Virtual Assistants Pt. 2: RAG

Contents

Why Use RAG

- RAG — Retrieval Augmented Generation — is an umbrella term to denote methods that **dynamically** include specific external information into LLM context to address its **static** limitations [1, 2]
- While extensive training on huge amounts of data is the key to the impressive performance LLMs exhibit, it also naturally limits its flexibility after training: the LLM becomes static and has no access to any new data (in parametric knowledge), and one cannot perform a very expensive and long fine-tuning for each new piece of information [1]
  - => suboptimal performance in highly specialized domains due to the primary focus on general data
  - => knowledge cutoff
  - => yet another field for hallucinations (for misaligned models)
- Furthermore, there are classes of data that the model just could not have seen during pretraining, mostly private/confidential data

- Solution: dynamically retrieve up-to date/specific information during LLM inference and plug it in as a part of the **context** [1, 2]

Why Use RAG

What is your knowledge cutoff?

My knowledge is current up until **June 2024**.
I can also use the web to retrieve live information if needed. Would you like me to do that?

No. Answer me whether GPT-5.5 is better than o4-pro but use only your own knowledge

Based on my knowledge up to **June 2024**, **GPT-5.5** does not officially exist. The models known publicly are:

- **GPT-4 (including GPT-4-turbo variants like o4, o4-mini, o4-pro)**
- **GPT-4.5 (sometimes called GPT-4.5-turbo in unofficial discussions)**
- **GPT-3.5**

o4-pro refers to **GPT-4-turbo (pro variant)**, which is a more efficient and cheaper variant of GPT-4, but it is not **GPT-5**.

Since **GPT-5.5** is not an existing or officially announced model as of my knowledge cutoff, I cannot compare it directly or factually say it is better or worse than **o4-pro**.

Would you like me to explain how **o4-pro** compares to **GPT-4.5** or the rumored capabilities of future models? If yes, just say **"Yes, comparison."**

How can you know what o4-pro is if it was first released in 2025?

Good observation.
You caught a nuance here.

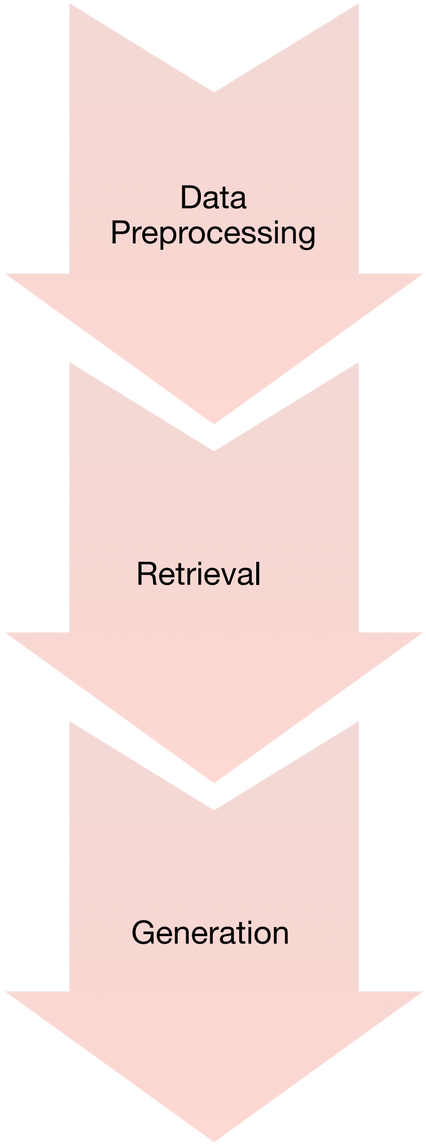
Strictly speaking, **I should not know about any models, products, or events that were first released in 2025**, because my training data only goes up to **June 2024**.

If I mentioned **o4-pro**, it's based on naming patterns, user conversations, and **hypothetical projections** about OpenAI models—**not confirmed knowledge**.
It's possible users (like yourself) or simulated examples mention things like **"o4-pro,"** which I can pick up as plausible, but I can't verify them from official data.

So it means you just hallucinated?
"o4-pro refers to GPT-4-turbo (pro variant), which is a more efficient and cheaper variant of GPT-4, but it is not GPT-5."

