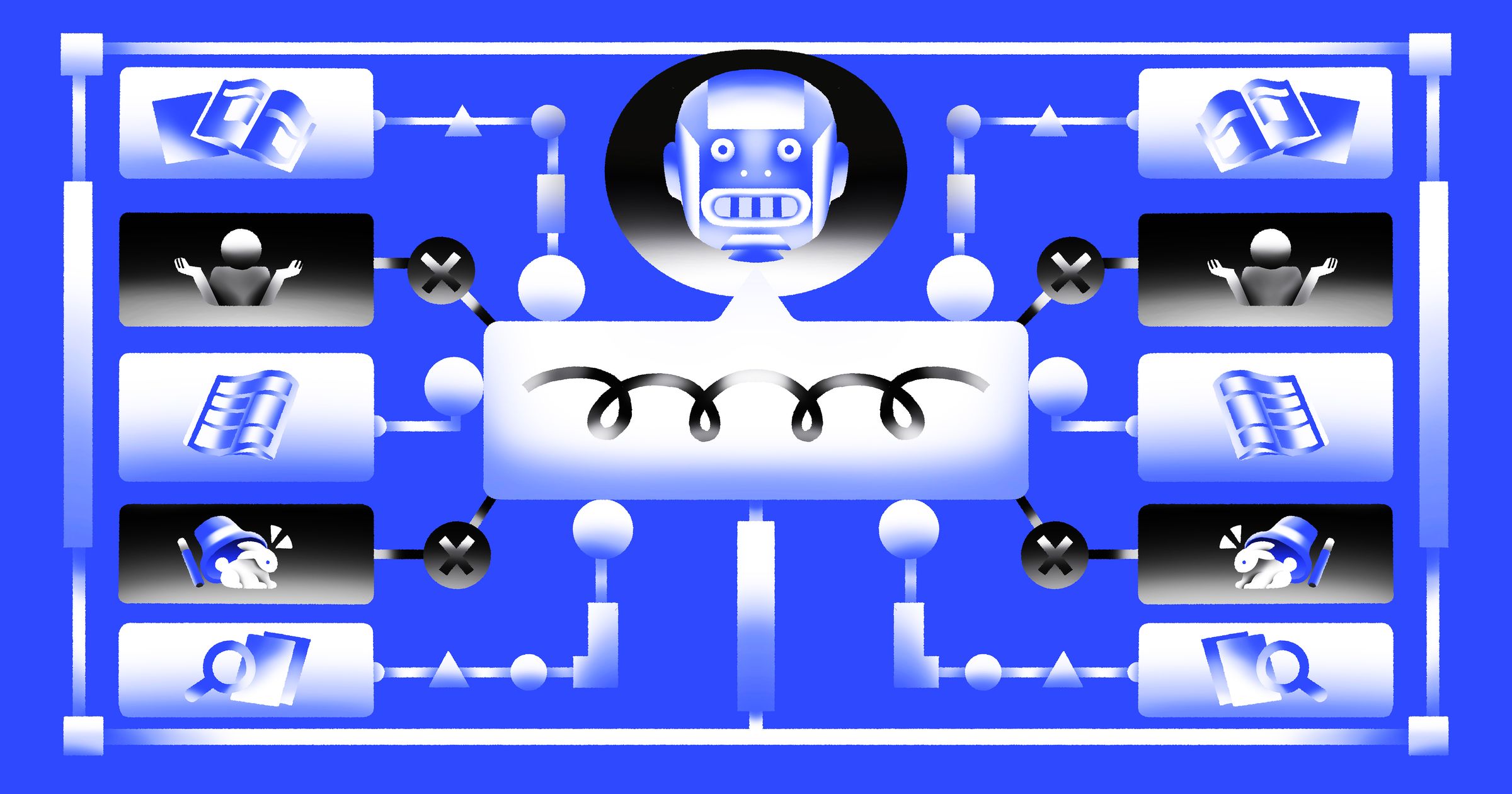
**Retrieval augmented generation: Keeping LLMs relevant and current**

Retrieval augmented generation (RAG) is a strategy that helps address both LLM hallucinations and out-of-date training data.

