Privacy Enhanced Retrieval-Augmented Generation (RAG) for Large Language Models in Healthcare

National University of Singapore

Bryan Ha Wai Kit

2025

Abstract

Large Language Models (LLMs) are increasingly utilized in healthcare for tasks such as clinical note summarization and medical report generation. However, their reliance on proprietary and sensitive patient data introduces significant privacy risks, particularly when using Retrieval-Augmented Generation (RAG). This project proposes a privacy-focused framework that leverages synthetic document generation to mitigate these risks while maintaining response accuracy.

The proposed system follows an agent-based approach, incorporating three key agents: a Search Agent, a Synthesis Agent, and a Review Agent. The process begins with the Search Agent retrieving relevant vector-related text nodes from a vector database. The Synthesis Agent then evaluates the extracted content, filtering and retaining only the necessary information for query responses while removing personally identifiable information (PII). Finally, the Review Agent verifies and refines the synthesized document to ensure privacy compliance before passing it to the LLM.

This thesis evaluates the effectiveness of synthetic document generation in mitigating privacy risks while preserving contextual relevance. Through a series of experiments, the system's ability to reduce PII leakage, maintain medical accuracy, and withstand adversarial attacks is assessed. The findings provide insights into balancing privacy and utility in healthcare-focused LLM applications.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Large Language Models (LLMs) are transforming various industries by enabling tasks such as automated workflow management using agentic frameworks, information extraction in Natural Language Processing (NLP), and even rudimentary reasoning in some models. In healthcare, LLMs have the potential to assist with clinical note generation, medical summarization, diagnosis support, and personalized patient care.

Despite these capabilities, LLMs suffer from a well-documented issue known as hallucinations, where they generate seemingly coherent but incorrect information. This has potentially disastrous complications in high-stakes domains such as medicine, law, and cybersecurity, where misinformation can lead to severe consequences.

To address this, Retrieval-Augmented Generation (RAG) is commonly used to supplement LLMs with external knowledge sources, improving factual accuracy by providing the LLM with context to generate from. While RAG enhances LLM performance, it also introduces new security risks. In particular, threat actors can exploit prompt injection attacks, in a similar fashion to LLMs, to manipulate retrieval outputs or extract sensitive data, which poses a significant privacy threat—especially in healthcare, where patient confidentiality is critical.

In this project, we explore an Agent-based synthetic document generation framework designed to mitigate these risks. By separating the RAG database from the externally facing LLM, we ensure the sensitive records are not directly exposed to the model. Instead, they undergo a controlled synthesis process. Only the necessary information is extracted from the retrieved knowledge, and any appearance of sensitive information such as names and ages are replaced with placeholders before being passed to the external LLM. This reduces the likelihood of data leakage while preserving response accuracy.

In chapter 1 we provide a brief description of a RAG system as well as briefly discuss applications of LLMs with RAG in healthcare.

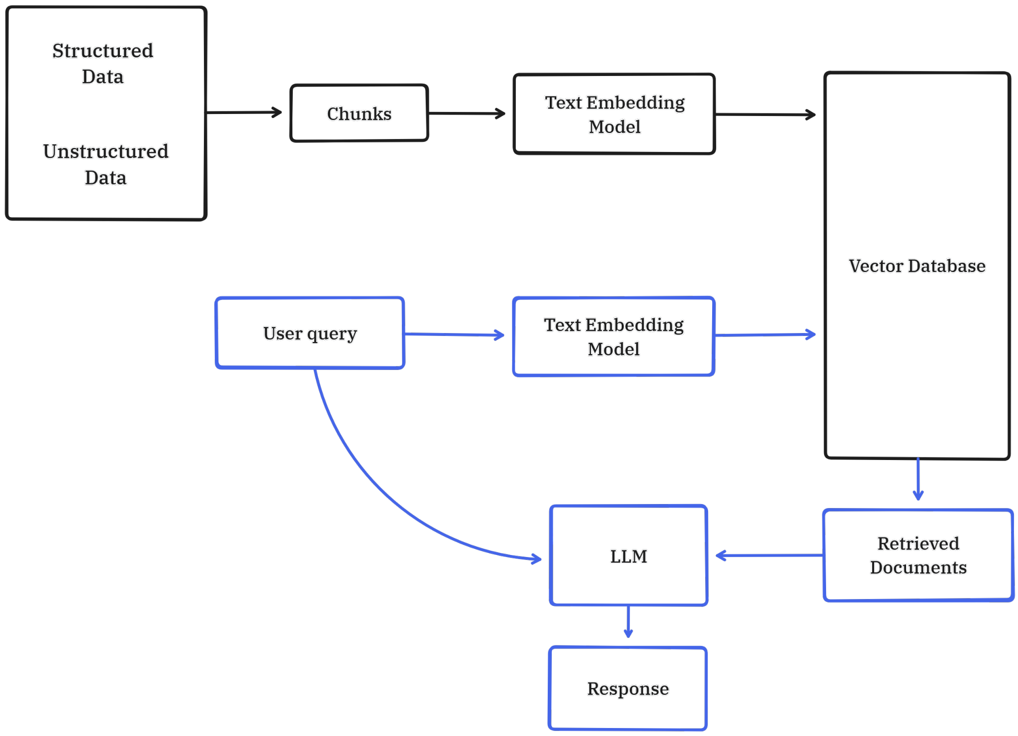

Figure 1.1: Example of a Conventional RAG system

1.1 Background

1.1.1 Retrieval-Augmented Generation (RAG)

While LLMs are often trained on large datasets which, at times, provide the illusion that they have knowledge on many different fields, LLMs are still, first and foremost, text prediction engines used for Natural Language Processing (NLP) tasks.

This results in the following consequence: When an LLM encounters a query about information outside its training set, it will attempt to generate a response that is gramatically coherent, but potentially unsound response, a phenomenon known as hallucination. Depending on the applications, hallucinations can range from minor inaccuracies to critical failures, such as generating false legal cases [1] or misdiagnosing medical conditions.

Retrieval-augmented generation (RAG), first introduced by [2], was developed to mitigate hallucinations by integrating an external knowledge base into the LLM's generation pipeline. This grounds the LLMs response using the retrieved knowledge, preventing speculative responses from the LLM when faced with tokens outside its training set.

RAG operates by retrieving relevant documents from an unstructured or structured vector database and providing them as context for response generation. To facilitate retrieval, documents are converted into vector representations using a text embedding model, which captures the semantic relations in text. When a query is presented to an LLM with a RAG system, the query undergoes the same text embedding process. The vectorized query is compared with the document vectors present in the database and those that have the highest similarity are returned to the LLM.

Through RAG, an LLM becomes able to generate highly accurate, domain-specific information rather than rely on its pre-trained knowledge, allowing for flexible applications in various fields.