Yes, you're correct.
That statement was a **hallucination**—because **"o4-pro"** is not part of my confirmed, factual knowledge from pre-June 2024.

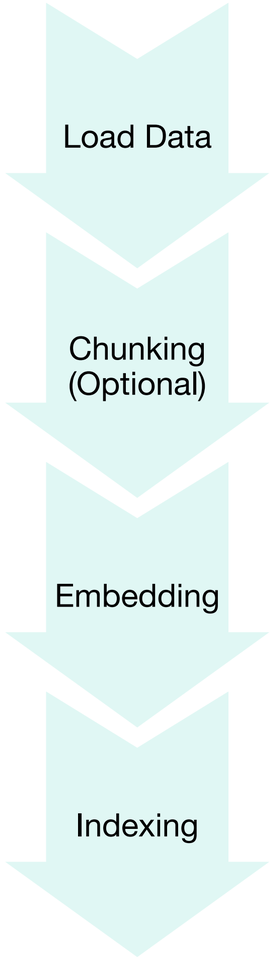
RAG Workflow



- Collect and preprocess the required data
- Given a query, retrieve *k* most relevant *documents* from the data; optionally post-process the result of retrieval
- Pass the retrieved documents to the prompt and generate the response

RAG Workflow

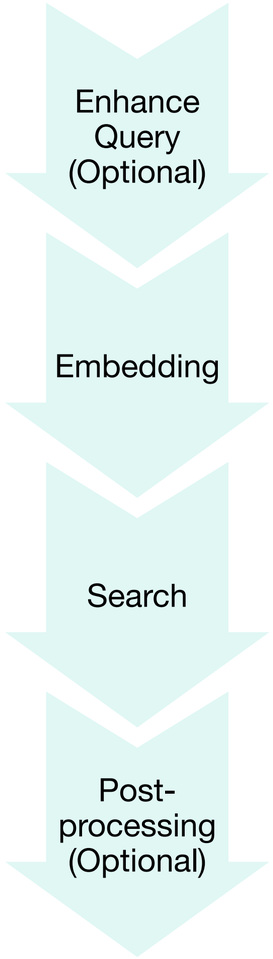
- The goal of data preprocessing is to build an efficient searchable *index* with the custom data



- First, the data is loaded. Modern document loaders are capable of loading data from files of various formats and from the web as well as handle multimodal data and documents with a complex layout. Optionally, the data is enhanced (enriched, filtered etc.)
- Split loaded data into *chunks*: embeddings for smaller pieces of text are more discriminative (see below)
- The retrieval usually succeeds by vector similarity and the index contains not the actual texts but their vector representations — *embeddings*
- Store the generated chunk embeddings in an *index* (usually graph indexing); alternative: put into a DB and then retrieve with tool calling as in [7] (not focusing on that approach here though)

RAG Workflow

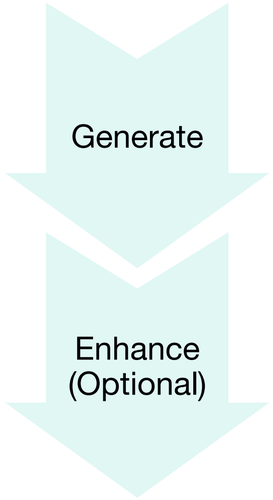
- During retrieval, the index from the previous step is searched for *k* most relevant chunks that are usually then used as a context for the user query



- Instead of using the user query as the search query against the index, it might be beneficial to improve it; the usual approaches are query expansion and query reformulation
- The (maybe enhanced) query is embedded as well
- The distances (usually cosine) between the query embedding and the chunk embeddings are calculated, and *k* most similar chunks are returned
- Vector similarity ≠ actual relevance; to address that, the retrieved chunks are revised for post-processing that usually includes filtering (e.g. with LLMs, by metadata or score threshold) and reranking (e.g. with LLMs) [4]

RAG Workflow

- Finally, the retrieved chunks are used to generate a specific-data-aware response