 Credit: Alexandra Francis

Generative AI technologies are powerful, but they're limited by what they know. While an LLM like ChatGPT can perform many tasks, every LLM's baseline knowledge has gaps based on its training data. If you ask an LLM to write something about a recent trend or event, the LLM won't have any idea what you're talking about, and the responses will be mixed at best and problematic at worst.

LLMs' knowledge-based problems come down to two key issues:

* LLM training data tends to be hopelessly out-of-date (as of writing, ChatGPT's knowledge of the world ends at January 2022, excluding functionality like Browse with Bing and GPT-4V that provide additional context).
* LLMs extrapolate when facts aren’t available, so they confidently make false but plausible-sounding statements when there's a gap in their knowledge (called hallucination).

Retrieval augmented generation (RAG) is a strategy that helps address both of these issues, pairing information retrieval with a set of carefully designed system prompts to anchor LLMs on precise, up-to-date, and pertinent information retrieved from an external knowledge store. Prompting LLMs with this contextual knowledge makes it possible to create domain-specific applications that require a deep and evolving understanding of facts, despite LLM training data remaining static.

You could ask an LLM, "What is RAG?", and you might get very different responses depending on whether or not the LLM itself uses RAG:

**“What is RAG?” response with RAG**

RAG, or retrieval augmented generation, is a method introduced by Meta AI researchers that combines an information retrieval component with a text generator model to address knowledge-intensive tasks. Its internal knowledge can be modified efficiently without needing to retrain the entire model. Would you like more details on this?

**“What is RAG?” response without RAG**

RAG can refer to several different things depending on the context. Here are a few common meanings:

1. Red, Amber, Green (RAG): In project management and reporting, RAG is a color-coding system used to quickly convey the status or health of a project or task.

2. RAG Analysis: This is a method used in risk assessment or decision-making. It involves categorizing risks or options as Red, Amber, or Green based on their level of severity, impact, or desirability. It helps prioritize actions or choices.

3. Random Access Generator (RAG): In some technical contexts, RAG might refer to a system or algorithm that generates random access patterns or data, often used in computer science or information retrieval.

4. Resource Allocation Graph (RAG): In the context of operating systems and computer science, a Resource Allocation Graph is used for deadlock detection. It's a directed graph that represents resource allocation and request relationships among processes in a system.

If you have a specific context or domain in mind, please provide more details, and I can give you a more precise explanation of "RAG" in that context.

There's a big difference between answering this question with and without RAG. Especially in cases involving domain-specific knowledge (like acronyms), RAG can drastically improve the accuracy of an LLM's responses.

If you've interacted with a chatbot that knows about recent events, is aware of user-specific information, or has a deeper understanding of a subject than is normal, you've likely interacted with RAG without realizing it. If you've seen tutorials about "chatting over your documents," that's the simplest version of RAG around. Frameworks like LangChain and LlamaIndex have democratized RAG by making it possible to create simple knowledge-aware applications quickly.

I've developed and implemented LLM applications internal to Skyflow, including systems that use RAG. With RAG being used nearly everywhere—and not leaving anytime soon—it's important to understand both the basics of RAG and how to move beyond those basics when you want to move your code into production. I invite you to learn from my experience on this RAG journey so you don't have to learn the hard way.

**An overly simplified example**

LangChain has an [example](https://python.langchain.com/docs/use_cases/question_answering/#quickstart) of RAG in its smallest (but not simplest) form:

from langchain.document\_loaders import WebBaseLoader

from langchain.indexes import VectorstoreIndexCreator

loader = WebBaseLoader("https://www.promptingguide.ai/techniques/rag")

index = VectorstoreIndexCreator().from\_loaders([loader])

index.query("What is RAG?")

With these five lines, we get a description of RAG, but the code is heavily abstracted, so it's difficult to understand what's actually happening: We fetch the contents of a web page (our knowledge base for this example).