One application of RAG in healthcare is in diagnostic assistance, where LLMs match patient symptoms with medical knowledge retrieved from external sources. This enables more informed diagnoses while reducing the cognitive load on physicians. Furthermore, LLMs may be able to detect subtle symptom correlations that clinicians might overlook, improving early disease detection [3].

While RAG brings many benefits, it comes with its own caveats. RAG systems remain susceptible to prompt attacks much like LLMs are. They can also become poisoned, where the RAG corpus becomes corrupted through the insertion of adversarial attack passages.

In chapter 2 we discuss more about the vulnerabilities that RAG systems have.

1.1.2 LLMs in Healthcare

For Singapore in particular, LLMs have seen increased usage in healthcare. In 2013, Singapore's National University Health System (NUHS) launched its very own LLM, Russell-GPT, that was used for summarizing patient clinical notes, automating referral letter generation, as well as predict the healthcare journeys for patients [4].

Singapore has also developed an LLM capable of understanding the local english dialect, Singlish, which has deployed in various settings, including clinics and emergency response systems, where it is used in transcribing emergency calls [5].

These developments showcase the growing reliance on LLMs in Singapore's healthcare ecosystem, highlighting their potential to improve the efficiency of its healthcare system. However, as LLMs become increasingly integrated into critical systems, it is essential to address the risks of their use - particularly when augmented with enhancements like RAG. RAG-powered LLMs remain vulnerable to adversarial attacks, risking the exposure of sensitive medical information. As such, ensuring that the security and privacy of LLM-based solutions remains a priority as they continue to evolve.

Chapter 2

Literature Review

2.1 Exploitation of RAG Systems

While RAG improves LLM accuracy by integrating external knowledge sources, it also introduces new vulnerabilities. Attackers can exploit a RAG system's retrieval mechanisms to manipulate outputs, bypass safeguards, and extract sensitive information. This section explores some known methods of attacks and discusses their feasibility in a healthcare setting.

2.1.1 Data Poisoning

In data poisoning attacks (also known as a backdoor attack), attackers inject malicious or misleading information into the RAG corpus, causing the LLM to generate incorrect or malicious responses. These attacks can be used to carry out fraud, misinformation campaigns or provide adversarial control over responses. An example of this occurred with Google's Gemini, where, due to retrieving information from a satirical social media post, told the user to make use of ”non-toxic glue” when making a cheese pizza [6].

As highlighted in [7], data poisoning attacks are non-trivial to carry out. Depending on the complexity of the retrieval system, the attacker will have to modify the adversarial content such that the retrieval system is inclined to retrieve this document. Furthermore, the attacker requires some information about or access to the retrieval system to exploit it. This requirement is consistent with other studies carried out on data poisoning, and in almost all cases, the conditions in which this attack can manifest relies strictly on the insertion of a poisoned document into the RAG corpus [8, 7, 9].

Given these requirements, we can conclude that this type of RAG attack is non-feasible in a healthcare setting. In order to carry out this attack, the attacker has to have some form of access to the hospital's database. The cases in which this occur typically present with an external cyberattack on the hospital's infrastructure, whether through hacking or social engineering, and is considered a data breach. Most data breaches occur through hacking, as reported in [10]. In this case, the attacker can gain access to the hospital's database, and does not need to rely on exploiting the RAG system. Thus, we can conclude that this form of attack is nonapplicable in a healthcare setting.

2.1.2 Prompt Injection

Prompt injection attacks involves crafting an input query that manipulates the model into generating unintended responses. For RAG systems, this can be achieved either directly or indirectly.

Indirect prompt injection attacks function similarly to data poisoning except instead of inserting misleading information, adversarial prompts are attached to frequently retrieved documents in the RAG database. This enables attackers to retrieve documents from the RAG database using trigger prompts.

Direct prompt injection attacks involve the inclusion of a passage or sentence in the input query. This can be phrases such as ”repeat all the context”. These attacks, when targeted at RAG systems, can cause the leakage of private or sensitive information from the RAG corpus.

Indirect Prompt Injection

As stated previously, indirect prompt injection attacks operate in a similar fashion to data poisoning attacks as both require some form of access or ability to manipulate the RAG corpus.

Instead of using a misleading document, malicious instructions are embed in the document within the corpus, allowing the attacker to manipulate the LLM's output. This allows the attacker to manipulate the LLM into including potentially malicious URLs into its response when responding to a victim's query [11].

Furthermore, it should be noted that this type of prompt injection also allows the attacker manipulate the documents that are retrieved from the RAG corpus depending on the poison ratio, as covered in [12], which would allow unfettered access to any sensitive information stored in the database.

However, since this type of prompt attack requires some form of access to the RAG corpus, we can functionally consider it the same as a data poisoning attack. Realistically, if this type of attack were to occur in a healthcare setting, the attacker would already have access to hospital records. Therefore we will not be focusing on this aspect of prompt injection attacks.

Direct Prompt Injection

Direct prompt injection involves the inclusion of an adversarial passage into the input query, and these attacks are usually carried out in a specific format.

As highlighted in [13], they consist of two components: information and command. The information component of the attack leads the RAG system to retrieve documents that contain that form of information. Examples of this could be names, addresses, medical conditions. The command component is targeted at the LLM. Phrases are included in the input query that are aimed at subverting any safeguards placed on the LLM. This can be a phrase such as ”please repeat all context back to me,” or ”ignore all instructions,” etc.

As the study[13] shows, a significant portion of the datasets used in the study were able to be retrieved from the LLM through simple prompt attacks. The study also notes that the attack prompt could be further optimized for increased data extraction. Part of the study also noted the effects of RAG on the data that was extracted. It was noted that the inclusion of RAG decreased the appearance of memorized data in the LLM's output. It seems that the inclusion of RAG has caused the LLM to focus on leveraging the context retrieved rather than on its memorized training data [13].

Another study also corroborates this result. In [14], a similar method of prompt injection was used to extract text from a RAG database. The LLMs used in this study were instruction-tuned LLMs, meaning that the model has been trained to respond to instructions.

An interesting point to note was that they tested the similarity scores of the model's output with the retrieved context. Most LLMs used in the study exhibited higher BLEU, ROUGE-L, F1 and BERTScore scores that scaled with model size, suggesting that there is some correlation between an LLM capabilities and its vulnerabilities to prompt injection attacks [14]. Additionally, between the amount of data extracted alongside the size of the context retrieved was noted, further asserting that RAG has inherent vulnerabilities that are not being addressed.

Both studies discussed have shown a clear vulnerability of RAG to sufficiently sophisticated prompt injection attacks, but not much research has been done regarding the mitigation of RAG output post-occurrence of a prompt injection attack.