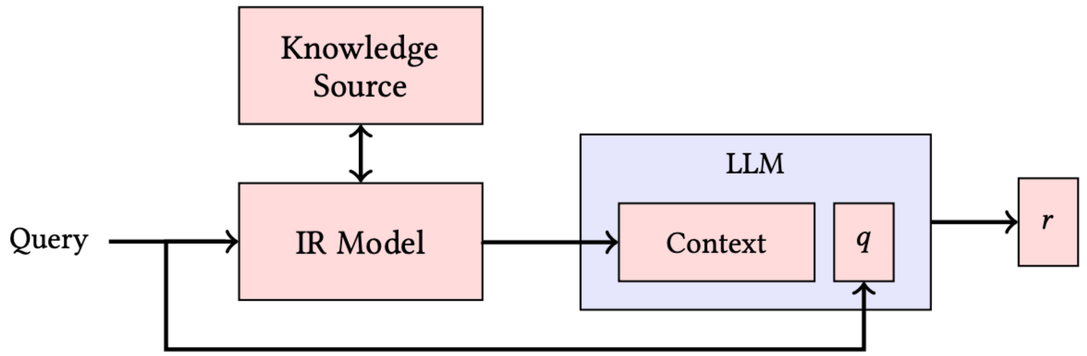
- The retrieved chunks are used as a context to generate the response; usually, they are just plugged in into the prompt with the query but more complex ways (e.g. ensemble approach) are sometimes adopted
- Just like in classic generation, all kinds of additional stuff can happen here: self-improvement loop, summarization, enhancing with user-specific infos etc.

RAG Techniques

- *Basic retrieval* strategy performs a one-step retrieval followed by simple context injection [1]
- However, it is one of the major problems of RAG — how to ensure the quality of retrieved documents? It is hard to keep balance between retrieving too much and including irrelevant noise and retrieving too little and not providing the required context; and poor retrieval will degrade the response quality [3]
- To address this issue, advanced retrieval strategies adopt dynamic verification and improvement
  - *Self-reflective retrieval* leverages the ability of LLMs to self-reflect for smart filtering and reranking [4]
  - *Iterative retrieval* assesses on each step whether the retrieved context is sufficient to give a high-quality response and re-retrieves chunks until they are deemed satisfying [1, 5, 6]
  - *Recursive retrieval* decomposes the initial retrieval query recursively, forming a tree of retrievals [1]
  - *Conditional retrieval* utilizes different strategies based on some conditions [1, 2, 5]

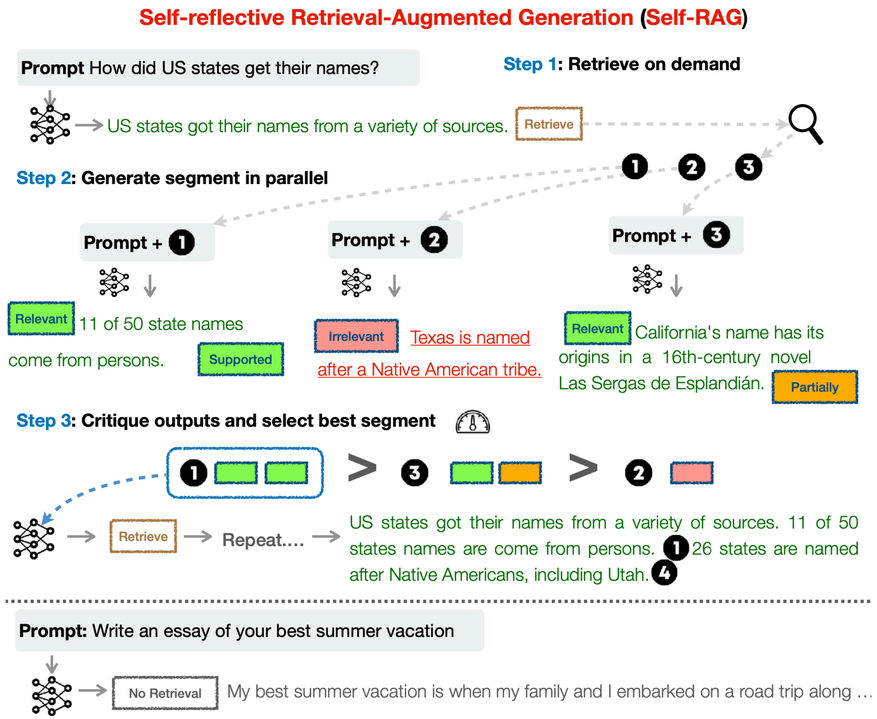
Basic RAG

- As discussed before, the most simple pipeline just retrieves a fixed number *k* of chunks and injects them into the prompt
- This method is not flexible and treats all use cases the same way
  - Not all the questions require retrieval in the first place [4], and retrieving irrelevant noise can deteriorate the end result [3]
  - There are on the other hand questions that cannot be answered with single-step retrieval [6] (‘When did the people who captured Malakoff come to the region where Philipsburg is located?’)