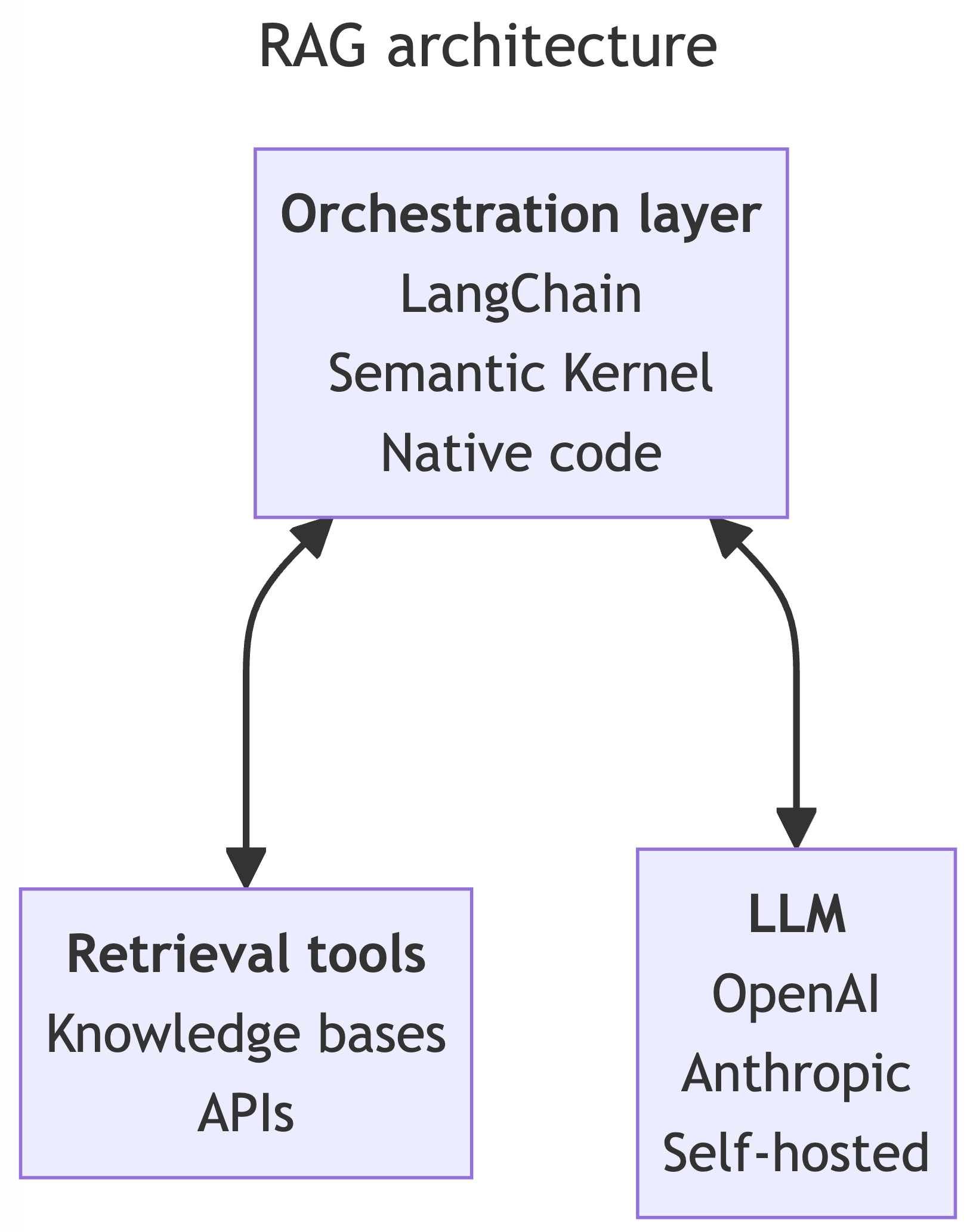
1. We process the source contents and store them in a knowledge base (in this case, a vector database).
2. We input a prompt, LangChain finds bits of information from the knowledge base, and passes both prompt and knowledge base results to the LLM.

While this script is helpful for prototyping and understanding the main beats of using RAG, it's not all that useful for moving beyond that stage because you don't have much control. Let's discuss what *actually* goes into implementation.

