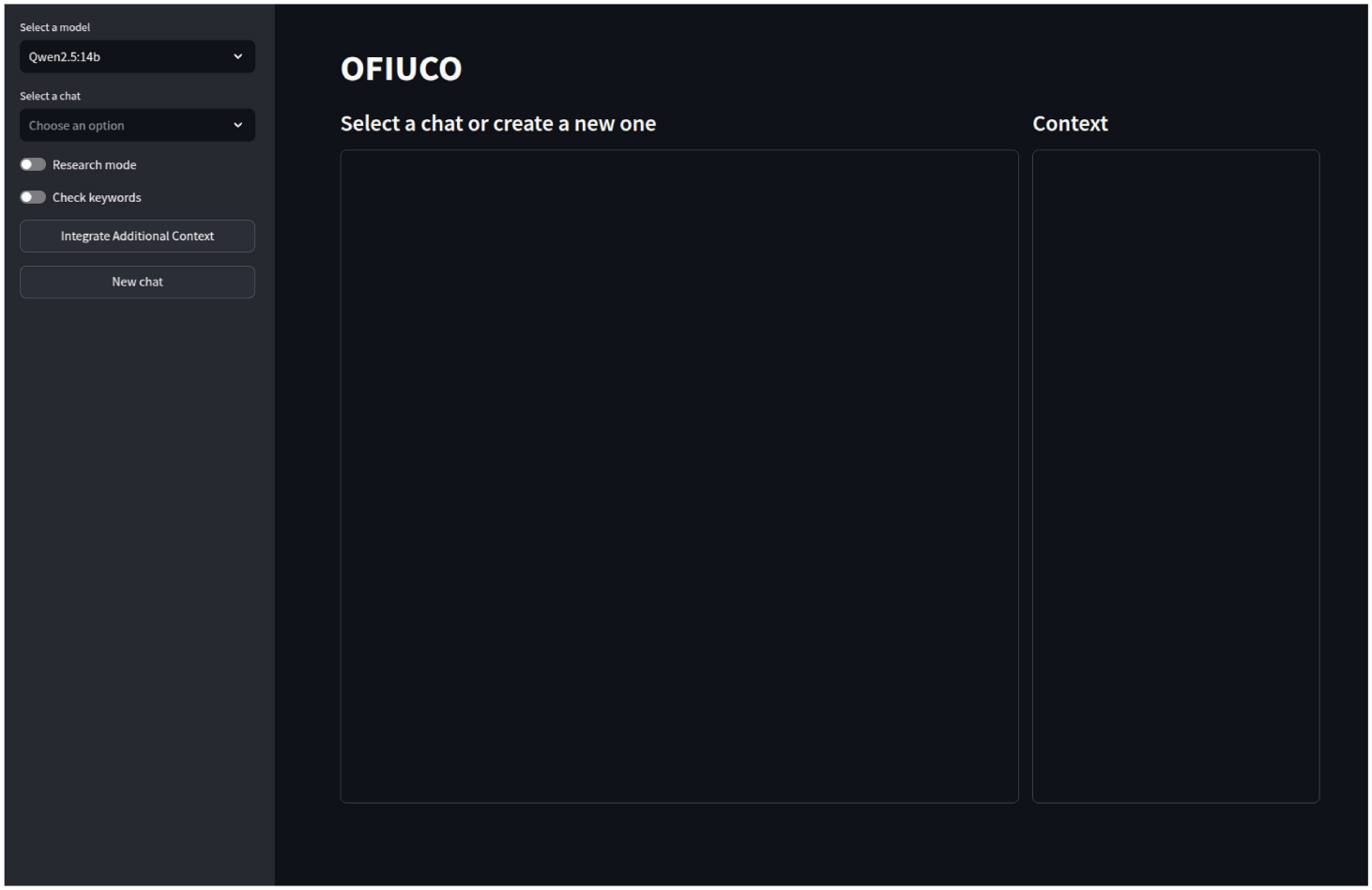


Figure 4.2: Ofiuco at launch

When, in contrary, a chat is selected, or either a new chat is created, the body will be populated with the chat history and the context of the conversation, if old messages exist, and the chat input will become available for use. The user can then start chatting with the LLM, and research on PubMed can be triggered at any moment by toggling the switch in the sidebar.

The conversation can continue indefinitely, and the user can switch between chats at any moment, the only real limitation is imposed by the context length of the model, a higher context length will allow for a longer conversation without the risk of the model forgetting the context of the conversation. On sufficiently powerful hardware, this limitation will become irrelevant, as most conversations will not last long enough to reach the context limit.