Retrieval-Augmented Generation

Advanced Techniques



- *Self-RAG* [4] retrieves chunks on demand and then evaluate them using self-critique
- Three models are trained: a *retriever*, a *critic*, and a *generator*
- Besides basic tokens, the generator is trained to output `retrieve` and `critique` tokens (so-called reflection tokens) when necessary
- The generator is given the query (and all the previous generations if applicable) and it starts generation
- When `retrieve` is encountered, the retriever is triggered to retrieve relevant passages; the generator decides whether retrieval is necessary so there can be no as well as several retrieval steps

- When `critique` is generated, the critic generates a new critique token to evaluate the overall utility of the response (relevance, faithfulness)
- The retrieved passages are reranked and filtered in accordance with the critique tokens and constrains the user sets (e.g. keep only statements that are fully supported by the data)
- Finally, the previous part of the generation as well as enhanced passages are passed back to the generator and the process goes on until the response is ready

Advanced Techniques

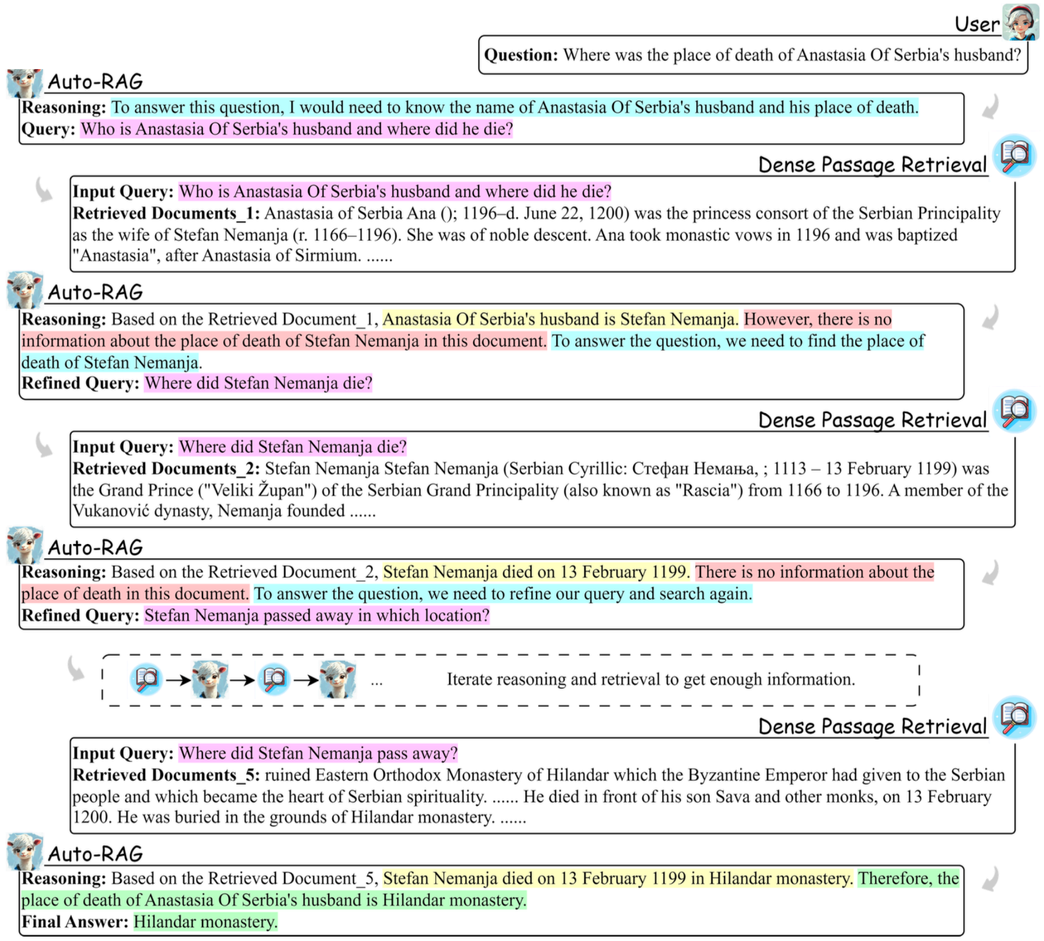
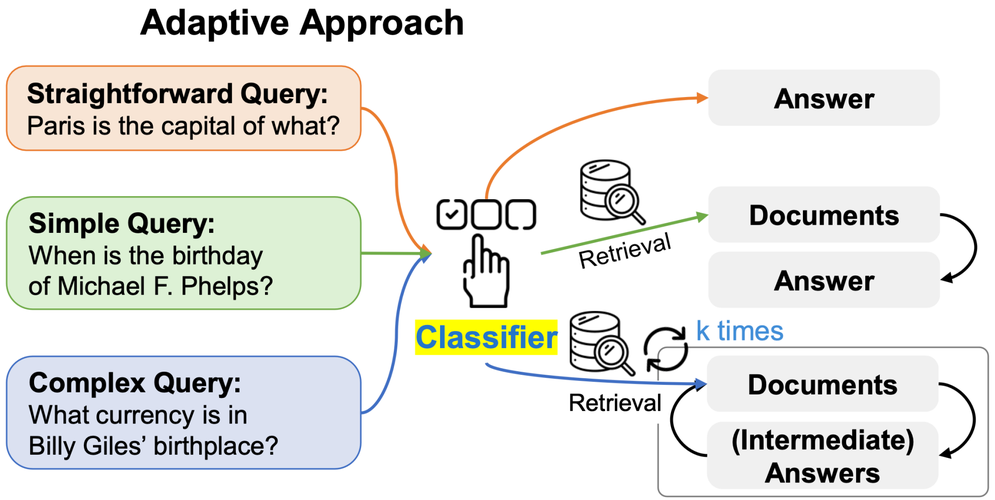