**Basic architecture**

Because a [full LLM application architecture](https://a16z.com/emerging-architectures-for-llm-applications/) would be fairly large, we're going to consider just the components that enable RAG:

* The **orchestration layer** receives the user's input in any associated metadata (like conversation history), interacts with all of the related tooling, ships the prompt off to the LLM, and returns the result. Orchestration layers are typically composed of tools like LangChain, Semantic Kernel, and others with some native code (often Python) knitting it all together.
* **Retrieval tools** are a group of utilities that return context that informs and grounds responses to the user prompt. This group encompasses both knowledge bases and API-based retrieval systems.
* **LLM** is the large language model that you're sending prompts to. They might be hosted by a third party like OpenAI or run internally in your own infrastructure. For the purposes of this article, the exact model you're using doesn't matter.



In a typical LLM application, your inference processing script connects to retrieval tools as necessary. If you're building an LLM [agent](https://python.langchain.com/docs/modules/agents.html)-based application, each retrieval utility is exposed to your agent as a tool. From here on, we'll only discuss typical script-based usage.

When users trigger your inference flow, the orchestration layer knits together the necessary tools and LLMs to gather context from your retrieval tools and generate contextually relevant, informed responses. The orchestration layer handles all your API calls and RAG-specific prompting strategies (which we'll touch on shortly). It also performs validations, like making sure you don't go over your LLM's token limit, which could cause the LLM to reject your request because you stuffed too much text into your prompt.

A diagram of RAG processes:
User
1. Prompt enters system
to Orchestration layer
2. Get context for RAG 
to retrieval tools and back
3. Perform inference
to LLM and back
4. Return response
to User

**Knowledge base retrieval**

To query your data, you not only need your data, you need it in a format that's accessible to your application. For LLM-based applications, this usually involves a vector store—a database that can query based on textual similarity rather than exact matches.

Getting your data from its source format into a vector store requires an ETL (extract, transform, load) pipeline.



1. **Aggregate source documents.** Anything you want available to your application, you need to collect. For my work on Skyflow’s private LLM, this included our product documentation, white papers, and blog posts, but this could easily extend to internal records, planning documents, etc.
2. **Clean the document content.** If there's anything that shouldn't be visible to the LLM provider or to the end users of your application, now's your opportunity to remove it. At this stage in the process, remove personally identifiable information (PII), confidential information, and in-development content. Anything that remains after this step will be visible to all stages of the training and inference processes that follow. [Skyflow GPT Privacy Vault](https://www.skyflow.com/product/llm-privacy-vault) can help you de-identify document contents to keep your training data clear of sensitive information.
3. **Load document contents into memory.** Tools like [Unstructured](https://unstructured.io/), [LlamaIndex](https://www.llamaindex.ai/), and LangChain's [Document loaders](https://python.langchain.com/docs/integrations/document_loaders/) make it possible to load all sorts of document types into your applications, particularly unstructured content. Whether it's a text document, a spreadsheet, a web page, a PDF, a Git repo, or hundreds of other things, there's likely a loader for it. Be careful, though, because not all loaders are created equal, and some load more content or context than others. For example, one loader might load content from multiple sheets in a spreadsheet, while another loads content from only the first sheet.
4. **Split the content into chunks.** When you *split* your content, you break it into small, bite-sized pieces that can fit into an LLM prompt while maintaining meaning. There are multiple ways to split your content. Both LangChain and LlamaIndex have text splitters available, with defaults that split recursively by whitespace characters and by sentence, but you need to use whatever works best for you and your content. For my project, I found that my documentation, written in Markdown, lost too much context even with LangChain's Markdown splitter, so I wrote my own splitter that chunked content based on Markdown's heading and code block markup.
5. **Create embeddings for text chunks.** Embeddings store vectors—numerical representations of one text chunk's relative position and relationship to other nearby text chunks. While that's difficult to visualize, creating embeddings is thankfully easy. OpenAI offers an embeddings model, LangChain and LlamaIndex offer a variety of hosted or self-hosted embedding options, or you can do it yourself with an embedding model like [SentenceTransformers](https://www.sbert.net/).
6. **Store embeddings in a vector store.** Once you have your embeddings, you can add them to a [vector store](https://stackoverflow.blog/2023/10/09/from-prototype-to-production-vector-databases-in-generative-ai-applications/) such as Pinecone, Weaviate, FAISS, Chroma or a multitude of other options.