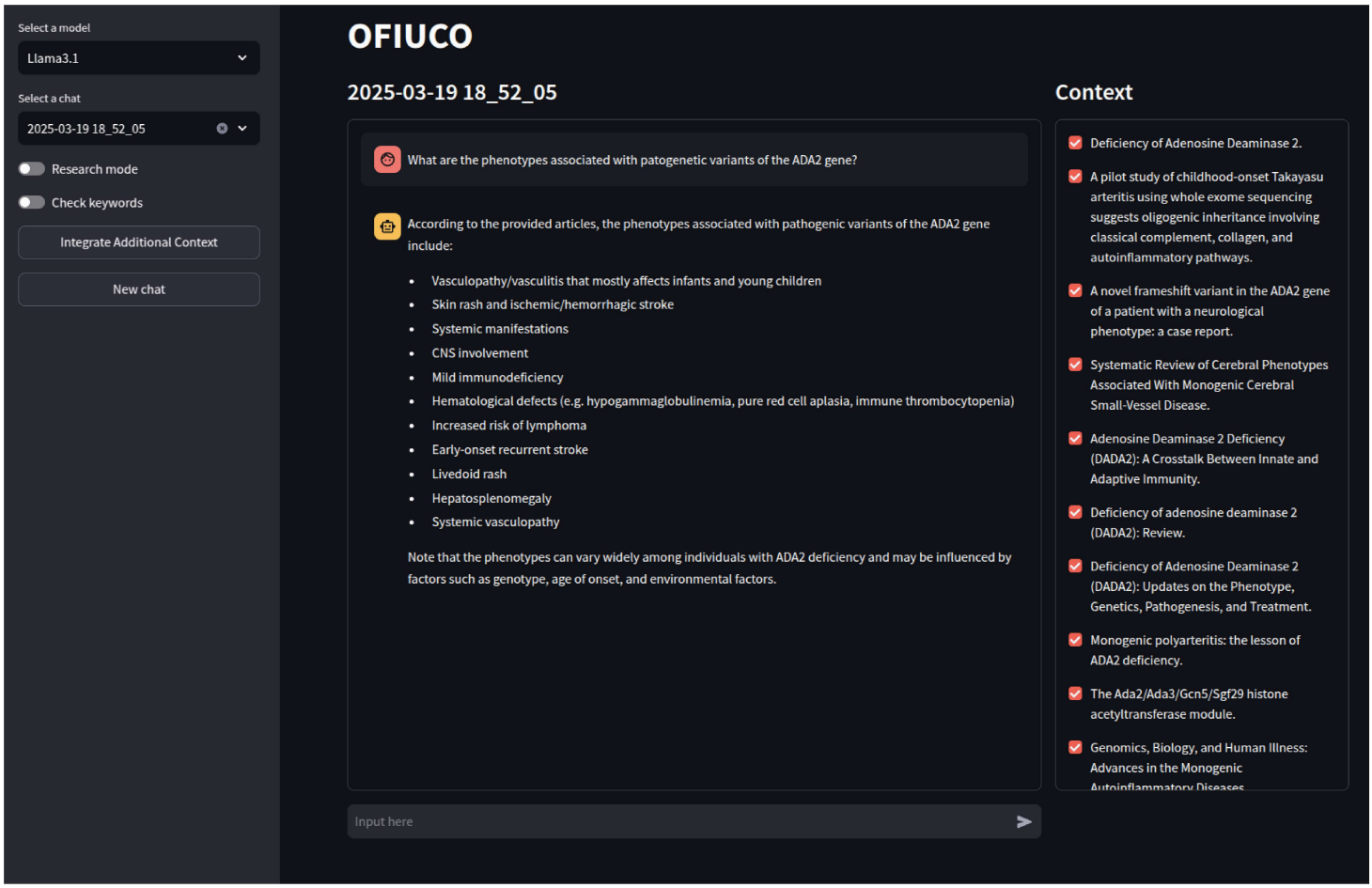
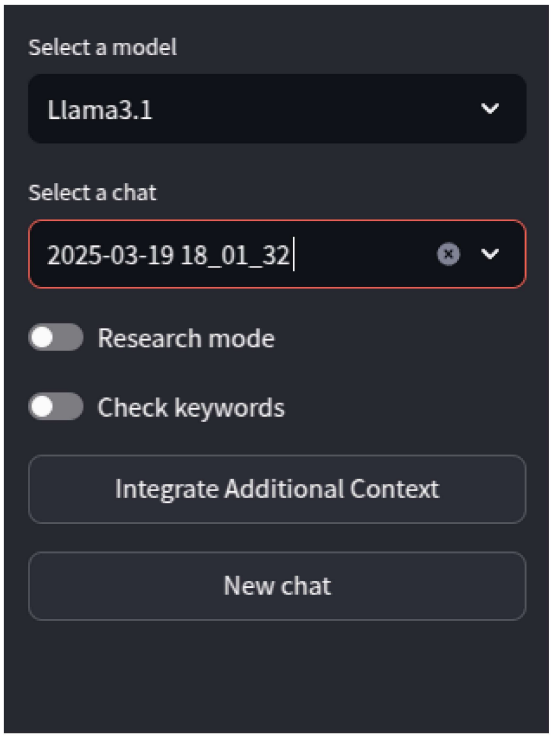


Figure 4.3: Ofiuco when a chat is selected

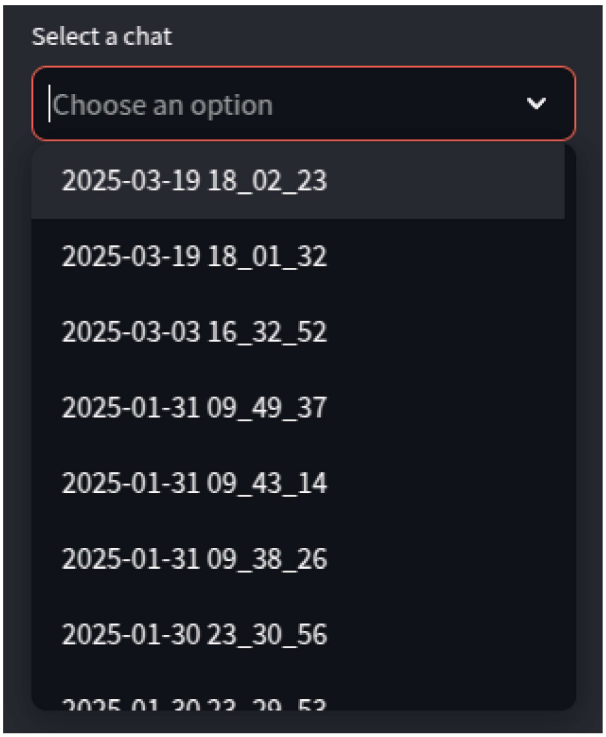
4.2.1 The Sidebar

The sidebar is the first thing the user sees when opening the application. Its main purpose is to access the different chats and settings available. The user can select a chat from the list, or create a new one by clicking the button at the bottom of the sidebar. The name for the chat is created automatically using the current date and hour to prevent repeated names.

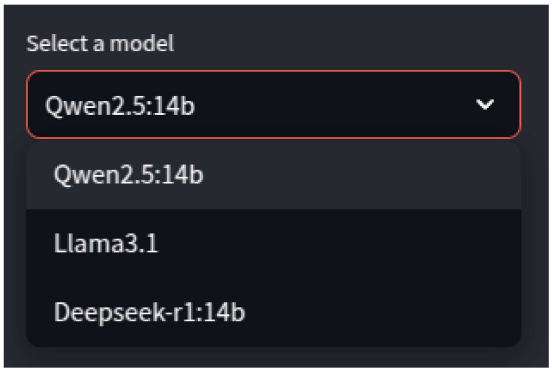
Additionally, the user can select which model he wants to talk to, currently can choose between Llama3.1:8b and Qwen2.5:14b (DeepSeek-R1 has also being added for testing). By design the model can be switched to another at any time, and any conversation can be continued with a different model without problems since previous messages and context about the conversation are passed to the new model seamlessly.



(a) Ofiuco sidebar



(b) Chat selection in Ofiuco



(c) Model selection in Ofiuco

Figure 4.4: Sidebar menus

The two toggles are strongly related, and can only be activated in three different configurations:

- Both toggles off: The application behaves normally, and the model responds directly to the user's input without any additional processing or supervision. This is mainly intended for the case of the user wanting to further continue the conversation with the model after this has responded to the first question.

- Research mode on, check keywords off: The application processes the user's input and performs a PubMed search for relevant articles before allowing the model to respond. This mode automates the research process without requiring user intervention after asking the question. This is the default configuration for new chats, but it can be activated at any moment.

