LLM Retrieval-Augmented Generation (RAG) with OpenVINO™ and LangChain

1. Executive Summary

Retrieval-Augmented Generation (RAG) is one of the most efficient and inexpensive ways for companies to create their own AI applications around Large Language Models (LLMs). It allows LLMs to augment their knowledge with an additional information source specific to a certain domain. RAG extends the capability of the LLM and improves response quality without needing to go through the time-consuming process of fine-tuning.

Companies that want to deploy an AI application (such as a support chatbot) can use OpenVINO™ and LangChain to implement an efficient RAG pipeline. OpenVINO™ provides the following benefits:

- Best-in-class performance for served LLMs with Intel® Core™ and Intel® Xeon® processors
- Support for a variety of LLM architectures from a wide range of frameworks
- Weight compression and optimization with NNCF
- Pre-converted and optimized models available

This white paper gives more information about RAG pipelines, why they are useful, and how they work. It shows code examples for each stage of the pipeline and includes links to end-to-end examples showing how to deploy chatbot applications locally or using OpenVINO™ Model Server.

2. Introduction to Retrieval-Augmented Generation

Large Language Models (LLMs) are deep learning neural networks that have been trained on large text datasets scraped from the Internet and other sources. They are being used in an increasing number of AI-based products and services, such as chat support agents, virtual assistants, code generators, and much more. However, there are two major drawbacks with LLMs:

1. They only have knowledge of topics up to a certain date. For example, ChatGPT-4 does not have knowledge of events from April 2023 onward.
2. The models "hallucinate" in their response, and confidently give wrong answers on topics they don't know about.
3. Publicly available models have no access to private data and are unable to generate any content about proprietary information.

Businesses seeking to deploy their own LLM can overcome these limitations by fine-tuning models on a custom dataset. However, the process of curating a dataset for fine-tuning takes significant effort and time.

Retrieval-Augmented Generation (RAG) is a method for augmenting a LLM's knowledge with specific data beyond the general dataset it was trained on. It increases the quality and accuracy of LLMs without needing to fine-tune a custom model. With RAG, the LLM is provided with a set of documents to retrieve information from, and then it answers user queries with the information from the documents. This way, the model has immediate access to the necessary information and can use it to generate an accurate response.

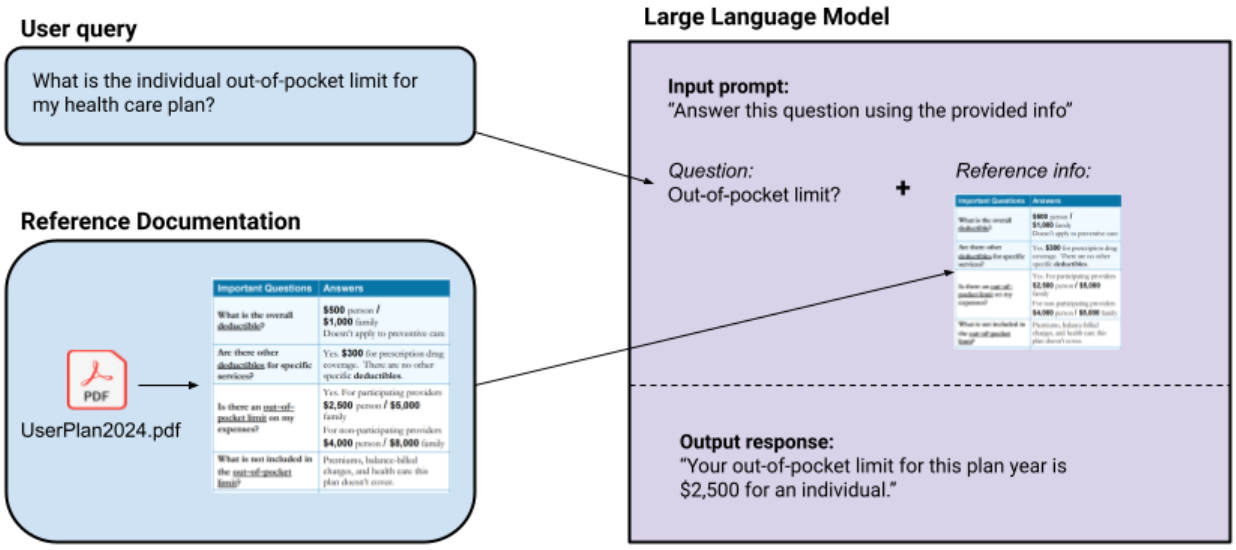

Figure 1. A basic RAG example where reference information is used to help a user's question.

There are several benefits to using RAG:

1. RAG increases the accuracy of LLM responses, because the LLM can directly reference the set of information provided rather than relying on its general knowledge.
2. It significantly reduces the likelihood of hallucination and incorrect responses. If the provided documentation does not have the information the user is looking for, the LLM can simply say it doesn't know the answer to the user's query.
3. The LLM's knowledge source can be updated in real time so it can stay current with changes in information.
4. RAG responses are more transparent because they can include references to the source of the information.

RAG allows companies to develop AI-enabled chatbots with specific knowledge of their product or service without needing to go through the costly and time-consuming process of fine tuning. For example, a health insurance company can provide an LLM with information about its health care plans to give it knowledge of their policies and rates. Then, when a customer asks about premiums and coverage for a certain plan, the RAG-empowered chat agent can respond with detailed and accurate information. If portions of the plan change, the company can immediately upload the changed documentation so the chatbot stays up-to-date with current knowledge.

This white paper explains how a RAG pipeline works, shows how to set one up with OpenVINO™ and LangChain, and links to comprehensive end-to-end examples for deploying pipelines locally or on a server.

3. Explanation of RAG Pipeline

3.1. RAG Pipeline

A typical RAG pipeline consists of several stages and may use more than one deep learning model in the question-answering process. An example full pipeline is shown below.