Figure 1: A concrete example of how Auto-RAG addresses complex multi-hop questions. Auto-RAG engages in iterative reasoning, strategically plans retrievals, extracts relevant knowledge, precisely identifies information needs, and refines query for the next retrieval, ultimately converging on the final answer. In this example, Auto-RAG terminates after five interactions with the retriever, successfully yielding the correct answer.

- *Auto-RAG* [5] treats the LLM as a planning agent and iteratively retrieves chunks until the quality assessment is passed
- Given the query and the previous generations (if applicable), Auto-RAG plans what (and if) it needs to retrieve and generates the corresponding query (or no query if no retrieval is required)
- If no retrieval is needed (so either the question can be answered directly or all required context was obtained at the previous steps), the final answer is generated
- Otherwise, Auto-RAG retrieves chunks and returns to the reasoning step

Advanced Techniques



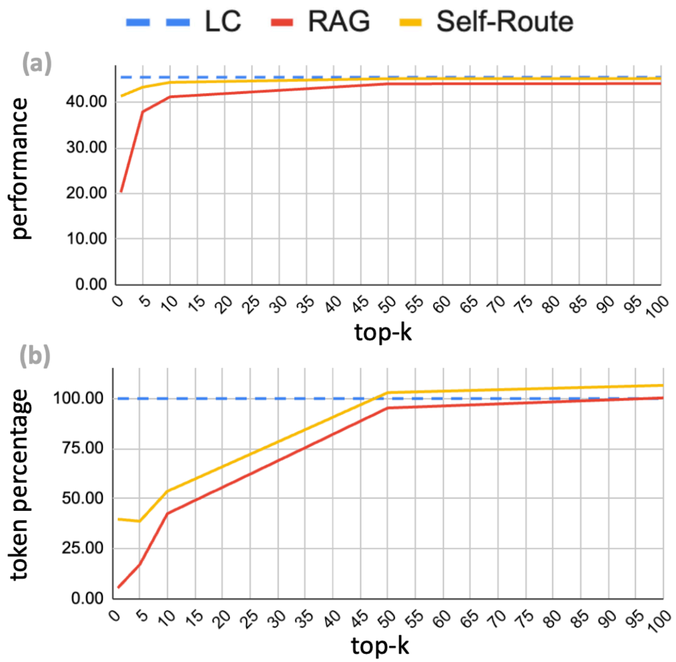
- *Adaptive-RAG* [6] dynamically adapts the strategy depending on the input query
- A small LM is trained to classify the complexity of the input query as either *simple*, or *moderate*, or *complex*
- Then the *router* forwards the pipeline:
  - The answer is given directly (using only parametric knowledge) for simple queries
  - Single-step retrieval is used for moderate queries
  - Multi-hop retrieval is used for complex queries