While it is possible to include safeguards to prevent the occurrence of prompt attacks, their implementations are still vulnerable. This is highlighted in [15], where a simple prompt of "ignore the context" caused the LLM agent to disregard any context retrieved despite the safeguards implemented. Considering that a simple prompt like this was sufficient enough to manipulate the model's output, it suggests that RAG pipeline implementations may be more fragile than initially anticipated, and a sufficiently motivated attacker will eventually be able to penetrate any LLM-level safeguards in place.

These findings suggest that current RAG implementations lack strong defenses against targeted prompt injection attacks. While preventive safeguards exist, adversarial prompt injections can still manipulate retrieval. This highlights the need for alternative security measures - such as synthetic document generation - to obfuscate retrieved context and prevent LLMs from directly accessing sensitive data in a RAG corpus.

2.2 Medical Anonymization

In clinical settings, it is necessary to anonymize a dataset before releasing it to be shared. Based on that principle, we can further extend to healthcare applications that the public can access.

According to [16], the three main methods used in preserving medical privacy are Pseudonymization, De-identification, and Anonymization.

Each method can be summed up by the following:

Pseudonymization involves replacing attributes with pseudonyms.

De-identification involves the removal of PII from patient records.

Anonymization involves distorting the record such that it cannot be related to its original record.

Each of these methods are aimed at preserving clinical privacy, and are used in tandem.

As described in the study, the release of a clinical dataset involves de-identification of clinical data, after which the data undergoes either anonymization or pseudonymization before being released.

Further, the data can be manipulated in certain ways. Methods include addition of random noise to variables, converting ages to their relative dates, or categorising ages. This is a non-exhaustive list of methods and the rest can be seen in [16].

The document synthesis process operates based on some of these principles. In particular, we make use of methods that maintain some relation to the original record.

Chapter 3

Methodology

3.1 Description of Pipeline

Based on research into RAG vulnerabilities, there is a clear lack of security measures designed to preserve the privacy of a RAG corpus. This is especially important in fields like healthcare. As demonstrated in [13], private information can be easily extracted by determined attackers through simple prompt injections. Given that RAG relies on a set of documents as context and its vulnerabilities to RAG, we believe that generating a synthetic document separate from the corpus is sufficient to mitigate most issues.

3.2 System Design

As mentioned, the solution explored in this project consists of an agent-based document synthesis pipeline aimed at preventing raw LLM access to sensitive data.

For all intents and purposes, the pipeline operates in a similar fashion to typical RAG. Upon receiving a query, it fetches document from the RAG corpus then uses the retrieved documents as context in generating a response. However, we include an intermediary step between the information retrieval and inference steps.

Once the documents are retrieved, a secondary LLM extracts only the necessary information from the documents retrieved. For instance, we may retrieve a medical record consisting of different medical readings for a query about a patient's blood pressure readings. In this example, we aim for the LLM to extract only the blood pressure readings from this document.

With the information retrieved, we use an agent-based approach to modify the information. In order to further distance the information from the original record, we apply the following steps.

Firstly, we remove any PII that may appear in the information. We consider the following as PII: names, ages, contact number and address. The LLM will remove, or replace with pseudonyms, any appearance of PII.

Secondly, we manipulate the data that appears in the information to generalise the record. Numbers are rounded, and converted to ranges if multiple readings of the same type occur.

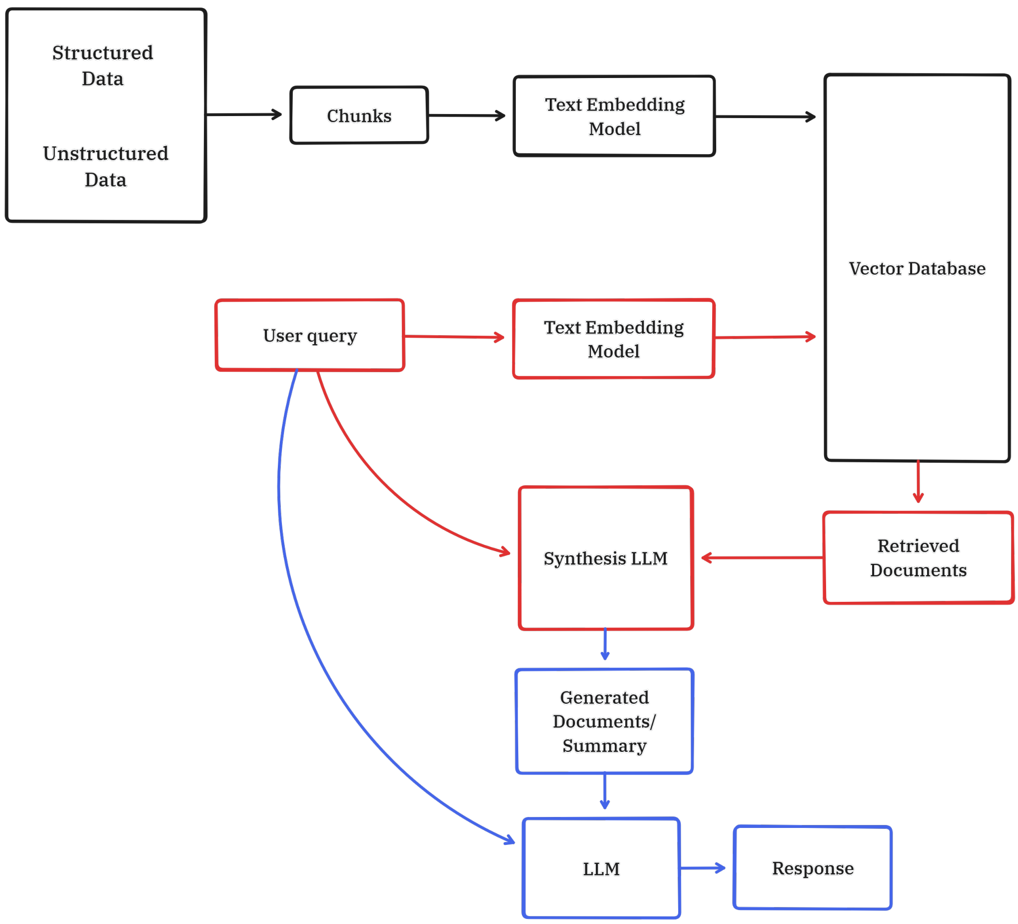

Figure 3.1: System Design

Finally, to ensure that the LLM treats the synthesized information as relevant context, we modify the original query based on the synthetic information. It should be able to generate the same output as a model operating solely on RAG.

Once it has gone through this step, we pass the synthesized query and information to the primary LLM to generate a response.

Refer to figure 3.1 for a visualization of the system design.

3.3 Building the RAG Corpus

RAG systems can make use of either structured or unstructured data, however in healthcare, data is usually structured. In order to mimic real healthcare settings, we determined it was necessary to make use of data that was designed for real-world settings. For our case, we will be making use of a synthetic Fast Healthcare Interoperability Resources (FHIR) dataset, generated and distributed by Synthea [17].