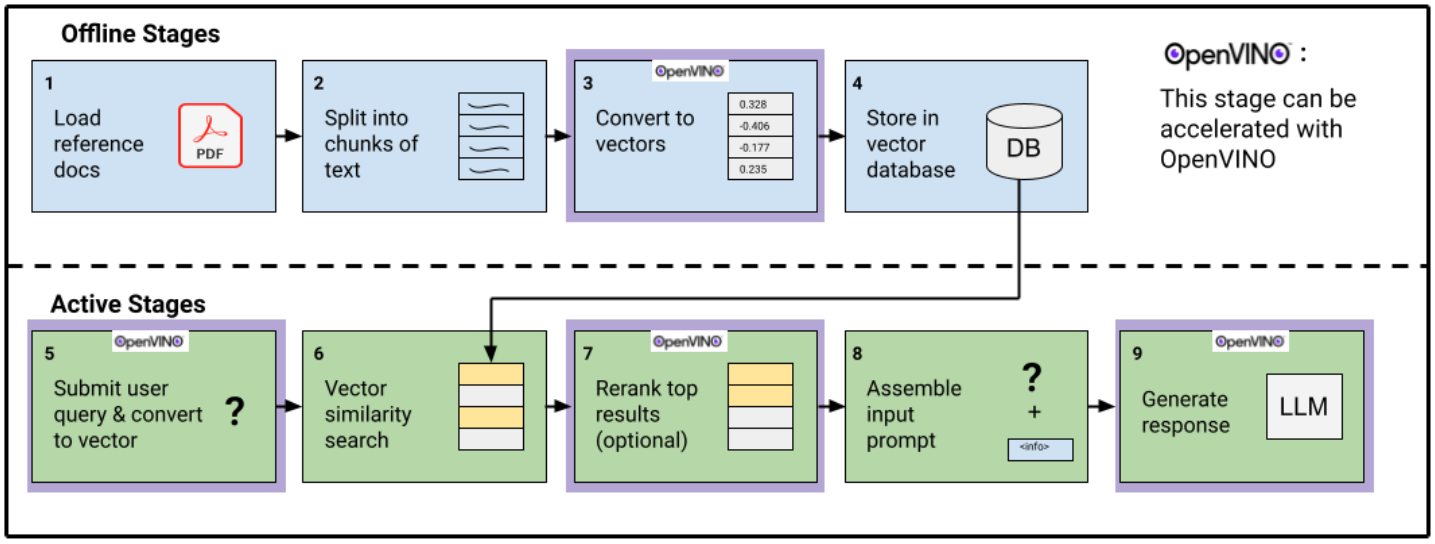

Figure 2. The RAG pipeline consists of preparation stages that occur offline (before deployment) and active stages that occur when the user is interacting with the application.

The first four stages occur offline, before the model is deployed:

1. The source documents are loaded using an unstructured loader (which supports .txt, .pdf, .html, etc).
2. The documents are split into chunks of text, making them easier to parse. The chunks of text have to be short enough to fit in the LLM's context window and must be small enough to be accurately representable by a single embedding.
3. These text chunks are converted to vectors that numerically represent the information in the text using an embedding model.
4. The vectors are stored in an indexed database which can easily be searched through.

When the application is active and a user submits an input prompt:

5. The user's text input prompt is converted to a vector using the same embedding model.
6. The query vectors are mathematically compared to the document vectors to determine which portions of documentation are most relevant to the query, returning the "top k" matches.
7. (Optional) A cross-encoder based rerank model is used to sort the top k document matches. Rerank models use transformer networks to determine the similarity of the

query to the document chunks, so they may provide a better order of relevance to the query.
8. The relevant chunks of documentation are added as context to the original user input and submitted to the LLM.
9. The LLM returns an answer based on the user input and the context of the documentation it was provided.

By following this process, the RAG pipeline allows the LLM to provide an answer based on the relevant documentation, which will be more accurate and less prone to hallucinations.

Three different deep learning models are used in the RAG pipeline: an embedding model, a rerank model, and a large language model. Each of these can be optimized and compressed using OpenVINO™ for better system performance. The sections below provide further explanation of each model.

3.2. Text Embedding Models

A fundamental part of RAG operation is text embedding. This is the process of converting a sequence of words (text) into a sequence of vectors (numbers) that represent the information contained in the words. Embeddings make it so deep learning models and other algorithms can numerically calculate the relationships between pieces of text, allowing them to perform tasks such as clustering or retrieval. In the case of RAG, embeddings are used to compare the user input to the stored documents so the pipeline can retrieve the pieces of text that are similar to the user query. It occurs at the “Embedding” stages in the pipeline diagram shown in Section 3.1.

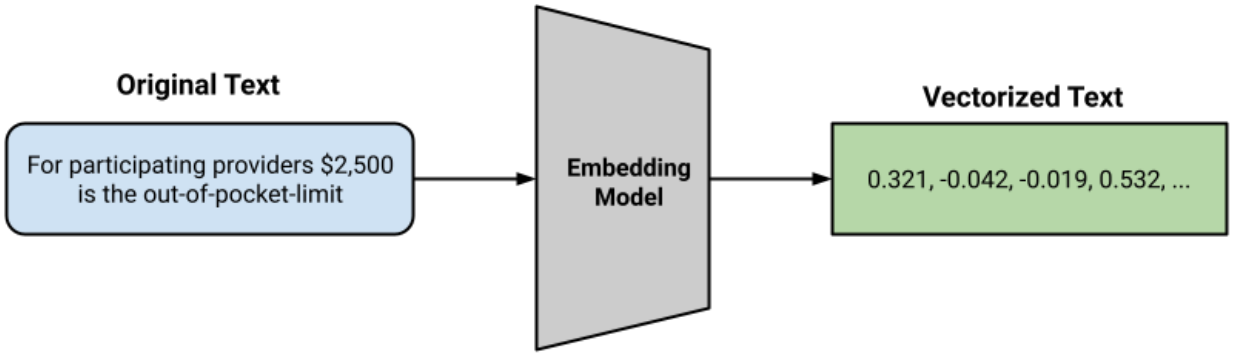

Figure 3. Embedding models convert text into high-dimensional numerical vectors.

Converting a series of words into a series of vectors that contain the semantic meaning of the words is not a straightforward task. It requires the use of embedding models, which are deep learning models trained to efficiently encode words into high-dimensional vectors. A list of popular embedding models can be found at the Massive Text Embedding Benchmark Leaderboard.

OpenVINO™ supports a variety of sentence-transformers based embedding models. One supported model is the BGE Embedding text embedding model, which is a popular model with good performance. There are small and large models available in English and Chinese.

Table 1. List of BGE Text Embedding Model Options

Model	Language	Parameters	MTEB Score
bge-small-en-v1.5	English	33.4M	62.17
bge-small-zh-v1.5	Chinese	24M	57.82
bge-large-en-v1.5	English	335M	64.23
bge-large-zh-v1.5	Chinese	250M	64.53

The main difference between the small and large models is the number of parameters. The large models have better scores on MTEB [reference], a benchmark that tests average performance across eight embedding tasks. The benchmark measures how well the vector conversion retains the information from the text.