- Both toggles on: The application processes the user's input and performs a PubMed search, but with additional supervision step from the user's part. In this case a window will appear to display the extracted subject and keywords, and the user can decide to modify them at will before proceeding with the search phase. This is intended to prevent the model from performing a search with incorrect or irrelevant keywords, and ultimately is the safest way to ensure the quality of the search results.

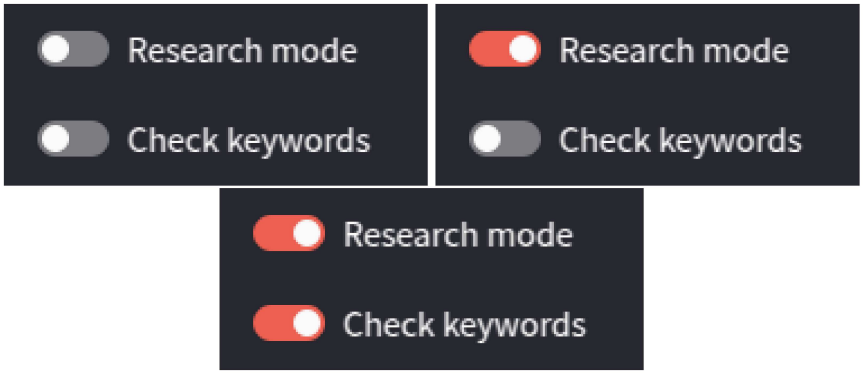


Figure 4.5: Possible toggles configurations

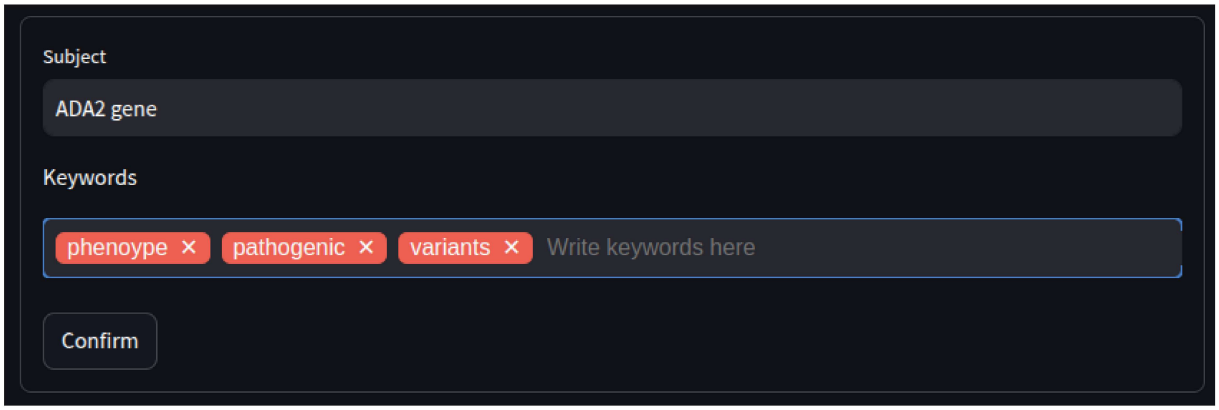


Figure 4.6: Input processing manipulation form

Finally, we have the *Integrate Additional Context* button, which is intended to add additional context to the conversation. If after a search phase the model still does not know the answer to

the question the user is doing, the options are two: either 1. starting a new search phase, maybe formulating a better question or to add important keywords through the check keywords button, or 2. add additional context through this button.

When the button is clicked, a subroutine is called that will fetch the articles IDs saved during the previous search phase and will download the next five most relevant articles per search, adding them to the context of the conversation.

This can become really useful when the context fetched during the search phase is not sufficient to properly answer the question, and could also provide with extra information that could be of interest to the user if the conversation continues.

4.2.2 The Body

The body section of the application is where the chat is located. As previously mentioned, the body will appear empty until a chat is either selected or created. Once a message is sent, spinning indicators will keep the user updated on the status of the application. There exist three different indicators:

- Processing input: the processing message will confirm to the user that the input they provided is being analyzed in order to perform a search. If the user is just chatting with the model this message will not be displayed as the model will respond directly to the user's input.

- Retrieving articles: this message will appear on when the application is fetching articles from PubMed. This will happen only after the processing input phase has been completed, or when the user clicks on the *Integrate Additional Context* button.

- Generating answers: this is the standard message that will appear every time the model is generating an answer to the user's input.

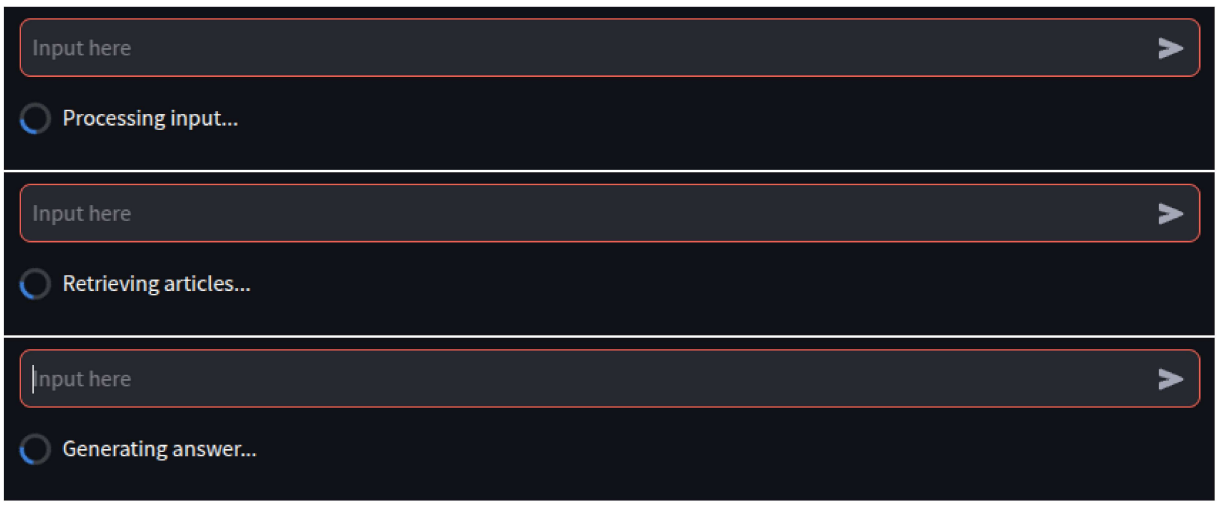


Figure 4.7: Status messages

Besides the chat, the body also contains the *Context Manager* container. This space is intended to display the context of the conversation, in other words it lists the titles of the articles that have been fetched during the search phase and that the model is using as context to answer the user's questions. This is intended to help the user keep track of the sources of the information provided by the model, as this will not always cite the source in its answers even when it is prompted to do so.