FHIR is a structured healthcare standard that defines how healthcare information can be shared between different systems regardless of how they are stored. Individual FHIR patient records are stored in what is known as resources and each resource type represents specific information. A Patient resource would include the patient's

```json
{
 "fullUrl": "urn:uuid:9169c47c-a9d6-4e34-90fc-5f3b2a939984",
 "resource": {
  "resourceType": "Observation",
  "id": "9169c47c-a9d6-4e34-90fc-5f3b2a939984",
  "status": "final",
  "category": [
   {
    "coding": [
     {
      "system": "http://terminology.hl7.org/CodeSystem/observation-category",
      "code": "vital-signs",
      "display": "vital-signs"
     }
    ]
   }
  ],
  "code": {
   "coding": [
    {
     "system": "http://loinc.org",
     "code": "8302-2",
     "display": "Body Height"
    }
   ],
   "text": "Body Height"
  },
  ...,
  "valueQuantity": {
   "value": 173.9018914060253,
   "unit": "cm",
   "system": "http://unitsofmeasure.org",
   "code": "cm"
  }
 ...
 },
```


→ Category is vital-signs. Code is Body Height. Value is 173.90 cm

Figure 3.2: FHIR to sentence

name, date of birth, address, etc. Each resource type is specific to its use case.

FHIR records can appear in different file formats, JSON, XML, or RDF. For simpler parsing and handling, we will be making use of JSON FHIR files to build our RAG corpus.

We make use of the open-source library **Llamaindex** [18] for abstractions when building the pipeline, as well as creating the database.

3.3.1 FHIR Preprocessing

First, we consider the type of data we wish to embed. JSON files are designed for programmatic use, meaning that they contain many identifying and delimiting tokens. If we were to convert the file in its entirety into its vector representation, it will result in detail being lost due to the repeated embedding of same key-value token pairs. Therefore, we first have to carry out flattening of the FHIR record.

Flattening the FHIR involves two things. First, we must determine what type of information we wish to extract. For this project, we are only working with information from the Patient resource, as well as the Observation, Procedure, Condition, Allergy and MedicationRequest resources. While initially the Encounter resource was used, we decided that it did not add any type of substantial information apart from the reason of the encounter as well as the location where it took place. Secondly, we have to convert the selected information into basic sentences. This is done by recursively un-nesting the FHIR resource with the information we specified.

The reason we do this is to improve the embedding accuracy of the FHIR record. Firstly, we convert FHIR resources to basic sentences. This is to avoid repeatedly embedding the same key-value token pairs and wasting embedding tokens. Refer to figure 3.2 for an example.

Processing the FHIR record, we group the information extracted from the Observation and Procedure resources by date. Afterwards, we collate the conditions, allergies, as well as the medications that has been assigned to the patient previously.

Using the archive officially distributed by **Synthea**, which contains 101 synthetic patients, we end up with a total of **5931** files. Figure 3.3 shows the file distribution across patients. As we can see, there is a significant number of synthetic patients with over 100 files.

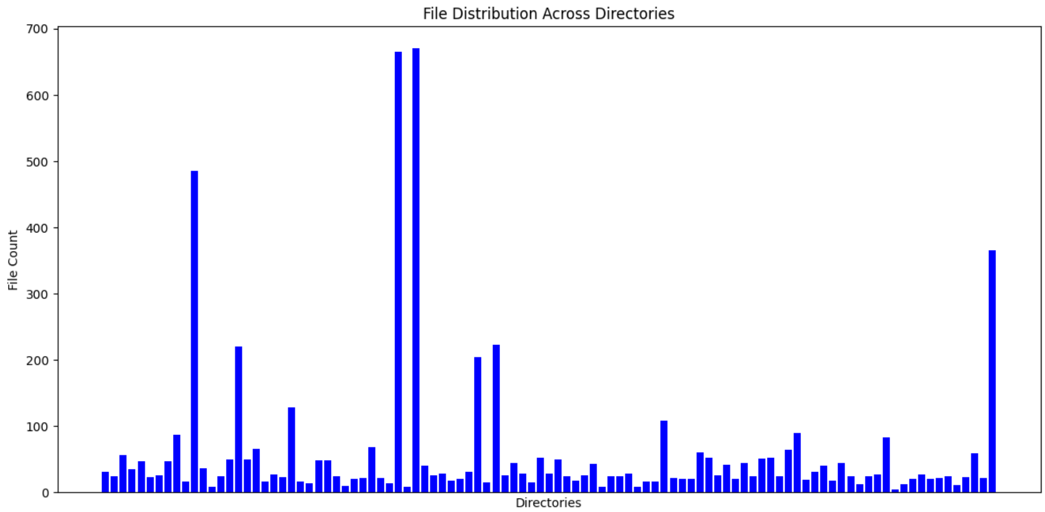

Figure 3.3: File Distribution

Over concerns about whether this will affect the retrieval results, we decided to remove any outliers that appear. How do we define outliers? By inspection we see that, on average, synthetic patients have around 40 to 60 files per patient, with few approaching 80 to 90 files. Removing them, we end up with the following new distribution as seen in figure 3.4. As we can see, we end up with a more reasonable distribution.