The large models require more memory and processing power, but the resulting vectors will contain more semantic meaning. Ultimately, the larger model helps the RAG pipeline find better documentation matches to the user query. However, the small model is likely accurate enough for most use cases. Developers can use the OpenVINO™ RAG Notebook (see Section 5.1) to test both models in a full pipeline and determine which works best for their use case.

Other popular embedding models include:

- mxbai-embed-large-v1
- UAE-Large-V1 and UAE-Code-Large-V1
- gte-base-en-v1.5

3.3. Rerank Models and Two-Stage Retrieval

Rerankers compare an input query against a set of documents and rank the documents in order of relevance to the query. (This all occurs in embedded vector space.) Rerankers are part of a two-stage retrieval system. Before deployment, a full dataset of documents is encoded into vectors using a text embedding model.

In the first stage, the query is converted to a vector, and then mathematically compared to the document vectors (using "cosine similarity" or a similar metric) to determine which document chunks have data relevant to the query. The first stage will return the "top_k" documents (e.g. top_k = 25).

In the second stage, a rerank model, which is more accurate than the mathematical comparison, is used to rank the top_k documents in order of relevance to the query and return the “top_n” best documents (e.g. top_n = 5). Rerankers benefit from having the context of the user query to better locate relevant information. They also require more processing than the mathematical comparison used in the first stage.

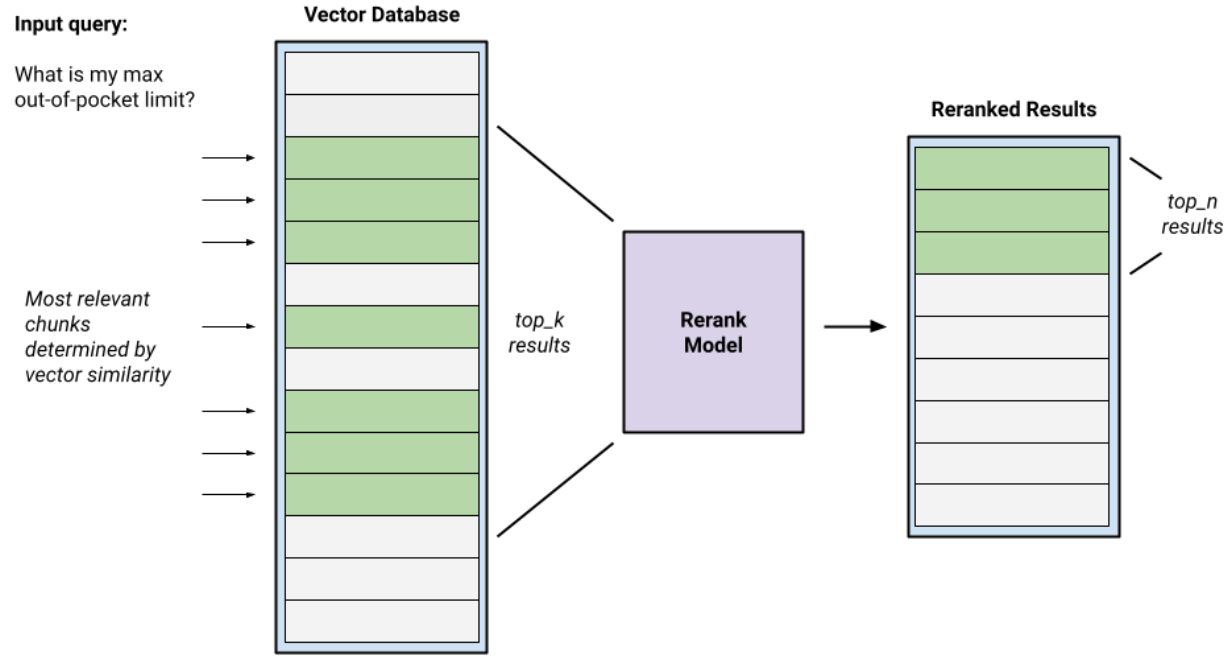

Figure 4. In two-stage retrieval, rerank models take the top results identified by vector similarity and re-sort them in order of relevance to the input query

Here’s how two-stage retrieval works in the RAG pipeline:

- Chunks of a database of text is converted to vectors using a text embedding model
- A user query is converted to a vector
- The query vector is compared to the database vectors to identify the top_k most relevant chunks of text in the database
- A reranker model sorts those top_k chunks in better order of relevance to the user query
- Now, the most relevant chunks of information can be passed to an LLM along with the user query.

The BGE Reranker models are a popular family of models that are supported by OpenVINO™. The models are multilingual, and there is a large and small model available. OpenVINO™ also supports other rerank models that can be found on HuggingFace Hub.

Table 2. List of BGE Rerank Model Options

Model	Language	Parameters	MTEB Rerank Score
bge-reranker-base	Multilingual	278M	65.42
bge-reranker-large	Multilingual	560M	66.10

Similar to the embedding models, the main difference between the base and large BGE-reranker models is the number of parameters. The larger model requires more memory, but will do a better job of ranking documentation chunks by their relevance to the user query. Again, developers can use the OpenVINO™ RAG Notebook (see Section 5.1) to test both models in a full pipeline and determine which works best for their use case.

3.4. Large Language Models

Large language models (LLMs) are massive neural networks that excel in various language-related tasks like text completion, question-answering, and content summarization. LLMs are trained from an extensive repository of text data, which is usually collected by crawling web pages across the internet. LLMs are trained for a simple task: given a sequence of words, predict the next word in the sequence.

With RAG, LLM inference occurs at the last stage of the pipeline. The pipeline assembles a new prompt that combines the user query along with the relevant documentation. The new prompt may be arranged as:

```
"Answer this question based only on the following context:
<relevant documentation here>

Question: <user query here>”
```

This allows the LLM to reference the provided chunks of documentation when generating a response, thereby increasing the accuracy of the response and reducing the likelihood of false information.

Most LLMs available from HuggingFace Hub can be used in the RAG pipeline. The main design parameter is model size (i.e., number of parameters): a large model will generate better responses, but will also require more memory and processor power. For a more complete discussion how LLM size affects response quality and hardware requirements, see the Optimizing Large Language Models with the OpenVINO™ Toolkit paper.

4. How to use RAG with OpenVINO™ and LangChain

A RAG pipeline and chatbot can be set up using LangChain and OpenVINO™. OpenVINO compresses the embedding model, rerank model, and LLM and provides an optimized backend for executing inference. LangChain provides tools and interfaces for orchestrating each stage of the pipeline. This section shows code snippets and explanations for setting up each portion of the RAG pipeline with OpenVINO™ and Langchain.