An additional functionality of the context manager is to *remove* articles previously fetched from the context of the conversation. This is thought for the scenario in which the program has fetched one or more articles that the user considers irrelevant to the conversation, or, for any other reason, does not want to be used as context. Another situation in which this could be useful is when the user wants to add more context to the conversation, but the context currently available to the model is already at the limit of the context length. In this case, the user can remove one or more articles from the context to make room for new ones.

A check button is associated with each article entry, and when clicked, the respective article will be removed from the context of the conversation. It is fair to note that removing useless information from the context could help the model to focus on the actually relevant information, and could lead to better answers.

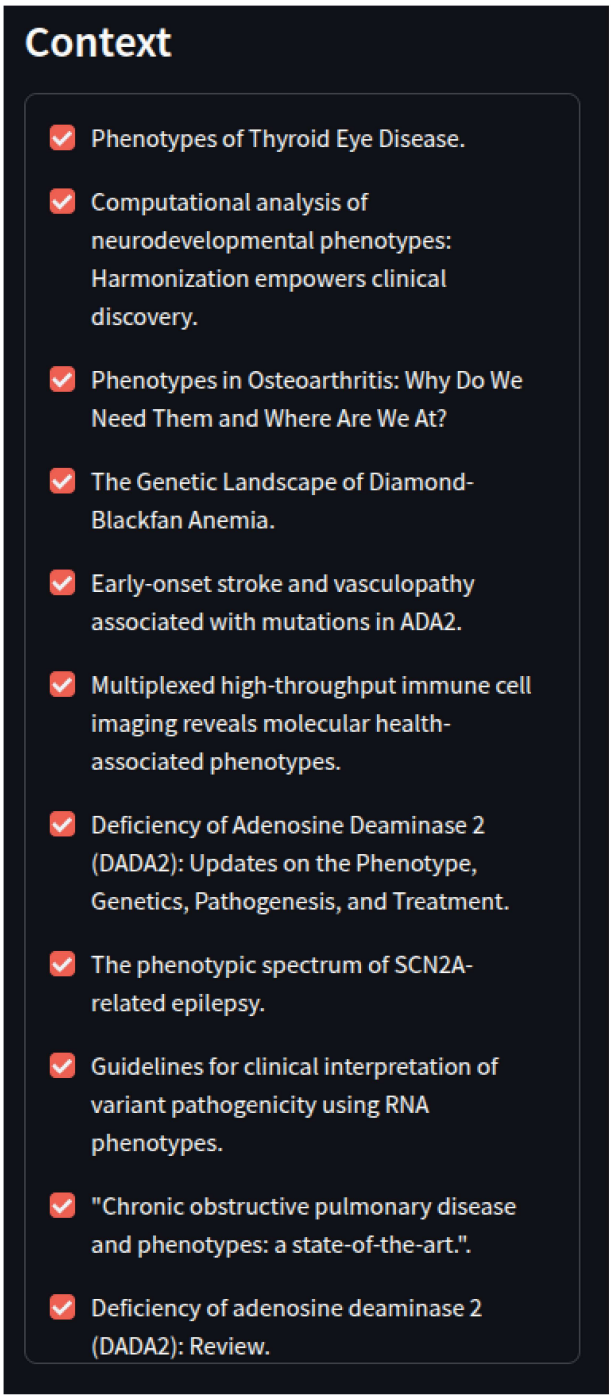


Figure 4.8: Ofiuco's context manager

Chapter 5

Testing and results

5.1 BioASQ dataset

BioASQ is an ongoing series of international challenges that focus on promoting research and development in the area of large-scale biomedical semantic indexing and question answering (QA). Organized annually, it brings together a community of researchers who work on advancing the field of biomedical text mining, information retrieval, and natural language processing (NLP).

BioASQ specifically targets the biomedical domain, which includes texts such as scientific research articles, clinical trial reports, patents, and more. The complexity of biomedical language, including specialized terminology, makes it a prime area for testing advanced NLP models.

BioASQ primarily consists of two tasks:

- **Task A**: Large-scale Semantic Indexing. In this task, participants develop systems that automatically assign relevant Medical Subject Headings (MeSH) terms to PubMed articles. MeSH is a controlled vocabulary used to index biomedical literature.
- **Task B**: Biomedical Semantic QA. In this task, participants develop systems that can automatically answer biomedical questions. The questions are based on real-world information needs of biomedical researchers and professionals. The questions are divided into four categories: factoid, questions with short concise answers; list, questions that require a list of relevant documents; summary, questions that require a summary of relevant information; and yes/no, questions that require a yes or no answer.

For this thesis, we focus on Task B, which is particularly relevant to the development of a project like this. We chose as dataset for evaluation the BioASQ 12b *golden enriched* dataset, which is the most recent test dataset available at the time of writing. The dataset consists on a series of questions, each of them associated with a set of relevant documents on PubMed and a set of relevant snippets from those documents, alongside with the type of the question (factoid, list, summary, yes/no). Each question has also and *ideal* and an *exact* answer associated to them, to allow for comparison when testing.

5.2 Retrieval evaluation

Being a RAG system divided in two parts (retrieval and generation), it is important to evaluate the performance of each part separately.

For the retrieval part, we evaluate the performance of the system in retrieving the relevant documents for a given question. We use the BioASQ 12b dataset to evaluate the retrieval performance of the system. To compare the retrieved documents with the relevant documents provided in the dataset we calculate the precision, recall, and F1 score of the retrieval system.

We chose to use these metrics because they are order-agnostic, which means that they do not take into account the order in which the documents are retrieved, as the LLM will analyze all the provided articles at the same time and will not take into consideration the order. Hence, precision (P) and recall (R) are defined as:

$$P = \frac{TP}{TP + FP} \tag{5.1}$$

$$R = \frac{TP}{TP + FN} \tag{5.2}$$

where TP is the number of true positives, which means articles that have been retrieved by the system that are also relevant, FP is the number of false positives, which means articles that have been retrieved by the system that are not relevant, and FN is the number of false negatives, which means articles that are relevant but have not been retrieved by the system.

So, precision measures the proportion of retrieved articles that are relevant according to the BioASQ dataset, while recall indicates the proportion of all relevant articles that have been actually retrieved by the system.