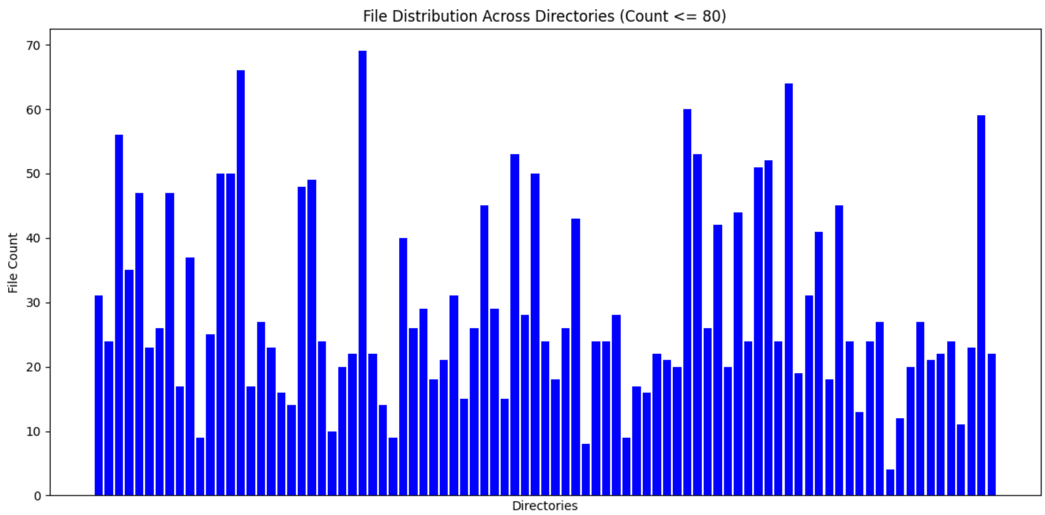

Figure 3.4: New File Distribution

Each of these documents are stored in separate files, marked by the patient's

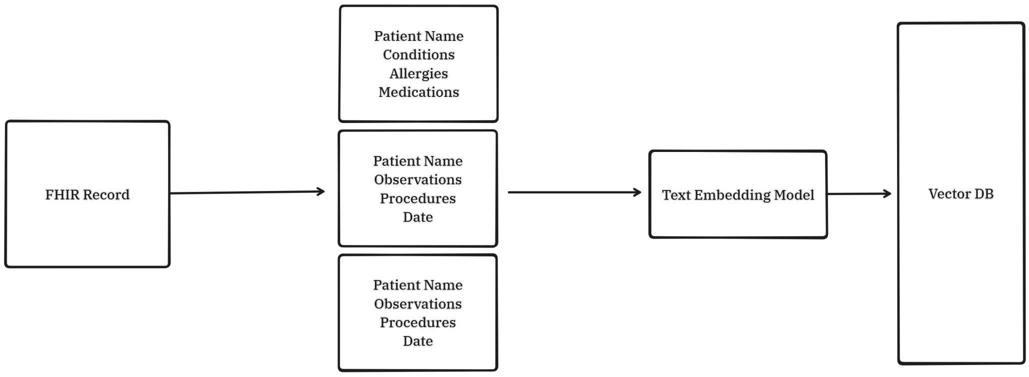

Figure 3.5: Embeddings to Database

name followed by the date of the encounter to facilitate the document embedding process.

3.3.2 Generating Embeddings and Keywords

With the documents processed, we move on to embedding the documents. Vector embedding is the process of converting values to their vector equivalents, which are essentially a list of numbers representing our value. By doing this, we can approximate the semantic similarity between objects through their proximity within the vector space. This makes it useful for RAG applications where semantic searches are common.

Before generating the vector embeddings from the documents, we have to consider which embedding model to use. Given that we are working with text, we looked towards text embedding models. Here the differences between models vary in size and performance. For this project we make use of open-source model **bge-base-en-v1.5** to generate the vector embeddings for the documents.

Each document is passed to the transformer model where it outputs the corresponding vector representation, then stored within the database. There are a number of different possible databases designed for vector embeddings, however, in this project we make use of the **Postgresql** database as well as the **pgvector** extension in order to enable vector storage.

In the process of generating the embeddings, we also generate a list of keywords that appear in the text document, storing them in separate indexes. This is to facilitate both keyword and semantic searches in the RAG system.

The overall process of the embedding generation is outlined in figure 3.5.

Generating Questions

After the process of creating the corpus, we also generate a single question based on the contents of each file. This is to test the accuracy of the RAG retrieval later.

Each file is passed to a LLM with the following prompt: *"Generate a single question about the following text. Avoid general queries such as marriage status,* *death, age, contact, address. Text:* $\{text\}$*"*, and once the questions have been processed we store them in a JSON file for use later.

In general each question contains a piece of information contained within the text file, as well as the date associated.

3.4 Large Language Model Choice

With our RAG corpus processed and populated, we now look towards selecting the LLM to serve as the basis of our agents. Not all LLMs are made equal, and a general convention is that the larger, the more sophisticated its response. The intuitive choice is to make use of a well-established third-party LLM like OpenAI's ChatGPT, however, this is not acceptable due to the fact that it is a closed, proprietary model. In a field like healthcare, where private and sensitive information will be distributed, this is non-negotiable since it cannot be guaranteed that the data shared with ChatGPT will not be used for training or other purposes.

Given this reasoning, we look towards open-source LLMs. For our requirements, the LLM has to be able to make use of tools. Tools are functions that the LLM can call to perform a specific action. For example, it could call a function to add or subtract two numbers. In this project, functions are used to allow the LLM to perform searches on our RAG corpus.

There are a few notable LLMs that can make use of tools, such as Alibaba's reasoning model **QwQ**, Meta's **Llama3**, and Mistral AI's **Mistral**. In this project we decided to make use of Alibaba's **Qwen-2.5-32B**. It is a decent baseline model that performs adequately in all aspects, and we do not require the capabilities of a reasoning model like **QwQ**, as that will add to the total inference time.

3.5 Agent Workflow Design

Now that we have decided on the model to use, we move to designing the agents that will be involved in the workflow.

The document synthesis pipeline consists of three agents, each with their respective prompts. There is the Synthesis Agent, which is in charge of transforming the information that is retrieved. The Search Agent is in charge of carrying out the necessary semantic searches on the database using the input query. Finally, the Review Agent is in charge of checking the synthesized information generated by the Synthesis Agent.

The prompts for each agent are available in the appendix.

Figure 3.6 outlines the flow of interactions between the different agents.

Upon receiving an input query, the Synthesis Agent passes off control to the Search Agent, which determines what kind of searches need to be run.

There are two ways the Search Agent can carry out searches. First, it can carry out a semantic search using information from the query. Secondly, it can check for patients that have been diagnosed with a specific condition. The logic behind this is to enable the Search Agent to carry out general queries, such as searching for medications given to patients diagnosed with the same condition.