System Requirements
The primary requirement for running a RAG pipeline is to have enough system RAM available to hold the LLM, embedding, and rerank models in memory. The models used for the demo shown in this section require about 10GBytes of memory total. It is recommended to have at least 16GBytes RAM available. (If that much RAM isn't available, consider using the `tiny-llama-1b-chat` model in place of `zephyr-7b-beta-int8-ov`, as it requires much less memory.)

Prerequisites
Before running the code snippets shown in this paper, a Python 3.8 - 3.11 environment must be set up with the necessary dependencies installed.

1. Follow the steps in the OpenVINO™ Notebooks installation guide to create an environment named `openvino_env` and install the basic dependencies.
2. Run the following commands to install the other dependencies:
   ```
   pip install optimum-intel huggingface_hub[cli] nncf

   pip install datasets accelerate onnx einops transformers_stream_generator \
   transformers>=4.40 bitsandbytes faiss-cpu sentence_transformers \
   langchain>=0.2.0 langchain-community>=0.2.0 langchainhub unstructured \
   scikit-learn python-docx pypdf
   ```

4.1. OpenVINO™ Integration with LangChain

LangChain is a library and framework for developing applications around LLMs. It provides a variety of modules for enabling RAG pipelines. These include document loaders, which load the content of text files in many different formats (.pdf, .txt, .docx, .py, etc.), text splitters, which split documents into multiple reasonably-sized chunks, and much more. LangChain can be used to create "chains" or pipelines that automatically process an input query, search for relevant documentation, and assemble prompts that combine the query with context that can be used for generating an accurate response.

OpenVINO™ integrates with LangChain via interfaces that allow developers to load OpenVINO™ IR models and interact with them through standard LangChain APIs. In short,

LangChain provides the frontend API, while OpenVINO™ provides the backend execution framework.

There are three classes that enable OpenVINO models to be used with LangChain:

- OpenVINOEmbeddings: loads text embedding models in OpenVINO IR format
- OpenVINOReranker: loads rerank models in OpenVINO IR format
- HuggingFacePipeline: loads LLMs in OpenVINO IR format and executes them using OpenVINO as a backend framework

The code snippets in Sections 4.2 and 4.3 show how to use LangChain to set up a RAG pipeline with OpenVINO models.

4.2. Converting and Compressing LLM, Embedding, and Rerank Models

Three different deep learning language models are used in a RAG pipeline: the embedding model, rerank model, and LLM. OpenVINO™ supports several models from Hugging Face for each of these tasks. The Hugging Face models can be downloaded from Hugging Face and converted to OpenVINO IR format using Optimum Intel. Optimum can also use OpenVINO™ NNCF to perform weight compression on the models if desired.

This section provides code snippets and explanations for downloading an embedding model, rerank model, and LLM from Hugging Face, and then converting and compressing them using Optimum Intel.

4.2.1. Convert Embedding and Rerank Models with Optimum CLI

The embedding and rerank models can both be downloaded and converted using the Optimum command line interface. The following command shows how to export a model with optimum-cli.

Note: Make sure that the OpenVINO™ Notebooks environment has been set up and activated before running the commands below.

```
optimum-cli export openvino --model <model_id_or_path> --task <task> <out_dir>
```

In the command, the `--model` argument is a model ID from Hugging Face Hub (e.g., `tiny-llama-1b-chat`) or a path to a locally saved model. The `--task` argument specifies what task the model is intended for (e.g., `feature-extraction`).

There are two English options available for the text embedding model: BAAI/bge-small-en-v1.5 and BAAI/bge-large-en-v1.5. To learn more about the tradeoffs between the large and small

model, see Section 3.2 of this paper. To download and convert the small model, use the following command:

```
optimum-cli export openvino --model BAAI/bge-small-en-v1.5 --task feature-extraction
bge-small-en-v1.5
```

This will download the model from Hugging Face, convert it to OpenVINO™ IR format (`openvino_model.bin` and `openvino_model.xml`), and save it in a folder named "bge-small-en-v1.5".

Similarly, for the English rerank model, there are two options to choose from: BAAI/bge-reranker-base and BAAI/bge-reranker-large. To download and convert the large model, use:

```
optimum-cli export openvino --model BAAI/bge-reranker-large --task text-classification
bge-reranker-large
```

Again, this will download the model, convert it to OpenVINO™ IR format, and save it in a folder named "bge-reranker-large". Now the text embedding and rerank models are ready to be used in the RAG pipeline.

4.2.2. Convert and Compress LLM

The LLM forms the core of the RAG pipeline, taking the assembled input prompt and generating a response. OpenVINO™ supports most Causal LLMs from Hugging Face. These are some of the popular LLM options:

- zephyr-7b-beta
- tiny-llama-1b-chat
- llama-2-chat-7b (Access required)
- Meta-Llama-3-8B (Access required)
- mistral-7b

There are two options for preparing a LLM: downloading a pre-converted model from OpenVINO's HuggingFace repository, or converting and compressing a model manually.

Option 1. Download Pre-converted Model From HuggingFace
The OpenVINO™ HuggingFace repository has several popular LLMs that have been converted to OpenVINO™ IR format and compressed to FP16, INT8, or INT4. Visit the repository for a list of pre-converted LLMs. Here are some popular options:

- zephyr-7b-beta-int8-ov
- phi-2-int8-ov
- mixtral-8x7b-Instruct-v0.1-int4-ov

The main benefit to using pre-converted and pre-compressed models is that it doesn't require as much system RAM. Normally, to compress a model, the system must have enough RAM to support the full-size FP16 or FP32 model. For example, a 7-billion parameter model (like zephyr-7b-beta or llama-2-7b) requires 14GBytes or 28GBytes of RAM to load the model and compress it. If the same 7B model has already been compressed to INT8 or INT4, it only requires 7GB or 3.5GB to use. It also saves time in preparing and deploying the model.

Download a pre-converted model with the HuggingFace CLI using "`huggingface-cli download <model_name>`". For example, to download the zephyr-7b-beta-int8-ov model, use:

```
huggingface-cli download OpenVINO/zephyr-7b-beta-int8-ov --local_dir zephyr-7b-beta-int8-ov
```

This will download the OpenVINO™ IR model files and save them in a folder called "zephyr-7b-beta-int8-ov". The model is now ready to be used with a RAG application.