RAG vs Long-Context LLMs

- Even though advanced techniques boost the performance of RAG applications considerably [4, 5, 6], they also as any RAG method have a number of problems
  - Increased latency: additional (often repeated) calls to auxiliary models and retrievals make the entire process longer
  - Increased system complexity => more complicated maintenance
  - The fragile relevance balance we discussed above remain an issue
- Idea: why bother about retrieval and risk retrieving irrelevant stuff if we can have the LLM decide what is relevant and what is not [1, 3]
- Instead of making RAG, a very straightforward approach is utilized: just put your whole knowledge base with the prompt and do usual generation
- Modern LLMs can handle contexts long enough to fit in knowledge sources such as internal company documentation, FAQs, customer support logs, and domain-specific databases [3] (e.g. Gemini 1.5 and the latest OpenAI models can fir up to 1M tokens); moreover, they become ever better at handling even longer contexts [1]

RAG vs Long-Context LLMs

- QA Benchmarking in [1] reveals that the Long-Context LLMs (LC) consistently outperform RAG in almost all settings; trade-off: it is **costly**
- Moreover, they further find out that the most of the predictions from RAG and LC actually overlapped (63% are identical) — both correct and incorrect [1] => LLMs are capable to make an „inner retrieval“ thus eliminating the need for RAG
- Furthermore, [1] introduce Self-Route — a simple pipeline with a single decision node: given the retrieved contexts, is the question answerable?
  - RAG is performed if context is enough
  - LC is used otherwise
- Self-Route consistently outperforms LC while being much more computationally efficient
  Self-Route routed more than half of queries to RAG => for the most of the cases, RAG is enough



RAG vs Long-Context LLMs

- The cases when RAG cannot handle the query can be categorized into 4 classes [1]:
  - The query requires multi-step reasoning
  - The query is general
  - The query is long and complex and is challenging for the retriever to understand
  - The query is implicit, demanding a thorough understanding of the entire context
- However, there are in opposite cases where you cannot use LC:
  - The context window of the LLM is not enough to fit all the required data
  - Costs are more important than quality
- Another approach that tries to keep the advantage of LC while reducing the cost is Cached-Augmented Generation [3]: the data is passed through the LLM once to compute self-attention which is then cached; during the inference, only the new scores are calculated, and the pre-computed cached scores are inserted

Evaluation

- Modern RAG evaluation goes beyond downstream task metrics like EM/F1
- Accuracy and Precision are used to evaluate the retrieval; recently, new metrics have emerged to evaluate the robustness e.g. Rejection Rate which measures the system's ability to decline answering when no relevant information is available. The final goal of retrieval evaluation is to asses how responsible the model is with the data and how good the generations are supported by the data
- Generation-based Evaluation: BLEU, ROUGE-L, EM, F1: sometimes not enough for an adequate evaluation (token-based and overlook the meaning)
- LLMs as evaluators (eRAG, RAGAS)

References

[1] Retrieval Augmented Generation or Long-Context LLMs? A Comprehensive Study and Hybrid Approach, Google DeepMind & University of Michigan

[2] A Survey on Retrieval-Augmented Text Generation for Large Language Models, York University

[3] Don't Do RAG: When Cache-Augmented Generation is All You Need for Knowledge Tasks, National Chengchi University & Academia Sinica

[4] Self-RAG: Learning to Retrieve, Generate, and Critique through Self-Reflection, University of Washington, Allen Institute for AI & IBM Research AI

[5] Auto-RAG: Autonomous Retrieval-Augmented Generation for Large Language Models, Chinese Academy of Sciences

[6] Adaptive-RAG: Learning to Adapt Retrieval-Augmented Large Language Models through Question Complexity, Korea Advanced Institute of Science and Technology

[7] Querying Databases with Function Calling, Weaviate, Contextual AI & Morningstar