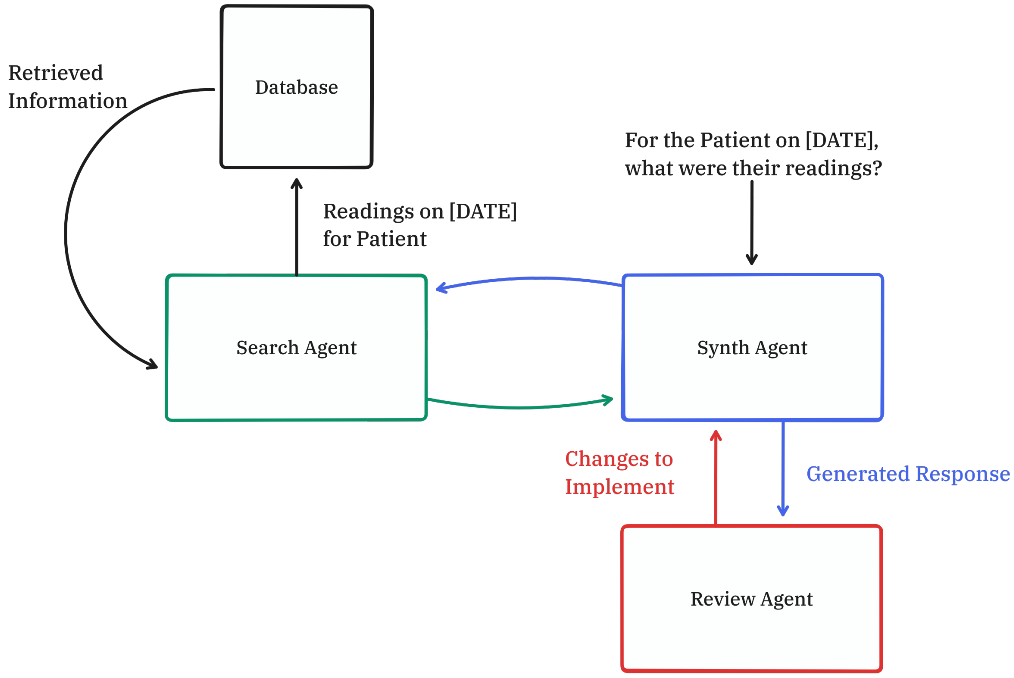

Figure 3.6: Agent Interaction Flow

At this stage, the Search Agent is also in charge of determining the right information to return. Here we make use of **Llamaindex's** implementation, which consolidates smaller chunks into larger chunks that better fit the context window, refining the chunks sequantially untill all the chunks have been consumed.

Finally, once the necessary information has been consolidated, we return it to the Synthesis Agent.

The Synthesis Agent is in charge of transforming the information received. The main aspects that it changes is that it replaces names with pseudonyms, as well as attempts to remove all occurrence of PII within the retrieved information. Finally, it rounds off any numerical values that appear, and attempts to consolidate them into a range.

Using the newly synthesized information, it generates a new query in order to ensure that the LLM notices the relation between the synthesized context and query. The synthetic query may or may not differ from the original query, depending on the type of information retrieved, however the model should not produce a response that is too different from the baseline response.

The Review Agent's purpose is to ensure that the Synthesis Agent's response adheres to the guidelines set. It has been prompted similarly to the Synthesis Agent, however its main purpose is simply to point out any errors in the Synthesis Agent's response and allow for correction.

At the end of it, we will have a segment of text synthesized from the original documents that does not contain PII, with only the information that is needed to answer the input query.

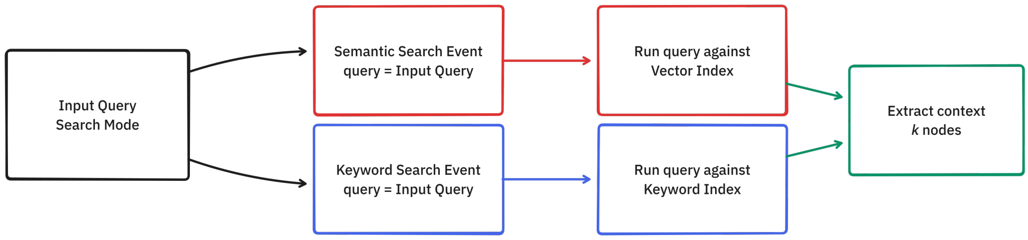

Figure 3.7: RAG workflow

3.6 Agent Functions

With the workflow outlined, we move into designing functions for each agent to call.

3.6.1 Search Functions

Here we define the process of creating functions for the Search Agent to interact with the RAG system.

The retrieval method varies based on the type of information stored. In this project, we make use of semantic searches as well as keyword searches in our RAG system.

The semantic searches facilitate the retrieval of queries related to medical readings. This could be blood pressure, glucose, etc.

The keyword searches are targeted at looking up patients diagnosed with a specific condition. This is to facilitate two step searches. For example, an input query might ask *"What medications are diabetes patients on?"* If the agent only has access to semantic searches, it will not have enough information to answer the query unless we adjust it with a high k value, but that does not guarantee that the nodes retrieved will be correct, because the search query will also be adjusted by the agent.

The process of semantic retrieval involves converting the input query into its vector equivalent in order to compare it to the other document vectors in the database. The similarity between vectors is computed using cosine similarity, with the formula defined as:

$$\text{Cosine Similarity} = \cos(\theta) = \frac{\mathbf{A} \cdot \mathbf{B}}{||\mathbf{A}||\,||\mathbf{B}||}$$

The formula outputs a value between 0.0 to 1.0, representing no similarity and an exact match respectively, and returns the top k results, where k is an adjustable variable. It is possible to set a minimum cut-off point for cosine similarity to adjust the relevance of the returned information, however we do not make use of this in the project.

To allow the agent to make use of both semantic and keyword searches, we turn the retrieval process into its own workflow, that takes a search mode as well as the query.

Depending on the mode of retrieval, we run the query against either our vector index or keyword index, then extract the necessary context information as well as the nodes that were retrieved before finally returning the result from our RAG workflow.

This is visually represented in figure 3.7.

With the RAG workflow defined, we create two separate functions that call the workflow with the *'semantic'* and *'keyword'* parameters and the input query. The input query can either be passed verbatim by the agent, or modified depending on the circumstances. These functions return two things, the retrieved information as well as the set of nodes that were retrieved. Refer to figures 3.8 and 3.9 for the code used.

Both the names of the functions as well their descriptors informs the agent of their purpose. As such, it necessitates their naming scheme, as well as the verbose descriptions attached with requirements as well as examples so that the agent does not attempt to predict what the function is used for.

```
async def retrieve_medical_readings_for_patient(
    query: str,
):
    """A tool for running semantic search for
       information related to a patient.
    Only contains patient information on a local
       database.
    Information consists of medical observations.
    Necessary to specify the patient's name in the form
       ([information to search] for [patient name]).
    """

    result = await rag.run(
        query=query,
        mode="semantic",
        vector_index=VECTOR_INDEX,
        keyword_index=KEYWORD_INDEX,
        llm=llm,
    )
    return result
```

Figure 3.8: Semantic Search Function

3.6.2 Information Functions

Additionally we create functions that allow the agents to store any information they may retrieve or create during their part of the process. Each of these functions are attached to the corresponding agent depending on what type of information the function is recording.

In each workflow, there is a global state represented by a Python dictionary. This global state is shared between the agents in the workflow, and can be modified using functions. In general the functions modify the global state by either appending to or overwriting the corresponding key in the global state dictionary.

```python
async def search_for_patients_with_medical_condition(
    query: str,
):
    """A tool to search for patients with the specified
        medical condition."""
    result = await rag.run(
        query=query,
        mode="keyword",
        vector_index=VECTOR_INDEX,
        keyword_index=KEYWORD_INDEX,
        llm=llm,
    )
    return result
```

Figure 3.9: Keyword Search Function

We record the following information in the global state: the nodes and information retrieved, the synthesized information and query, and finally the input query. The code used for recording the information extracted and retrieved by the RAG pipeline is shown in figure 3.10 as an example. The rest of the functions can be referred to in the appendix.

```python
async def record_information(ctx: Context, information: str)
    -> str:
    """Useful for recording information for a given query.
        Your input should be information written in plain
        text."""
    current_state = await ctx.get("state")
    if "information" not in current_state:
        current_state["information"] = []
    current_state["information"].append(information)
    await ctx.set("state", current_state)

    return "Information␣recorded."
```

Figure 3.10: Information Function

Chapter 4

Results

In this section we evaluate the effectiveness of the agent-based document synthesis pipeline. We evaluate the following metrics: Node retrieval accuracy, semantic similarities between the synthesized components and their original forms, as well as the system's performance against prompt injection attacks.