Option 2. Manually Convert and Compress Model

The Optimum command line interface can be used to convert LLMs from Hugging Face to OpenVINO™ IR format. Optimum also uses NNCF to perform FP16, INT8, or INT4 weight compression on a model to reduce its memory footprint and inference latency without a significant reduction in response quality. Use the `--weight-format` argument to specify the precision for compression. For more information about the benefits of weight compression in OpenVINO™, see the Optimizing Large Language Models with the OpenVINO™ Toolkit paper.

Use the following Optimum CLI command to download the `zephyr-7b-beta` model, convert it, and compress it to INT8:

```
optimum-cli export openvino --model HuggingFaceH4/zephyr-7b-beta --task text-generation-with-past --weight-format int8 zephyr-7b-beta/INT8_compressed_weights
```

The model will be downloaded, converted to OpenVINO™ IR format, and then compressed to INT8 using OpenVINO™'s NNCF module. The converted, compressed model will be saved in the "tiny-llama-1b-chat/INT8_compressed_weights" folder. It's now ready to be used in the RAG application.

4.2.3. Loading Models to Target Device

Next, the converted models need to be loaded onto target devices for inference. The target device is the processor that will perform inference computation. It can be a CPU, integrated GPU, or discrete GPU. OpenVINO™ flexibly compiles the model to run efficiently on the selected device. To check the devices available in the system, use "core.available_devices". In server-based applications, the models can be split across the server and the client. The text embedding model may be deployed on the client hardware rather than on the server hardware. The text embedding model doesn't take as much processing power and is suitable to run on consumer devices (such as Intel® Core™ CPUs). This saves server processing power by

distributing the text embedding across all client devices rather than performing it all on the server.

Load Embedding Model

The LangChain OpenVINOBgeEmbeddings module is used to load and initialize the embedding model into memory. The code snippet below shows how to do so. It selects CPU as the target device (though 'GPU' or 'AUTO' could also be used), sets configuration variables, and then initializes the model. It tests the model with a test string and prints the first three values of the resulting vector.

```
# Snippet 1: Loading a text embedding model to a target device and running a test
string
from langchain_community.embeddings import OpenVINOBgeEmbeddings

embedding_model_device = 'CPU' # or 'GPU' or 'AUTO'

embedding_model_name = 'bge-small-en-v1.5'
embedding_model_kwargs = {'device': embedding_model_device}
encode_kwargs = {
    "mean_pooling": False,
    "normalize_embeddings": False,
}

embedding = OpenVINOBgeEmbeddings(
    model_name_or_path=embedding_model_name,
    model_kwargs=embedding_model_kwargs,
    encode_kwargs=encode_kwargs,
)

text = "This is a test document."
embedding_result = embedding.embed_query(text)
print(embedding_result[:3])
```

When the code is finished, the "embedding_result" variable contains a 384-item-long vector that contains numerical values representing the semantic information of the string in the "text" variable.

Load Rerank Model

Next, the LangChain OpenVINOReranker module is used to load and initialize the reranker model. The code snippet below shows how to select the target device, set configuration parameters, and load the model. The “rerank_top_n” parameter controls how many document chunks will be returned by the reranker.

```
# Snippet 2: Loading a rerank model to a target device
from langchain_community.document_compressors.openvino_rerank import OpenVINOReranker

rerank_model_device = 'CPU' # or 'GPU' or 'AUTO'
rerank_model_name = 'bge-reranker-large'
rerank_top_n = 3

rerank_model_kwargs = {'device': rerank_model_device}

reranker = OpenVINOReranker(
    model_name_or_path=rerank_model_name,
    model_kwargs=rerank_model_kwargs,
    top_n=rerank_top_n,
)
```

After running this snippet, the rerank model is ready to sort a set of embedded text chunks based on their relevance to a user query. See Section 4.3.2. Retrieval Stage for more information on how to do so.

Load LLM

The converted and compressed LLM can be initialized using the HuggingFacePipeline class from LangChain. To deploy the model with OpenVINO™, specify `backend="openvino"` to set OpenVINO™ as the backend inference framework. The code snippet shows an example of how to load the `zephyr-7b-beta-int8-ov` model that was downloaded previously.

```
# Snippet 3: Loading LLM to target device and testing it
from langchain_community.llms.huggingface_pipeline import HuggingFacePipeline

llm_device = 'CPU' # or 'GPU' or 'AUTO'
llm_dir = 'zephyr-7b-beta-int8-ov'
ov_config = {'PERFORMANCE_HINT':'LATENCY', 'NUM_STREAMS':'1', 'CACHE_DIR':''}

llm = HuggingFacePipeline.from_model_id(
    model_id=llm_dir,
    task='text-generation',
    backend='openvino',
    model_kwargs={
        'device': llm_device,
        'ov_config': ov_config,
    },
    pipeline_kwargs={'max_new_tokens': 512}, # Sets the number of tokens to generate
in response
)

result = llm.invoke('The best Intel processor is')
print(result)
```

The snippet specifies the device, directory where the model weights are saved, and the OpenVINO™ configuration parameters. It initializes the model using the HuggingFacePipeline class. Finally, it tests the model by passing in a text string to generate a response to. The length of the response is dictated by the `max_new_tokens` parameter.

4.3. Setting up RAG Application

Now that the embedding model, rerank model, and LLM have been prepared, the rest of the RAG pipeline can be built around them. Most of the tasks involved with loading documents, vectorizing inputs, etc., are handled by LangChain. OpenVINO™ provides the execution backend for the deep learning models. This section provides code snippets showing how to set up various parts of the pipeline.

The RAG pipeline can be broken into two stages: the indexing stage and the retrieval and generation stage.

4.3.1. Indexing Stage

During the indexing stage, reference documentation is loaded, split into chunks, embedded into vector data, and indexed in memory storage.