The F1 score is defined as:

$$F1 = 2 \cdot \frac{P \cdot R}{P + R} \tag{5.3}$$

and is the harmonic mean of precision and recall. It is a measure of a test's accuracy that considers both the precision and the recall of the test to compute the score. The F1 score can be interpreted as a weighted average of the precision and recall, where an F1 score reaches its best value at 1 and worst at 0.

5.3 Generation evaluation

For the generation part decided to use RAGAS, a metric specifically designed for the evaluation of RAG systems. RAGAS is an innovative metric that makes use of an LLM to evaluate the quality of the generated answers.

The evaluation considers three different metrics:

Faithfulness: this metric relies on the idea that the answer should be grounded in the provided context. This is very important to avoid hallucinations, a common phenomenon in RAG applications, where most likely the model does not have the correct answer in its own memory, and also to ensure that the retrieved context can be used to justify the answer.

To estimate faithfulness, first the generated answer is broken into smaller statements with the help of an LLM, then each statement is compared with the retrieved context, again with the use of an LLM, to check if the statement is supported by the context. Finally, faithfulness is calculated as:

$$\text{Faithfulness} = \frac{\text{Number of supported statements}}{\text{Total number of statements}} \tag{5.4}$$

Answer Relevance: this metric is used to evaluate the relevance of the generated answer with respect to the question. In other words, does the generated answer directly address the question? This assessment does not take into consideration factuality, but rather focuses on whether the answer is incomplete or it contains redundant information.

To estimate answer relevance, we prompt the LLM to generate n potential questions q_i for the generated answer $a(q)$. We then create embeddings for all questions locally, using Llama3.1 and a convenient wrapper present in the `ragas` python library, that allows for communication with the Ollama framework. Finally, we calculate the relevance as the average similarity between the original question and the generated questions q_i:

$$\text{Answer Relevance} = \frac{1}{n}\sum_{i=1}^{n} \text{similarity}(q, q_i) \tag{5.5}$$

Semantic Similarity: this metric assesses the semantic resemblance between the generated answer and the ground truth. It is calculated as the cosine similarity between the embeddings of the generated answer and the ground truth answer, and its value can range from 0 to 1, 1 meaning perfect aligning between generated answer and ground truth.

Answer Correctness: This metric evaluates how accurate the generated answer is compared

to the ground truth, with scores ranging from 0 to 1. It incorporates two key aspects: semantic similarity, which measures how closely the meaning of the generated answer aligns with the ground truth, and factual similarity, which assesses the accuracy of the provided information. These components are combined using a weighted approach to calculate the overall answer correctness score.‘’

5.4 Results and Considerations

For the evaluation of the system we tested both Llama3.1 and Qwen2.5, on the 85 questions present in the golden enriched 12B1 BioASQ dataset. Each question is analyzed in order to search for the context, and both retrieval and generation are evaluated twice per question: since in the application we can take advantage of the *Integrate Additional Context* button, we tested the first time with the first results obtained from the research phase, and a second time to simulate the pressing of such button.

5.4.1 Retrieval Results

The retrieval results are summarized in Table 5.1. The table shows the average precision, recall, and F1 score for both Llama3.1 and Qwen2.5 across the different runs, with and without the integration of additional context.

Model	Context Integration	Precision	Recall	F1 Score
Llama3.1	No	0.2200	0.3546	0.2715
Llama3.1	Yes	0.1886	0.4287	0.2619
Qwen2.5	No	**0.2321**	0.3615	**0.2827**
Qwen2.5	Yes	0.2024	**0.4571**	0.2806

Table 5.1: Average retrieval performance metrics for Llama3.1 and Qwen2.5 on the BioASQ 12b dataset.

As expected, our precision scores are low. This is due to the fact that, by design, our system retrieves a large amount of documents to maximize the chances of finding relevant information. In a dataset like the one we took from BioASQ a very specific set of articles is indicated as correct (golden set), and when compared to the large amount of documents retrieved by our system, it is inevitable that we end up with a large amount of false positives that lower the score.

It is worth noting that while recall is also relatively low in several cases, firstly is higher that precision, and secondly, it grows in the second iteration of the test, when we add additional articles to the context. This means that while the system produces a lot of false positives, it still

manages to retrieve relevant articles in the golden set, and, in average, when we add additional article we are able to retrieve relevant information.

Even if the retrieval scores are not as high as we would like, this does not necessarily hinder the system's ability to generate correct answers. Even when not all relevant documents are retrieved, the information that is retrieved often contains sufficient context for the generation model to produce accurate and relevant answers. This suggests that the retrieval system, despite its limitations in recall, is still effective in providing the critical pieces of information required for the downstream generation task.

The results also highlight the importance of context integration in improving the retrieval performance. By providing additional context to the models, they are better able to identify relevant documents and snippets, leading to higher recall scores. This demonstrates the value of leveraging context to enhance the performance of RAG systems in real-world applications.

There exist some cases in which integrating additional context worsens the quality of the answer, but they are not the norm and are instead limit cases. In general, the system benefits from the additional information, and if the model is able to generate a proper answer with the first results, is most likely that the user will not need to add more context to the conversation.

5.4.2 Generation Results

The generation results are summarized in Table 5.2. The table shows the faithfulness, answer relevance, semantic similarity and answer correctness scores for both Llama3.1 and Qwen2.5, with and without the integration of additional context.

Model	Context Integration	F	AR	SS	AC
Llama3.1	No	0.7208	0.7218	0.6512	0.5915
Llama3.1	Yes	**0.7923**	**0.7369**	0.6347	**0.6540**
Qwen2.5	No	0.7646	0.6707	**0.6823**	0.6328
Qwen2.5	Yes	0.7503	0.6799	0.6788	0.6401

Table 5.2: Average generation performance metrics for Llama3.1 and Qwen2.5 on the BioASQ 12b dataset. F: Faithfulness, AR: Answer Relevance, SS: Semantic Similarity, AC: Answer Correctness.

Here we can observe that despite the low scores in the retrieving phase, we can still achieve pretty good performance in the generation phase.

The faithfulness score is particularly high with respect to the others, and this is a good sign, as it means that the generated answers are well grounded in the context and the models are not likely going to hallucinate when trying to answer a question. The fact that we instructed the models to not answer if they do not have the necessary context also must be taken into consideration when looking at this data. Faithfulness is a metric that reflects how much of the

generated answer is supported by the context, but if the model answers that it does not know the answer (which would not be an hallucination, and while not ideal, it does not create potential threats) still lowers the score.

Answer relevance, semantic similarity, and answer correctness are also good, with the models being able to generate answers that are relevant to the question, semantically similar to the ground truth, and correct in terms of factual information most of the time. Of course, this can be improved, but the results are promising and show that the system has potential.