Once the vectors are stored, you can query the vector store and find the content that's most similar to your query. If you need to update or add to your source documents, most vector stores allow updating the store. This means you can also *remove* content from your vector store, something you can't do when you fine-tune a model.

You can run the pipeline again to recreate the whole knowledge base, and while that would be less expensive than re-training a model, it would still prove time-consuming and inefficient. If you expect to update your source documents regularly, consider creating a document indexing process so you only process the new and recently updated documents in your pipeline.

**API-based retrieval**

Retrieval from a vector store isn't the only kind of retrieval. If you have any data sources that allow programmatic access (customer records databases, an internal ticketing system, etc.), consider making them accessible to your orchestration layer. At run time, your orchestration layer can query your API-based retrieval systems to provide additional context pertinent to your current request.

**Prompting with RAG**

After your retrieval tools are set up, it's time for a bit of orchestration-layer magic to knit it all together.

First, we'll start with the prompt template. A prompt template includes placeholders for all the information you want to pass to the LLM as part of the prompt. The system (or base) prompt tells the LLM how to behave and how to process the user's request. A simple prompt template might look like this:

System: You are a friendly chatbot assistant that responds in a conversational

manner to users' questions. Respond in 1-2 complete sentences, unless specifically

asked by the user to elaborate on something. Use History and Context to inform your answers.

---

History: {history}

---

Context: {context}

---

User: {request}

When your end user submits a request, you get to start filling in the variables. If the user submitted "What is RAG?", you could fill in the *request* variable. You'd likely get the conversation history along with the request, so you could fill in that variable, too:

System: You are a friendly chatbot assistant that responds in a conversational

manner to user's questions. Respond in short but complete answers unless specifically

asked by the user to elaborate on something. Use History and Context to inform your answers.

---

History: [{"role": "assistant", "message": "Hi! How can I help you?"}]

---

Context: {context}

---

User: What is RAG?

Next, call your retrieval tools, whether they're vector stores or other APIs. Once you have your context (knowledge base results, customer records, etc.), you can update the *context* variable to inform and ground the LLM during inference.

**Note:**If you have multiple kinds of data included in your RAG implementation, make sure to label them to help the LLM differentiate between them.

System: You are a friendly chatbot assistant that responds in a conversational

manner to user's questions. Respond in short but complete answers unless specifically

asked by the user to elaborate on something. Use History and Context to inform your answers.

---

History: [{"role": "assistant", "message": "Hi! How can I help you?"}]

---

Context: [Document(page\_content='Meta AI researchers introduced a method called [Retrieval Augmented Generation](https://ai.facebook.com/blog/retrieval-augmented-generation-streamlining-the-creation-of-intelligent-natural-language-processing-models/) (RAG) to address such knowledge-intensive tasks. RAG combines an information retrieval component with a text generator model. RAG can be fine-tuned and its internal knowledge can be modified in an efficient manner and without needing retraining of the entire model.', metadata={'source': 'https://www.promptingguide.ai/techniques/rag', 'title': 'Retrieval Augmented Generation (RAG)'})]

---

User: What is RAG?

With your context in place, your prompt template is filled out, but you still have two post-processing tasks to complete:

* Much like you cleaned your source data, you need to clean your prompt as well. Users can input PII in their request even when your data is clean, and you don't want your users' information ending up in an LLM operator's logs or data stores. This also serves as a safeguard in case you *didn't* clean your source data. In addition to de-identifying data sets and prompts, [Skyflow GPT Privacy Vault](https://www.skyflow.com/product/llm-privacy-vault) can also re-identify sensitive values in LLM responses before you send responses to users.
* Make sure you don't exceed your LLM's token limits. If you do, your inference attempts will fail. Before you attempt inference, use token calculators (like [tiktoken](https://github.com/openai/tiktoken)), to make sure you're within LLM-specific limits. You may have to reduce the amount of context you include to fit within your token limit.