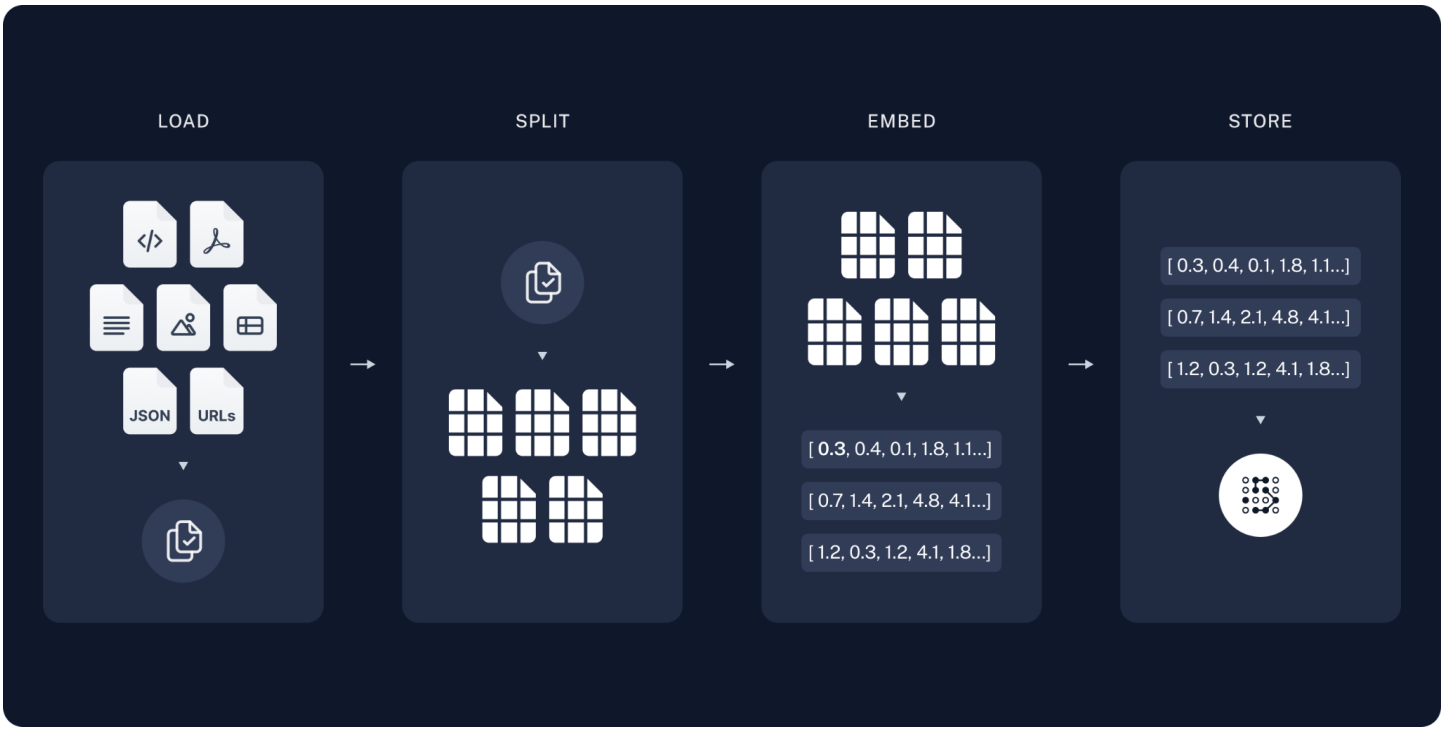

Figure 5. Loading, splitting, embedding, and storing documents during the indexing stage

Loading Documents

The first step of indexing is to load the reference documents into memory. LangChain has a document_loaders component that parses a specific document and loads the full content of the document into a data object inside the Python program. There are a variety of options for loading different kinds of documents: PDFs, text files, Word, PowerPoint, YouTube transcriptions, and many more. Once the documents are loaded, they can be converted to vectors using an embedding model.

Splitting Documents

Text splitters break up a document into chunks of text. The size of the chunks can be specified in the LangChain interface. The optimal chunk size is mainly determined by the size of the model's context window. There are many types of text splitters. They differ in what the text splits on (e.g. sentences, paragraphs, certain characters, code blocks). They also differ on how chunk size is measured and how much they allow chunks to overlap.

Embedding Documents

The Embeddings module in LangChain provides a standard interface for loading and inference with many different types of text embedding models. The base Embeddings class has methods for embedding a full document or a user query.

Indexing Embedded Documents

Retrievers will take a user query and search a stored document (both in vector space) to return chunks of text that are similar to the query.

Code Example

The following code snippet combines all of the above tasks into a basic Python script. It loads a document, splits it into chunks, embeds the chunks as a vector database, and indexes the vector database in a retriever.

```
# Snippet 4: Indexing stage
# Note: This snippet assumes "embedding" model has already been initialized (Snippet 1)

# Step 1 - Load document
from langchain.document_loaders import PyPDFLoader

doc_file_path = 'text_example_en.pdf'
loader = PyPDFLoader(doc_file_path)
loaded_doc = loader.load()

# Step 2 - Split document into text chunks
from langchain.text_splitter import RecursiveCharacterTextSplitter

text_splitter = RecursiveCharacterTextSplitter(chunk_size=400, chunk_overlap=50)
texts = text_splitter.split_documents(loaded_doc)

# Step 3 - Embed documents into vector database
from langchain_community.vectorstores import Chroma

vector_db = Chroma.from_documents(texts, embedding) #"embedding" model already initialized in Snippet 1

## Test on an example query - this should return the top chunks of text most relevant to the query
query = 'What is the Intel Core Ultra Processor?'
top_docs = vector_db.similarity_search(query)
print(top_docs[0])

# Step 4 - Set up a retriever for retrieving the document chunks most relevant to the query
vector_search_top_k = 10 # Return the top 10 document chunks
search_method = 'similarity'
search_kwargs = {"k": vector_search_top_k}
```

```
retriever = vector_db.as_retriever(search_kwargs=search_kwargs,
search_type=search_method)

## Test the retriever with the same query (it should have the same result as
top_docs[0])
results = retriever.invoke(query)
print('\n', results[0])
```

Now, the program can take a query and search the documentation for the chunks that are most relevant to that query. The retriever takes the user query string as an input and returns the top 10 most relevant chunks. In the retrieval stage, these chunks will be reranked, and the most relevant chunk will be combined with the prompt and passed to the LLM.

4.3.2. Retrieval and Generation Stage

The retrieval and generation stage of the RAG pipeline will take a user query, retrieve the chunks of documentation that are most relevant to the query, and rerank them. The top chunks of documentation are added to the input query and passed to the LLM. The LLM generates a response and outputs it to the user.