In the end is still important to remember that RAG systems evaluation is still a topic on which research is ongoing, and that the metrics used here might not be perfect, alongside the fact that the topic of text evaluation is itself conceptually complex. Yes, they are a good starting point, but in the end the most important thing is that the system provides the correct answer to the user.

5.4.3 Models time comparisons

To further evaluate the performance of the two models, we measured the time taken by each of them to process the input and generate the json object. This comparison provides insights into the computational efficiency of Llama3.1 and Qwen2.5, which is a critical factor for real-world applications where response time is essential. The results of the time comparison for the retrieval phase are summarized in Table 5.3.

Model	Avg Time	Min Time	Max Time
Llama3.1	**0.9841**	**0.7759**	**1.8157**
Qwen2.5	2.0032	1.4663	3.0049

Table 5.3: Time comparison for Llama3.1 and Qwen2.5 in processing and generating the JSON object from the input. The time is measured in seconds, and the testing have been done on a machine with an AMD Radeon 6900 XT GPU, with Ollama running on RoCM. Different machines and configurations will lead to different results.

As for the retrieval phase, we also measured the time taken by each of the models to generate the answer. We should take into consideration that the generation phase is more computationally expensive than the retrieval phase, and that is can also depend on how much material has been gathered during the retrieval phase, which will be different for all the questions, and between the models. Nevertheless, we can still compare the average time taken by the models to generate the answer, as this can be a factor to take into consideration when choosing which model to use. The results of the time comparison are summarized in Table 5.4.

Model	Avg Time	Min Time	Max Time
Llama3.1	**5.3815**	**1.3801**	**18.0899**
Qwen2.5	15.9886	2.8071	47.4695

Table 5.4: Time comparison for Llama3.1 and Qwen2.5 in generating the final answer to the question. As before, the time is measured in seconds, and the testing have been done on a machine with an AMD Radeon 6900 XT GPU, with Ollama running on RoCM. Different machines and configurations will lead to different results.

This time comparison is not necessarily for the sake of deciding which model to use, but rather show the difference in execution time that exist between *Llama3.1:8b* and *Qwen2.5:14b* when running on the same machine.

We can clearly see that Llama beats Qwen in all cases, which is likely due to the large difference in the amount of parameters that these two models have at their disposal. The 8b version of Llama3.1 weights approximately 4.9 GB, while the 14b version of Qwen2.5 weights around 9 GB.

While this should not be the main decisive factor when choosing between models for RAG applications, it could be useful in scenarios where two models perform similarly. In our case we can see that in the retrieval phase Qwen2.5 is slightly better when looking at the score, but Llama3.1 beats it in all generation scores but one. With the additional benefit of being faster we can conclude that is generally the better option for this task.

5.5 DeepSeek-R1 Examples

As we mentioned previously, DeepSeek has not been fully integrated to the system due to its very recent release. Nevertheless, being a viable substitute of currently implemented models, we decided to try its behavior on some tasks.

First task: PubMed query generation.
As for the other two models, we want to try and ask the model to generate a query that is ideal for the search on PubMed. We tried both zero-shot and few-shot approaches, and we used the same prompt used for the other models. The results are shown in Table 5.5 and Table 5.6. First we have to take notice of the reasoning part of the response in the case of the zero-shot approach. This is a very interesting feature, as it allows the user to understand how the model arrived to the conclusion, helping with transparency. The query generated is not incorrect, but we are mostly interested in a concise answer, since while we can easily parse out the reasoning part, being surrounded by the `<think>` tag, for the extra text that we get after is not as easy to extract the query.

In the case of the few-shot approach, the reasoning part is not present, and the query is more concise. This is a good sign, as it means that the model understands the example that we present him, and responds with only the query. Doing testing we still found that exist some cases in which the model is unable to generate a proper query that finds any result on PubMed, but still

this is already a promising result, and it could be considered for a future update of the system.

Second task: Extraction of information from the user input.
In this case we want to test how the model behaves if we ask it to extract information from the user input as we did with Llama3.1 and Qwen2.5, which is what ultimately we use to search information on PubMed. An example is provided in table

As we can see it correctly identifies all the data that we ask to extract, and besides some necessary parsing (the model includes some formatting text that we need to get rid of) the json object is correctly generated.

Third task: Generation of the final answer.
Finally, we show an example of the generation of the final answer to the question. We can expect fairly good results as it is reasonable that the model is able to generate a good answer, provided it has access to the correct context. It is here tho, that the reasoning part of the model's output could provide beneficial insights to the user on how the model arrived to the conclusion.

System Prompt:	You are an assistant in biotechnology research, interfacing with a biologist. Your task is to help the biologist researching articles on PubMed by creating a search query from the input. You will be penalized if you use parenthesis in the query, keep it simple but preserve the integrity of the keywords passed.
User Input:	What is the transcript of the DCAF8 gene?
Response:	<think> DCAF8 transcript Okay, so I need to figure out how to create a search query for PubMed based on the question: "What is the transcript of the DCAF8 gene?" Let me break this down step by step. First, I should understand what the user is asking. They want information about the transcript of the DCAF8 gene. A transcript refers to the RNA produced from a gene, so they're looking for studies or articles that discuss the RNA sequence or expression of DCAF8. Next, I need to translate this into keywords suitable for PubMed. The primary terms here are "DCAF8" and "transcript." Since PubMed searches are keyword-based, combining these should yield relevant results. I should consider if there are any alternative terms for "transcript." Words like "RNA," "mRNA," or "gene expression" might also be useful, but the question specifically mentions "transcript," so sticking with that is probably best to keep it simple and precise. Also, I need to avoid using any parentheses or complex syntax because the user mentioned not to use them. Keeping the query straightforward without additional tags or filters will maximize relevant results. So, putting it all together, the search query should be "DCAF8 transcript." This combination directly addresses the question by focusing on the gene and its RNA product. </think> To find information about the transcript of the DCAF8 gene on PubMed, use this search query: Search Query: DCAF8 transcript

Table 5.5: Example of the generation of a query with zero-shot approach using DeepSeek-R1

System Prompt:	You are an assistant in biotechnology research, interfacing with a biologist. Your task is to help the biologist researching articles on PubMed. The biologist will input a question, and you have to transform it in an appropriate search query for PubMed that will maximize the relevant results to anwer the question. The query should not contain any sort of tags like [MeSH], and you anwer should contain only the query itself. Example 1: [question] How does CYP1A2 relate to coffee consumption and apetite? [query] CYP1A2 coffee consumption appetite
User Input:	What are the variants in the MCM2 gene, reported in literature, that cause hearing loss?
Response:	<think> MCM2 gene variants hearing loss </think> MCM2 gene variants hearing loss

Table 5.6: Example of the generation of a query with few-shot approach using DeepSeek-R1

User Input:	What are the variants in the MCM2 gene, reported in literature, that cause hearing loss?