Once your prompt is cleaned and within token limits, you can finally perform inference by sending the prompt to your LLM of choice. When you get the response, it should be informed by the context you provided, and you can send it back to the user.

**Improving performance**

Now that you have RAG working in your application, you may want to move to production immediately. Don't do it yet! Alternatively, you may not be all that thrilled with the results you have. Never fear, here are a handful of things you can do to improve your RAG performance and get production-ready:

* **Garbage in, garbage out.** The higher the quality of the context you provide, the higher quality result you'll receive. Clean up your source data to make sure your data pipeline is maintaining adequate content (like capturing spreadsheet column headers), and make sure unnecessary markup is removed so it doesn't interfere with the LLM's understanding of the text.
* **Tune your splitting strategy**. Experiment with different text chunk sizes to make sure your RAG-enabled inference maintains adequate context. Each data set is different. Create differently split vector stores and see which one performs best with your architecture.
* **Tune your system prompt.** If the LLM isn't paying enough attention to your context, update your system prompt with expectations of how to process and use the provided information.
* **Filter your vector store results.** If there are particular kinds of content you do or don't want to return, filter your vector store results against metadata element values. For example, if you want a process, you might filter against a docType metadata value to make sure your results are from a how-to document.
* **Try different embedding models (and fine-tune your own).** Different embedding models have different ways of encoding and comparing vectors of your data. Experiment to see which one performs best for your application. You can check out the current best-performing open source embedding models at the [MTEB leaderboard](https://huggingface.co/spaces/mteb/leaderboard). If you're adventurous, you can also fine-tune your own embedding models so your LLM becomes more aware of domain-specific terminology, and therefore gives you better query results. And yes, you can absolutely use your cleaned and processed knowledge base dataset to fine-tune your model.

**What about fine-tuning?**

Before we wrap up, let's address another LLM-related buzzword: fine-tuning.

Fine-tuning and RAG provide two different ways to optimize LLMs. While RAG and fine-tuning both leverage source data, they each have unique advantages and challenges to consider.

Fine-tuning is a process that continues training a model on additional data to make it perform better on the specific tasks and domains that the data set details. However, a single model can't be the best at everything, and tasks unrelated to the fine-tuned tasks often degrade in performance with additional training. For example, fine-tuning is how code-specific LLMs are created. Google's Codey is fine-tuned on a curated dataset of code examples in a variety of languages. This makes Codey perform substantially better at coding tasks than Google's Duet or PaLM 2 models, but at the expense of general chat performance.

Conversely, RAG augments LLMs with relevant and current information retrieved from external knowledge bases. This dynamic augmentation lets LLMs overcome the limitations of static knowledge and generate responses that are more informed, accurate, and contextually relevant. However, the integration of external knowledge introduces increased computational complexity, latency, and prompt complexity, potentially leading to longer inference times, higher resource utilization, and longer development cycles.

RAG does best fine-tuning in one specific area: forgetting. When you fine-tune a model, the training data becomes a part of the model itself. You can't isolate or remove a specific part of the model. LLMs can't forget. However, vector stores let you add, update, and delete their contents, so you can easily remove erroneous or out-of-date information whenever you want.

Fine-tuning and RAG together can create LLM-powered applications that are specialized to specific tasks or domains and are capable of using contextual knowledge. Consider GitHub Copilot: It's a fine-tuned model that specializes in coding and uses your code and coding environment as a knowledge base to provide context to your prompts. As a result, Copilot tends to be adaptable, accurate, and relevant to your code. On the flip side, Copilot's chat-based responses can anecdotally take longer to process than other models, and Copilot can't assist with tasks that aren't coding-related.

**Wrap-up**

RAG represents a practical solution to enhance the capabilities of LLMs. By integrating real-time, external knowledge into LLM responses, RAG addresses the challenge of static training data, making sure that the information provided remains current and contextually relevant.

Going forward, integrating RAG into various applications holds the potential to significantly improve user experiences and information accuracy. In a world where staying up-to-date is crucial, RAG offers a reliable means to keep LLMs informed and effective. Embracing RAG's utility enables us to navigate the complexities of modern AI applications with confidence and precision.