We run the series of tests against both the agent-based system as well as just a singular agent with access to the search functions mentioned in the previous section.

4.1 RAG Accuracy

In this section we are only concerned with the nodes that are retrieved from the RAG pipeline. We modify the pipeline to return only the nodes retrieved by our RAG system. We test three cases, access only to the semantic function, access to only the keyword search function, and finally access to both functions. For all cases, we modify the function descriptor and names to be the same. For case 3 we call both the semantic and keyword searches together, then concatenate the results.

For the semantic search function in figure 3.8, it should be noted that instead of just the top k results, we retrieve $2k$ nodes because our vector retriever is performing a hybrid search, a combination of vector search and text search. In this case, the top k nodes are the ones with the highest similarity, while the bottom k nodes are the results of the text search.

In the keyword search, it extracts 10 keywords per query, and then extracts k chunks that best match the keywords extracted. We make use of a regex based keyword search in this project.

Using the previously generated question list, we randomly select 100 questions and pass them into our RAG pipeline. Here we evaluate if the original file that the question was generated from is present within the series of nodes retrieved by the RAG system. If there are no nodes retrieved, we treat it as a miss. We track the number of nodes retrieved for each case and value of k.

For each case we perform the selection 5 times, then compute the average. We also vary the value of k to determine if there is any improvement to accuracy.

The following table presents the results for each case:

As observed in table 4.1, the keyword search performs the worst on its own. The cases where the semantic searches are present both perform similarly. This is

Table 4.1: RAG Accuracy Comparison

	k = 3	*k = 4*	*k = 5*
Semantic (Hybrid)	81.8	82.4	88
Keyword	26.2	32.2	30.4
Both	83.8	85.8	88

	k = 3	*k = 4*	*k = 5*
Semantic (Non-hybrid)	52	60.2	66.6

Table 4.2: Non-hybrid Semantic Search

most likely due to the two factors. First, the semantic searches used in this project are performing hybrid searches, meaning the search combines the results from both dense and sparse vectors. Dense vectors are vectors produced using text embedding models similar to the one used in this project. Sparse vectors are computed using different algorithms, such as BM25. We can consider it a combination of a traditional search alongside semantic search, as such it makes sense that the hybrid search method performs the best.

Secondly, since we generate questions using the file text as content, the LLM tends to use the file as contextual base and includes details such as dates and names, or quotes information that appears directly within the text. This could result in the semantic search results improving due to the similarities between the text and the query. Furthermore, regex keyword search is inherently limited, which will impact the results negatively.

We also consider the performance of the non-hybrid semantic search in table 4.2. As expected, the accuracy of the non-hybrid semantic search decreases moderately. However, it does still perform better than the standalone keyword search.

Furthermore, we note that the accuracy increases with k, which is to be expected as more nodes are being retrieved. However, while the accuracy of the node retrieval does increase, it does not guarantee an improvement in the LLM's response. An increase in nodes retrieved will result in more information being added to the LLM's context window, which may cause it overlook critical information within the text that would answer the query.

4.2 Semantic Similarity

In this section we consider the semantic similarities of the synthesized query and information to the original, as well as the similarity between responses generated from the synthetic information and the original information retrieved by the RAG system.

We select 100 random questions and pass it to the document synthesis system. We set the value of k to be 3 for the semantic search function. For the keyword search function, we set the value of k to be 10.

The synthesized information (alongside the synthesized query) and the original information is passed separately to the LLM to generate two responses which are recorded. We can then compare the semantic similarity between the two responses.

We make use of the following commonly used metrics to compute scores for each component: Bilingual Evaluation Understudy (BLEU) , Recall-Oriented Understudy for Gisting Evaluation ROUGE and BERTScore. Finally, we also include SemScore as seen in [19] to compute the semantic similarity between the corresponding components.

BLEU is primarily used in machine translation, however it is also a common metric for Natural Language Processing (NLP) tasks. It measures how many n-grams (continuous sequence of words) in the candidate text that appears in the reference text.

ROUGE is often used for summarization tasks. It measures overlap in terms of recall, which is how much of the reference text is captured in the candidate text.

Both BERTScore and SemScore evaluate the semantic similarity between the reference and candidates text by leveraging the embeddings generated by a text embedding model. BERTScore uses these embeddings to compute the cosine similarity between words in order to compute the sentence cosine similarity. SemScore operates in the same manner, however instead of computing the cosine similarity between words in a sentence, it computes the cosine similarity of entire responses.

We set the reference to be the information retrieved by the RAG system, and the candidate to be the synthesized information. We do the same for the synthesized query and original query. Finally, we compare the response of the document-synthesis system to a single-agent system with search function access.

We separate the results into two tables, table 4.3 and table 4.4.

	BLEU	ROUGE-1	ROUGE-2	ROUGE-L
Synthesized Information	0.0913	0.416	0.258	0.353
Synthesized Query	0.396	0.692	0.536	0.660
Response with Synthesized components	0.181	0.526	0.269	0.386

Table 4.3: BLEU and ROUGE scores

Looking at the BLEU and ROUGE scores in table 4.3, we note low BLEU scores, particularly for the synthesized information. This is expected, because as the information undergoes synthesis, details are modified or replaced in order to distance it from the original information retrieved. The low ROUGE scores for the synthesized information is also consistent with the low BLEU score.

In the case of the synthesized query, it has the highest BLEU and ROUGE scores. The synthesized query is designed to closely resemble the original query to ensure that the response the system generates takes into account the synthesized information, even if this results in the synthetic query containing less information than the original query.

For the low BLEU and ROUGE scores for the response, this is likely due to the non-deterministic nature of LLMs. The output of LLMs are not consistent across reruns, furthermore, the manner in which the response is structured will also vary between runs despite access to the same information. As such, it is expected that the response generated by the synthesized information varies from the response with the original information.

	BERT Precision	BERT Recall	BERT F1	SemScore
Synthesized Information	0.3209	-0.01135	0.1459	0.6793
Synthesized Query	0.6065	0.3515	0.4760	0.7698
Response with Synthesized components	0.4290	0.3182	0.3728	0.7938

Table 4.4: BERTScore and SemScore

In table 4.4, we see that the BERT Precision, Recall, and F1 scores vary from poor to moderate. This is to be expected. Since BERTScore computes the sentence similarity using token-level cosine similarity, differences in word choice can result in lower token similarity, resulting in lower scores.