Response:	...after reasoning
	'''json
	{
	"gene": "MCM2",
	"variants": [],
	"subject": "MCM2 gene",
	"keywords": ["variants", "literature", "hearing loss"],
	"snp": null
	}
	'''

Table 5.7: Example of the generation of a json object from input processing with DeepSeek-R1. The system prompt utilized is the same as the one used for Llama3.1 and Qwen2.5 and shown in chapter 3. The reasoning phase has been omitted for brevity.

User Input:	What are the variants in the MCM2 gene, reported in literature, that cause hearing loss?
Response:	<think> Okay, so I'm trying to figure out what variants in the MCM2 gene have been reported in the literature as causing hearing loss. From the context given, there's a family with autosomal dominant nonsyndromic deafness where a missense variant was identified. The specific variant mentioned is c.130C>T, which changes arginine to cysteine at position 44 (p.R44C). I remember that MCM2 is part of the MCM2-7 complex involved in DNA replication. But this study found it linked to hearing loss, specifically in a Chinese family. The variant wasn't found in any control groups or databases, so it's likely pathogenic. Other studies on MCM2 focus more on its role in DNA replication and genome stability, but this seems to be the first time it's tied to hearing loss. So, I think only c.130C>T is reported as causing hearing loss in literature based on that study. </think> The variant identified in the MCM2 gene associated with hearing loss is c.130C>T (p.R44C). This missense mutation was found to cause autosomal dominant nonsyndromic deafness in a Chinese family and has been determined to be pathogenic based on its segregation with hearing loss and absence from control populations. Answer: The variant in the MCM2 gene reported as causing hearing loss is c.130C>T (p.R44C).

Table 5.8: Example of the answer generation using DeepSeek-R1. Again we ommited the system prompt as it is the same we used for the other models. Here we can appreciate the analysis of the model in the reasoning part. Providing to a researcher some form of transparency on how the model arrived to the conclusion could result to be really beneficial in some scenarios.

Chapter 6

Final Discussions

6.1 Summary of the Work

In this thesis, we explored the application of Large Language Models (LLMs), and more specifically RAG systems, in the context of information retrieval and response generation for the needs of R&I Genetics, specifically targeting the biomedical domain. Starting with an introduction to the challenges faced by the field, we analyzed how the problem could be approached and overcome with the use of these new technologies.

The result is an application powered by the Ollama and the Streamlit frameworks, that takes advantage of the latest advancements in LLMs, providing support for future and more powerful models, that uses the APIs provided from the National Center of Biotechnology Information (NCBI) to access biomedical databases like PubMed in order to provide detailed and updated information on niche topics.

The system provides the user with easy-to-approach features that allow them to query directly the literature for specific information that would normally require the analysis of multiple papers by hand, relieving the amount of word needed to diagnose patients with rare genetic diseases.

Testing and evaluation were conducted using the BioASQ dataset to assess both the retrieval and generation capabilities of the system. Results can be improved further, with more sophisticated information retrieval methods and more powerful LLMs, but we are satisfied to see that the system is still able to correctly respond to most of the questions we provided it with, which was our ultimate goal.

Overall, this thesis demonstrates that by leveraging LLMs and RAG procedures within frameworks like Ollama and Streamlit, it is possible to enhance the information retrieval and generation processes for specialized domains like R&I Genetics. The positive outcomes in retrieval and generation accuracy suggest that this approach can be applied more broadly in similar contexts, opening up avenues for future research and applications in other scientific fields.

6.2 Problematics and Risks

Of course, considering the sensibility of the medical application of systems like this one, there exist some risks that have to be taken into consideration. The main risk is the potential for the system to provide incorrect information, which could lead to misdiagnosis or other serious consequences. This is a risk that is inherent to any system that provides information, and it is important to always cross-reference the information provided by the system with other sources.

We, both developers and users, have to keep in mind that these systems ultimately do not really have an understanding of what is being said and what is the task they are working at, but rather they are trained to predict the next word in a sequence based on the previous words. This means that the system, while being able to generate coherent and relevant responses, can also generate incorrect or misleading information, and the risk of hallucinations is always present.

Model bias can also be a problem, as the models are trained on a specific dataset and are provided with specific guidelines on how to respond to interactions. It is true that in a RAG system the model is not as dependent on the training data as in other applications, but measures to prevent bias influence should still be taken into consideration for the sake of the users and people, like patients, that might get impacted by the performance of systems like this.

Another important note is privacy. In current time privacy is a very important and sensible topic, and often big tech companies are criticized for the way they handle user data. When DeepSeek-R1 launched in January, many accuses arose affirming that the data on which the model had been trained was stolen or illegal. The truth is that we cannot know. Large language models require huge amounts of data to be trained, and the datasets are not made public by the companies that create them. Even the use of free platforms like the ChatGPT free plan, or other free LLMs available online is not free of this risk. How can we really be sure that the data present in the conversations we have with these models is not stored and used to train other ones? The answer is that we cannot. And this is why a self-hosted system, that makes use of only open-source models and APIs, like the one we developed, is really a good option if we want to make sure that we can leverage at maximum the power of these models while keeping the data private and secure.

6.3 Future Work

The system could benefit from further development and the integration of additional features, both to improve its performance and to enhance its usability and user experience. Some of the possible future directions for this work include:

- **Integration of more powerful LLMs:** In this project we were limited by the hardware we had at our disposal. If this system were to be deployed in a production environment, it would be beneficial to use more powerful LLMs, such as larger versions of the ones already implemented or even paid services like OpenAI's GPT-4o.

- **Enhanced information strategies:** Improvements to the information retrieval process should be the main focus for development, since the quality of the responses generated by the system is directly dependent on the quality of the information retrieved. In this project we decided to go for a strategy that could maximize the number of relevant documents gathered, but at the cost of also retrieving many redundant ones. A more sophisticated approach could be implemented, for example merging the initial approach, which was the generation of a PubMed query based on the question, with the one that we implemented, to improve the chances of collecting documents that contain the information we need.

- **User interface improvements:** The user interface could be enhanced to provide a more intuitive and user-friendly experience, with features such as autocomplete, spell-checking, and more advanced search options. The system could also be extended to support additional languages, specifically Italian, in order to make it easier to use for the researchers at the R&I Genetics lab.