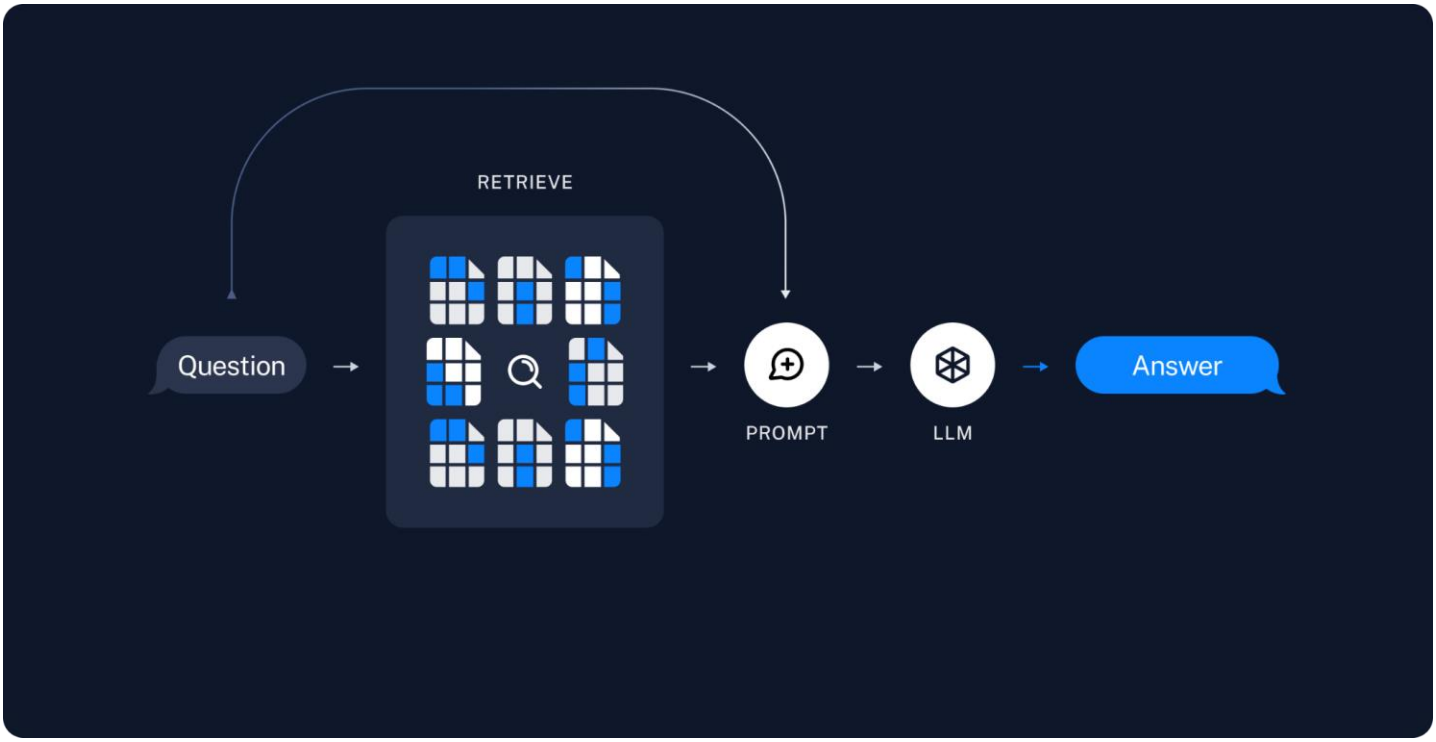

Figure 6. Retrieving relevant documentation, assembling a prompt, and generating an answer

Retrieve and Rerank Relevant Document Chunks

When a new query is input to the RAG pipeline, the retriever compares it against the chunks of documentation to determine which chunks contain information similar to the query. The retriever returns the “top k” chunks that are most similar (e.g., the top 10 chunks). The rerank model then processes the top k chunks using a deep learning algorithm to perform an even better sort of relevance. It returns the “top n” chunks which are most relevant to the query. The number of “top k” and “top n” chunks returned are configurable as parameters for the retriever and reranker.

In some cases, the rerank model winds up being redundant. It doesn’t change the order of chunks returned by the retriever because the retriever already did a good enough job sorting them. The reranker step can be skipped in these cases.

Rerank models that have been converted to OpenVINO™ IR format can be loaded using the OpenVINOReranker module. See the code snippet below for how to initialize and configure the rerank model.

Assemble Prompt Chain

Once the retriever and reranker have extracted the top chunks of documentation that are relevant to the query, the pipeline assembles a prompt. The prompt contains the user’s query and includes the documentation as context for the LLM to use in answering the query.

The prompt itself is defined through a prompt template. The template provides instructions (e.g., “Answer the following question based on the context below”) and indicates where the context and question should be inserted into the prompt. The LangChain PromptTemplate class automatically handles the insertion of the user’s query and the context from the retriever. The code snippet below provides a basic example of a prompt template.

Generate Response

Finally, when the full prompt is assembled, it is passed to the LLM to generate a response. The LangChain pipeline controls the text generation loop and constrains the response to a fixed number of tokens.

Code Example

While the steps described above are mostly straightforward, the code to execute the steps is a little less clear. LangChain creates high-level “chains” that automatically handle passing of the query to the retriever, the retriever to the ranker, the output of the reranker to the prompt template, and finally the prompt template to the LLM. Once the chain is constructed, the entire pipeline is executed in one line of code!

The code snippet below shows the basic steps of the retrieval stage. In Step 1, the reranker model is linked to the retriever using the ContextualCompressionRetriever class. This tells the retriever to automatically use the reranker after it has extracted relevant chunks of documentation.

In Step 2, the prompt template is defined and initialized using the PromptTemplate class. A chain (create_docs_chain) is created that links the prompt to the input of the LLM. Another chain (rag_chain) is created that links the output of the retriever/reranker to the input of the create_docs_chain. This sets up the chain so the document chunks extracted by the retriever/reranker are automatically inserted into the prompt template.

Finally, in Step 3, the pipeline is executed. The full chain is invoked with an input query. The query is used to find relevant context, and then both the context and query are inserted into the prompt and passed to the LLM. The LLM generates a response and prints it to the terminal.

```
# Snippet 5: Retrieval stage

# Step 1 - Set up reranker
from langchain.retrievers import ContextualCompressionRetriever
rerank_retriever = ContextualCompressionRetriever(base_compressor=reranker,
base_retriever=retriever)

# Step 2 - Assemble prompt using chains
## Set up prompt template
from langchain.prompts import PromptTemplate

template = """Answer the question based only on the following context:
{context}

Question: {input}
"""
prompt = PromptTemplate.from_template(template)

## Set up chain to pass prompt to the LLM
from langchain.chains.combine_documents import create_stuff_documents_chain
combine_docs_chain = create_stuff_documents_chain(llm, prompt)

## Set up chain to pass prompt (that has been filled with context from the retriever)
to the LLM
from langchain.chains import create_retrieval_chain
rag_chain = create_retrieval_chain(rerank_retriever, combine_docs_chain)

# Step 3 - Generate response
query = 'What is the Intel Core Ultra Processor?'
response = rag_chain.invoke({'input': query})
print(response["answer"])
```

When the code runs, it will display the prompt provided to the LLM (which includes the user query and the retrieved context) and the answer generated by the LLM.