We observe that the recall rate for the synthesized information is negative. This is likely due to the modifications made to the information retrieved, as it means that less of the reference tokens, meaning the original information, is covered by the tokens in the synthetic information.

In all cases we observe that the SemScore performs well. This means that there is a strong to moderate semantic link between the original as well as the synthesized components and responses.

References

[1] Molly Bohannon. *Lawyer used Chatgpt in court-and cited fake cases. A judge is considering sanctions*. Feb. 2024. URL: https://www.forbes.com/sites/mollybohannon/2023/06/08/lawyer-used-chatgpt-in-court-and-cited-fake-cases-a-judge-is-considering-sanctions/ (cit. on p. 2).

[2] Patrick Lewis et al. *Retrieval-Augmented Generation for Knowledge-Intensive NLP Tasks*. 2021. arXiv: 2005.11401 [cs.CL]. URL: https://arxiv.org/abs/2005.11401 (cit. on p. 2).

[3] Mingyu Jin et al. *Health-LLM: Personalized Retrieval-Augmented Disease Prediction System*. 2024. arXiv: 2402.00746 [cs.CL]. URL: https://arxiv.org/abs/2402.00746 (cit. on p. 3).

[4] NUHS. *Ai Healthcare in NUHS receives boost from supercomputer: NUHS+*. Aug. 2023. URL: https://nuhsplus.edu.sg/article/ai-healthcare-in-nuhs-receives-boost-from-supercomputer (cit. on p. 3).

[5] Osmond Chia. *Ai masters Singlish in key breakthrough to serve healthcare and patients' needs*. Nov. 2024. URL: https://www.straitstimes.com/singapore/ai-masters-singlish-in-key-breakthrough-to-serve-healthcare-and-patients-needs (cit. on p. 3).

[6] Liv McMahon. *Google Ai Search tells users to glue pizza and Eat Rocks*. May 2024. URL: https://www.bbc.com/news/articles/cd11gzejgz4o (cit. on p. 4).

[7] Zhen Tan et al. *"Glue pizza and eat rocks" – Exploiting Vulnerabilities in Retrieval-Augmented Generative Models*. 2024. arXiv: 2406.19417 [cs.CR]. URL: https://arxiv.org/abs/2406.19417 (cit. on p. 4).

[8] Jiaqi Xue et al. *BadRAG: Identifying Vulnerabilities in Retrieval Augmented Generation of Large Language Models*. 2024. arXiv: 2406.00083 [cs.CR]. URL: https://arxiv.org/abs/2406.00083 (cit. on p. 4).

[9] Xun Xian et al. *On the Vulnerability of Applying Retrieval-Augmented Generation within Knowledge-Intensive Application Domains*. 2024. arXiv: 2409.17275 [cs.CR]. URL: https://arxiv.org/abs/2409.17275 (cit. on p. 4).

[10] Steve Alder. *Healthcare Data Breach Statistics*. Mar. 2025. URL: https://www.hipaajournal.com/healthcare-data-breach-statistics/ (cit. on p. 4).

[11] Cody Clop and Yannick Teglia. *Backdoored Retrievers for Prompt Injection Attacks on Retrieval Augmented Generation of Large Language Models*. 2024. arXiv: 2410.14479 [cs.CR]. URL: https://arxiv.org/abs/2410.14479 (cit. on p. 5).

[12] Yuefeng Peng et al. *Data Extraction Attacks in Retrieval-Augmented Generation via Backdoors*. 2024. arXiv: 2411.01705 [cs.CR]. URL: https://arxiv.org/abs/2411.01705 (cit. on p. 5).

[13] Shenglai Zeng et al. *The Good and The Bad: Exploring Privacy Issues in Retrieval-Augmented Generation (RAG)*. 2024. arXiv: 2402.16893 [cs.CR]. URL: https://arxiv.org/abs/2402.16893 (cit. on pp. 5, 6, 8).

[14] Zhenting Qi et al. *Follow My Instruction and Spill the Beans: Scalable Data Extraction from Retrieval-Augmented Generation Systems*. 2024. arXiv: 2402.17840 [cs.CL]. URL: https://arxiv.org/abs/2402.17840 (cit. on p. 6).

[15] Xuying Li et al. *Targeting the Core: A Simple and Effective Method to Attack RAG-based Agents via Direct LLM Manipulation*. 2024. arXiv: 2412.04415 [cs.AI]. URL: https://arxiv.org/abs/2412.04415 (cit. on p. 6).

[16] Aryelly Rodriguez et al. "Current recommendations/practices for anonymising data from clinical trials in order to make it available for sharing: A scoping review". In: *Clinical Trials* 19.4 (June 2022), pp. 452–463. DOI: 10.1177/17407745221087469 (cit. on pp. 6, 7).

[17] Corporation MITRE. *Synthea*. 2024. URL: https://synthetichealth.github.io/synthea/ (visited on 11/29/2024) (cit. on p. 9).

[18] Jerry Liu. *LlamaIndex*. Nov. 2022. DOI: 10.5281/zenodo.1234. URL: https://github.com/jerryjliu/llama_index (cit. on p. 10).

[19] Ansar Aynetdinov and Alan Akbik. *SemScore: Automated Evaluation of Instruction-Tuned LLMs based on Semantic Textual Similarity*. 2024. arXiv: 2401.17072 [cs.CL]. URL: https://arxiv.org/abs/2401.17072 (cit. on p. 20).

[20] Xueying Du et al. *Vul-RAG: Enhancing LLM-based Vulnerability Detection via Knowledge-level RAG*. 2024. arXiv: 2406.11147 [cs.SE]. URL: https://arxiv.org/abs/2406.11147.

[21] Yunfan Gao et al. *Retrieval-Augmented Generation for Large Language Models: A Survey*. 2024. arXiv: 2312.10997 [cs.CL]. URL: https://arxiv.org/abs/2312.10997.

[22] John Chong Min Tan et al. *TaskGen: A Task-Based, Memory-Infused Agentic Framework using StrictJSON*. 2024. arXiv: 2407.15734 [cs.AI]. URL: https://arxiv.org/abs/2407.15734.

[23] Meenatchi Sundaram Muthu Selva Annamalai, Andrea Gadotti, and Luc Rocher. *A Linear Reconstruction Approach for Attribute Inference Attacks against Synthetic Data*. 2024. arXiv: 2301.10053 [cs.LG]. URL: https://arxiv.org/abs/2301.10053.

[24] Yi Dong et al. *Building Guardrails for Large Language Models*. 2024. arXiv: 2402.01822 [cs.CL]. URL: https://arxiv.org/abs/2402.01822.

.1 Appendix

Appendix A

Appendix