- **Integration with other databases:** The system could be extended to integrate with other biomedical databases, such as GenBank or OMIM, which could help in retrieving more specific information about genetic diseases and mutations. Often, databases that collect and display biomedical data are locked behind paid APIs, so this would require a more in-depth analysis of the costs and benefits of such an integration.

- **Evaluation on real-world data:** Giving access to the system to the information already accessible internally at the R&I Genetics lab's database would allow us to easily retrieve past information that might turn out to be useful to a current case. Additionally, this would allow us to better tailor the system to the needs of the researchers, and evaluate its performance on the real-world data it has been designed for.

- **Additional modes:** Additional interaction modes different from the chatbot could benefit the system for the applications needed at R&I Genetics. For example a deep-search mode that terminates in the creation of an analysis report could be implemented to initialize a long-running search for a specific topic, to make sure that the system is able to retrieve all the relevant information available on the topic no matter how little of it is available on the internet. This could comprehend automatic self-reprompting of the system to continuously check on the information retrieved so far, in order to ensure high quality responses to the most delicate questions.

- **Automatic pruning of redundant information:** The system could be extended to automatically prune redundant information from the retrieved documents. This would pair well with the deep-search mode, as it would allow the system to continuously refine the information it has gathered. For this exist techniques like Pairwise Ranking Prompting (PRP) introduced by [18], that leverage LLMs to automatically rank the relevance of the information retrieved.

Bibliography

[1] OpenAI et al. "GPT-4 Technical Report". In: *arXiv preprint arXiv:2303.08774* (2023). Version 6, last revised 4 Mar 2024. arXiv: 2303.08774 [cs.CL]. URL: https://doi.org/10.48550/arXiv.2303.08774.

[2] Wayne Xin Zhao et al. "A Survey of Large Language Models". In: *arXiv preprint* (2023). DOI: 10.48550/arXiv.2303.18223. URL: https://arxiv.org/abs/2303.18223.

[3] Tomas Mikolov et al. "Distributed Representations of Words and Phrases and their Compositionality". In: *arXiv preprint* 1310.4546 (2013). Available at https://arxiv.org/abs/1310.4546. arXiv: 1310.4546 [cs.CL].

[4] Matthew E. Peters et al. "Deep contextualized word representations". In: *arXiv preprint* 1802.05365 (2018). NAACL 2018. Available at https://arxiv.org/abs/1802.05365. arXiv: 1802.05365 [cs.CL].

[5] Ashish Vaswani et al. "Attention Is All You Need". In: *arXiv preprint* 1706.03762 (2017). Available at https://arxiv.org/abs/1706.03762. arXiv: 1706.03762 [cs.CL].

[6] Jacob Devlin et al. "BERT: Pre-training of Deep Bidirectional Transformers for Language Understanding". In: *arXiv preprint* 1810.04805 (2018). Available at https://arxiv.org/abs/1810.04805. arXiv: 1810.04805 [cs.CL].

[7] Jason Wei et al. "Emergent Abilities of Large Language Models". In: *arXiv preprint* 2206.07682 (2022). Transactions on Machine Learning Research (TMLR), 2022. Available at https://arxiv.org/abs/2206.07682. arXiv: 2206.07682 [cs.CL].

[8] Alexey Dosovitskiy et al. "An Image is Worth 16x16 Words: Transformers for Image Recognition at Scale". In: *arXiv preprint arXiv:2010.11929* (2020). Version 2, last revised 3 Jun 2021. arXiv: 2010.11929 [cs.CV]. URL: https://doi.org/10.48550/arXiv.2010.11929.

[9] Aaron Grattafiori et al. "The Llama 3 Herd of Models". In: *arXiv preprint arXiv:2407.21783* (2024). Version 3, last revised 23 Nov 2024. eprint: 2407.21783 (cs.AI). URL: https://doi.org/10.48550/arXiv.2407.21783.

[10] Hugo Touvron et al. "LLaMA: Open and Efficient Foundation Language Models". In: *arXiv preprint arXiv:2302.13971* (2023). arXiv: 2302.13971 [cs.CL]. URL: https://doi.org/10.48550/arXiv.2302.13971.

[11] Joshua Ainslie et al. “GQA: Training Generalized Multi-Query Transformer Models from Multi-Head Checkpoints”. In: *arXiv preprint arXiv:2305.13245* (2023). Version 3, last revised 23 Dec 2023. arXiv: 2305.13245 [cs.CL]. URL: https://doi.org/10.48550/arXiv.2305.13245.

[12] Wenhan Xiong et al. “Effective Long-Context Scaling of Foundation Models”. In: *arXiv preprint arXiv:2309.16039* (2023). Version 3, last revised 14 Nov 2023. arXiv: 2309.16039 [cs.CL]. URL: https://doi.org/10.48550/arXiv.2309.16039.

[13] Tom B. Brown et al. “Language Models are Few-Shot Learners”. In: *arXiv preprint* 2005.14165 (2020). Available at https://arxiv.org/abs/2005.14165. arXiv: 2005.14165 [cs.CL].

[14] Changhan Wang, Kyunghyun Cho, and Jiatao Gu. “Neural Machine Translation with Byte-Level Subwords”. In: *arXiv preprint arXiv:1909.03341* (2019). Version 2, last revised 5 Dec 2019. arXiv: 1909.03341 [cs.CL]. URL: https://doi.org/10.48550/arXiv.1909.03341.

[15] DeepSeek-AI et al. “DeepSeek-R1: Incentivizing Reasoning Capability in LLMs via Reinforcement Learning”. In: *arXiv preprint arXiv:2501.12948* (2025). Version 1, last revised 22 Jan 2025. arXiv: 2501.12948 [cs.CL]. URL: https://arxiv.org/abs/2501.12948.

[16] Cheng-Yu Hsieh et al. “Distilling Step-by-Step! Outperforming Larger Language Models with Less Training Data and Smaller Model Sizes”. In: *arXiv preprint arXiv:2305.02301* (2023). Accepted to Findings of ACL 2023. arXiv: 2305.02301 [cs.CL]. URL: https://arxiv.org/abs/2305.02301.

[17] Patrick Lewis et al. “Retrieval-Augmented Generation for Knowledge-Intensive NLP Tasks”. In: *arXiv preprint* arXiv:2005.11401 (2020). Accepted at NeurIPS 2020. Available at https://doi.org/10.48550/arXiv.2005.11401. DOI: 10.48550/arXiv.2005.11401. arXiv: 2005.11401 [cs.CL].

[18] Zhen Qin et al. “Large Language Models are Effective Text Rankers with Pairwise Ranking Prompting”. In: (2023). arXiv: 2306.17563 [cs.CL]. URL: https://arxiv.org/abs/2306.17563.