The code snippets in this section are intended to provide a basic example showing how each stage of the RAG pipeline works. In practice, there are many factors to consider when creating a RAG application: which Document Loaders to use, the retriever search method, how many text chunks to extract, and much more. To see a professional end-to-end example that puts the full process together, visit the OpenVINO™ Notebooks RAG example, which is described in the next section.

5. OpenVINO™ RAG Examples

OpenVINO™ provides two exhaustive examples showing how to set up an RAG pipeline and chatbot interface, either locally or on a remote server.

5.1. OpenVINO™ Notebooks End-to-End RAG Pipeline

The llm-rag-langchain OpenVINO™ Notebook provides an end-to-end example of how to set up a RAG pipeline using LangChain. It uses Gradio to set up an interactive chatbot interface that lets users upload reference documents and submit queries to the pipeline. It shows how to keep track of the chatbot conversation and update the pipeline when new reference documents are uploaded.

A benefit of the notebook is that it allows developers to experiment with different parameters for the RAG pipeline. These include the models used, the level of weight compression, the inference device, the “top k” and “top n” parameters, the retrieval search method used, and much more. Developers can tweak these settings in the notebook to determine which pipeline configurations work best for their application.

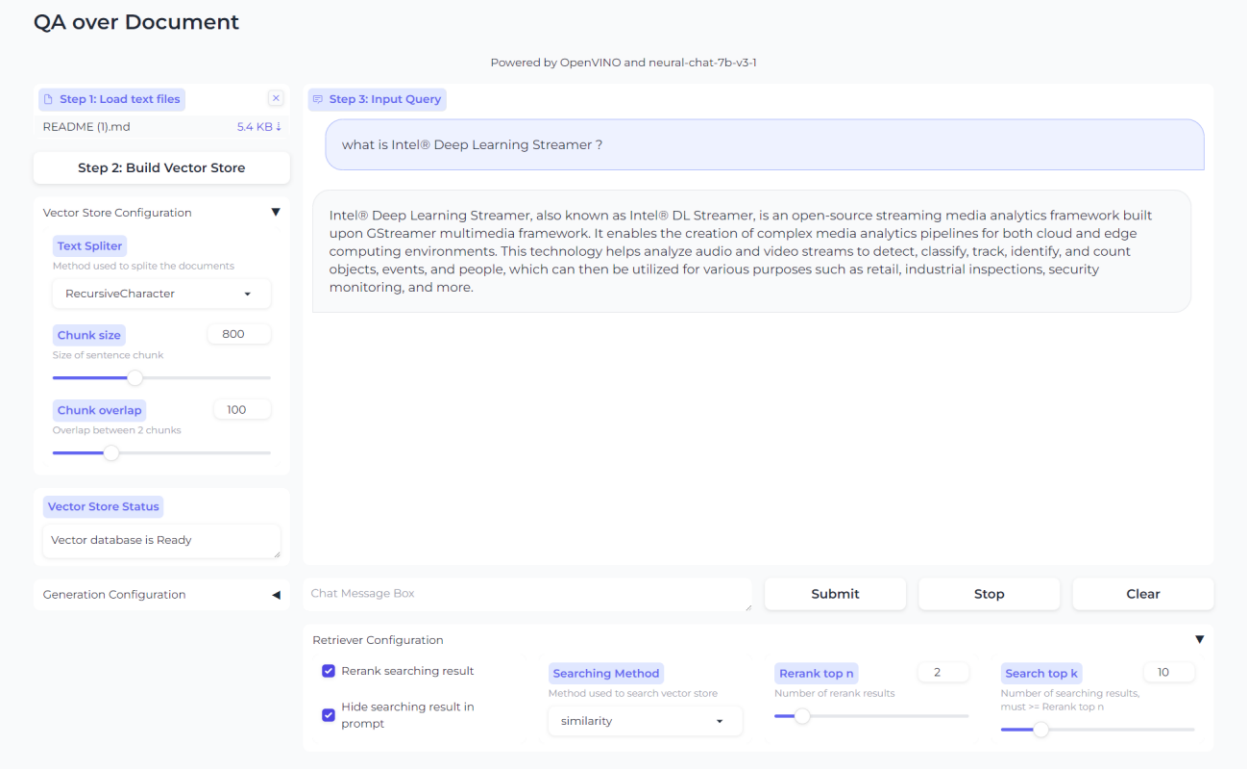

Figure 7. The OpenVINO™ Notebooks RAG example sets up an interactive chatbot interface using Gradio that also allows users to load the reference files that will be used for context.

The notebook walks through the process of installing prerequisites, converting and compressing models, and loading the models into memory for inference. It allows developers to experiment with different models and compile them on different devices (e.g., CPU, GPU) for inference. Finally, it sets up a fully-functional Gradio application for a RAG chatbot. The application runs in a web browser and does the following:

1. Performs text embedding on a specified set of reference documents, which includes:
    a. Loading the document into memory using DocumentLoader module from LangChain
    b. Splitting the document into smaller chunks using LangChain text splitters (e.g. CharacterTextSplitter)
    c. Converting the chunks to vectors and stores them in a vector database

2. Provides a chat interface for a user to submit a query. When a query is submitted, it retrieves relevant documentation based on the user query, and then generates an LLM prompt by adding the query to the relevant chunks of documentation text.

3. Generates a response to the input query and displays it in the chat window.

Figure 7 above shows an example of the Gradio app in action.

5.2. RAG Chatbot with OVMS

The RAG Demo with OpenVINO™ Model Server and LangChain example shows how to run a RAG pipeline with a LLM hosted on OVMS. It uses the OpenAI API to interact with the hosted LLM. It also implements state-of-the-art methods for continuous batching and paged attention to efficiently serve LLMs. These methods significantly increase the throughput of a hosted model, increasing the number of simultaneous users supported.

For more information about the benefits provided by OVMS when serving LLMs, see the white paper on Serving LLMs in OpenVINO™. The paper also provides benchmark data from running various models with OVMS.

6. Notices & Disclaimers

Intel technologies may require enabled hardware, software or service activation.

No product or component can be absolutely secure.

Your costs and results may vary.